Practical No. 1

Aim:- Write a program for tokenization of given input.

Introduction

Tokenization is the process of breaking down a stream of text into words, phrases, symbols, or other meaningful elements, known as tokens. These tokens are the basic units of the text and serve as the building blocks for various natural language processing (NLP) tasks such as text analysis, information retrieval, and machine translation.

A basic breakdown of the theory behind tokenization:

1. Text Input: The process starts with a piece of text, which can be a sentence, paragraph, or an entire document.

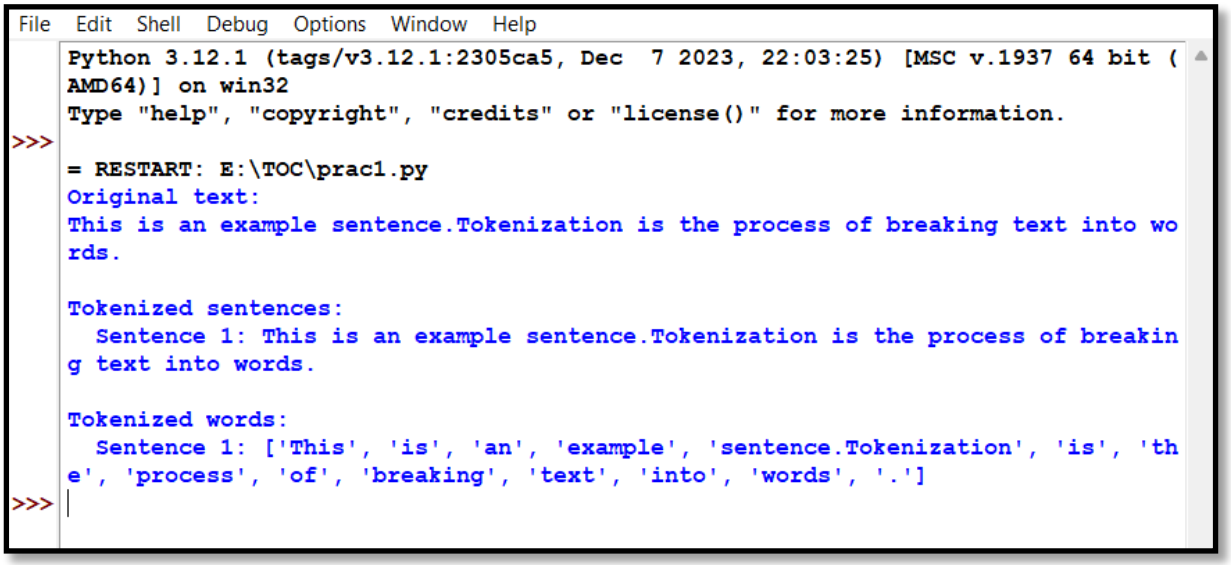
2. Tokenization Rules: Tokenization involves applying specific rules or patterns to segment the text into individual tokens. These rules can vary depending on the requirements of the task or the language being processed.

3.Token Types: Tokens can represent different linguistic units such as words, punctuation marks, numbers, or special characters. The choice of token types depends on the objectives of the analysis.

Code:

```
import nltk
from nltk.tokenize import word_tokenize, sent_tokenize

#nltk.download('punkt')  # Download the necessary data for tokenization

def tokenize_text(text):
    # Tokenize the text into sentences
    sentences = sent_tokenize(text)

    # Tokenize each sentence into words
    words = [word_tokenize(sentence) for sentence in sentences]

    return sentences, words

if __name__ == "__main__":
    # Example input text
    input_text = "This is an example sentence.Tokenization is the process of breaking text into
words."

    # Tokenize the input text
    sentences, words = tokenize_text(input_text)

    # Display the results
    print("Original text:")
    print(input_text)

    print("\nTokenized sentences:")
    for i, sentence in enumerate(sentences, start=1):
        print(f"  Sentence {i}: {sentence}")

    print("\nTokenized words:")
    for i, sentence_words in enumerate(words, start=1):
        print(f"  Sentence {i}: {sentence_words}")
```

Output:


File Edit Shell Debug Options Window Help
Python 3.12.1 (tags/v3.12.1:2305ca5, Dec 7 2023, 22:03:25) [MSC v.1937 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: E:\TOC\prac1.py
Original text:
This is an example sentence.Tokenization is the process of breaking text into words.
Tokenized sentences:
Sentence 1: This is an example sentence.Tokenization is the process of breaking text into words.
Tokenized words:
Sentence 1: ['This', 'is', 'an', 'example', 'sentence.Tokenization', 'is', 'the', 'process', 'of', 'breaking', 'text', 'into', 'words', '.']
>>>

Practical No. 2

Aim:- Write a program for generating regular expressions for regular grammar

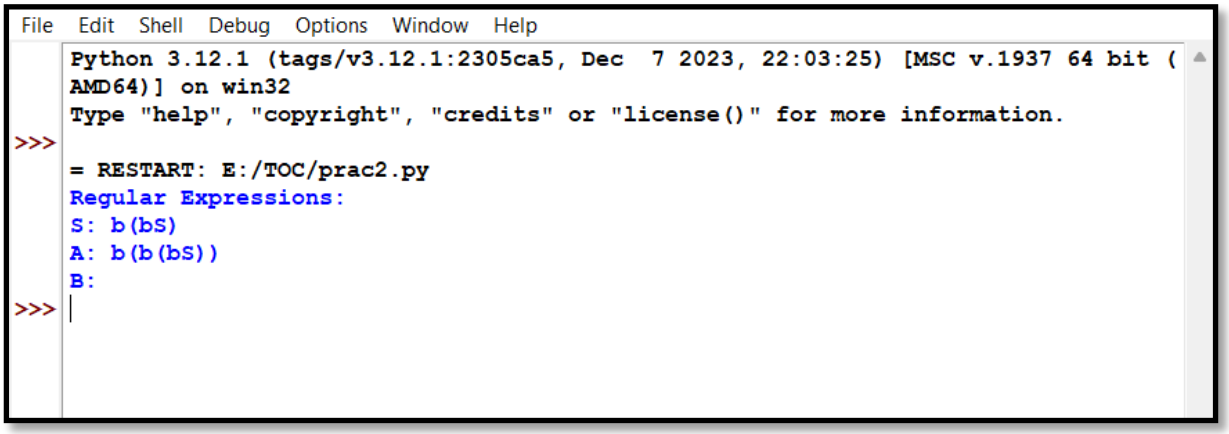
Introduction

1. Definition of Regular Grammar: Regular grammars consist of a set of terminal symbols (alphabet), non-terminal symbols (variables), a start symbol, and production rules. These rules specify how non-terminal symbols can be replaced by strings of terminal symbols.
2. Translation of Production Rules: Each production rule in the regular grammar is translated into a regular expression. This involves representing the possible strings generated by the rule using regex syntax.
3. Example Translation: For instance, if a production rule states that a non-terminal symbol "A" can produce either "aA" or "b", the corresponding regular expression for "A" would be "a*A | b". Here, "a*" denotes zero or more occurrences of "a", and "|" represents alternation between the two choices.
4. Combining Regular Expressions: If a non-terminal symbol has multiple production rules, the regular expressions corresponding to these rules are combined using alternation. This ensures that the regex captures all possible choices.
5. Handling Epsilon Productions: Epsilon (ε) productions, which generate the empty string, are accounted for by including the empty string or an optional expression (e.g., "?") in the regular expression.
6. Testing: After generating the regular expressions, it's crucial to test them against sample strings to verify that they accurately represent the language described by the regular grammar.
7. Utilization: Once validated, the generated regular expressions can be used for various tasks such as pattern matching, text searching, and string validation, providing a systematic way to process strings following specific syntactic rules.

Code:

```
import re

def generate_regex(grammar):
    regex_rules = {}

    # Convert each grammar rule to a regular expression
    for rule in grammar:
        non_terminal, production = rule.split(' -> ')
        if production[0] == production[-1] == "'":  # Handles terminals enclosed in single quotes
            regex_rules[non_terminal] = production[1:-1]
        elif production == 'ε':  # Handles epsilon production
            regex_rules[non_terminal] = ''
        else:
            regex_rules[non_terminal] = production

    # Replace non-terminals with their corresponding regular expressions
    for non_terminal, regex in regex_rules.items():
        for other_non_terminal in regex_rules:
            regex = re.sub(other_non_terminal, f'({regex_rules[other_non_terminal]})', regex)

        # Add the final regular expression for the non-terminal
        regex_rules[non_terminal] = regex

    return regex_rules

if __name__ == "__main__":
    # Example regular grammar
    regular_grammar = [
        "S -> aS",
        "S -> bA",
        "A -> aB",
        "A -> bS",
        "B -> bB",
        "B -> ε"
    ]

    # Generate regular expressions
    regex_rules = generate_regex(regular_grammar)

    # Display the results
    print("Regular Expressions:")
    for non_terminal, regex in regex_rules.items():
        print(f"{non_terminal}: {regex}")
```

Output:


File Edit Shell Debug Options Window Help
Python 3.12.1 (tags/v3.12.1:2305ca5, Dec 7 2023, 22:03:25) [MSC v.1937 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: E:/TOC/prac2.py
Regular Expressions:
S: b(bS)
A: b(b(bS))
B:
>>>

Practical No. 3

Aim:- Design a Program for creating machine that accepts three consecutive one.

Introduction

1 Objective: Design a finite automaton (FA) or regular expression that recognizes strings containing three consecutive occurrences of the digit "1".

2 Finite Automaton (FA):
- Define states representing different stages of processing the input string.
- Transition between states based on encountering "0" or "1".
- Include a special state to denote acceptance when three consecutive "1"s are encountered.

3 Regular Expression (Regex):
- Construct a regex pattern to match strings with three consecutive "1"s.
- Utilize repetition operators like "{3}" to indicate exactly three occurrences.
- Design patterns to handle possible combinations of "0"s and "1"s before and after the consecutive "1"s.

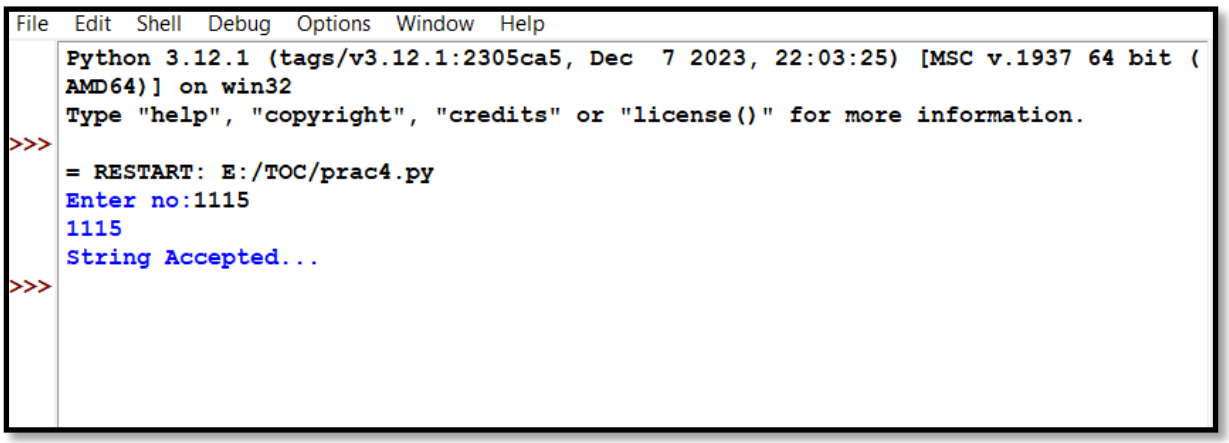
4 Example Regex:
- Pattern: ".*111.*"
- Explanation: Matches any string (".*") containing three consecutive "1"s ("111") surrounded by any number of characters.

5 Testing:
- Validate the FA or regex against sample strings to ensure correct recognition of strings with three consecutive "1"s.
- Test various scenarios to confirm robustness and accuracy.

Code:

```
f=0;
n=input("Enter no:")
print(n)
for j in range(len(n)-2):
    if n[j]=='1' and n[j] == n[j + 1] and n[j + 1] == n[j + 2]:
        f=1

if f==1:
    print("String Accepted...")
else:
    print("String not Accepted...")
```

Output:


File Edit Shell Debug Options Window Help
Python 3.12.1 (tags/v3.12.1:2305ca5, Dec 7 2023, 22:03:25) [MSC v.1937 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: E:/TOC/prac4.py
Enter no:1115
1115
String Accepted...
>>>

Practical No. 4

Aim:- Design a Program for creating machine that accepts the string always ending with 101.

Introduction

1 Objective: Design a finite automaton (FA) or regular expression to recognize strings that always end with the pattern "101".

2 Finite Automaton (FA):
- Define states representing different stages of processing the input string.
- Transition between states based on encountering "0" or "1".
- Designate a final state that is reached only when the input ends with "101".

3 Regular Expression (Regex):
- Construct a regex pattern to match strings ending with "101".
- Use anchors like "$" to signify the end of the string and ensure "101" appears at the end.

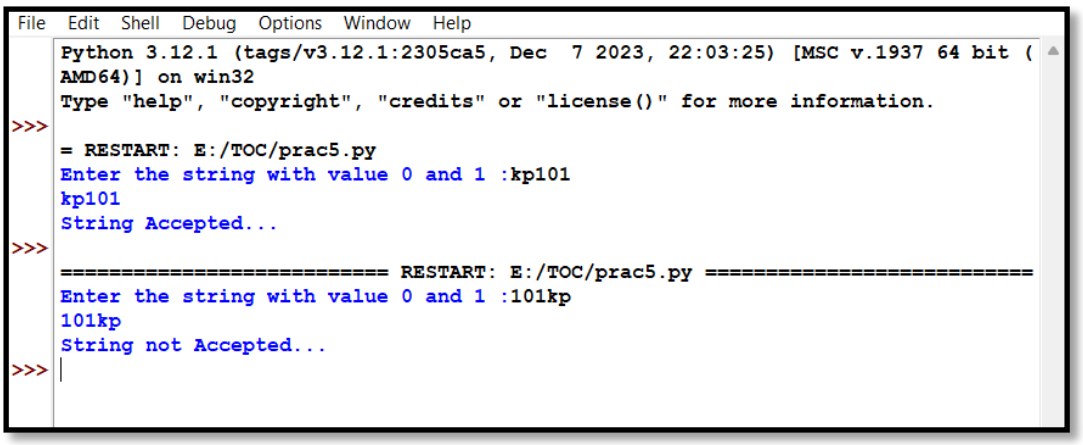
4 Example Regex:
- Pattern: ".*101$"
- Explanation: Matches any string (".*") ending with "101" at the end of the line.

5 Testing:
- Validate the FA or regex against sample strings to ensure correct recognition of strings ending with "101".
- Test various scenarios to confirm robustness and accuracy.

Code:

```
f=0;
n=input("Enter the string with value 0 and 1 :")
print(n)
for j in range(len(n)):
    if n[-1]=='1' and n[-2] =='0'and n[-3] =='1':
        f=1

if f==1:
    print("String Accepted...")
else:
    print("String not Accepted...")
```

Output:


File Edit Shell Debug Options Window Help
Python 3.12.1 (tags/v3.12.1:2305ca5, Dec 7 2023, 22:03:25) [MSC v.1937 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: E:/TOC/prac5.py
Enter the string with value 0 and 1 :kp101
kp101
String Accepted...
>>>
========== RESTART: E:/TOC/prac5.py ==========
Enter the string with value 0 and 1 :101kp
101kp
String not Accepted...
>>>

Practical No. 5

Aim:- Design a program for accepting decimal number divisible by 2.

Introduction

1 Objective: Design a finite automaton (FA) or regular expression to recognize decimal numbers that are divisible by 2.

2 Finite Automaton (FA):
- Define states representing different remainders when dividing by 2.
- Transition between states based on encountering digits from 0 to 9.
- Designate a final state that is reached only when the input ends with a digit representing an even number.

3 Regular Expression (Regex):
- Construct a regex pattern to match decimal numbers divisible by 2.
- Utilize regex syntax to represent even numbers, such as alternation for the last digit being 0, 2, 4, 6, or 8.

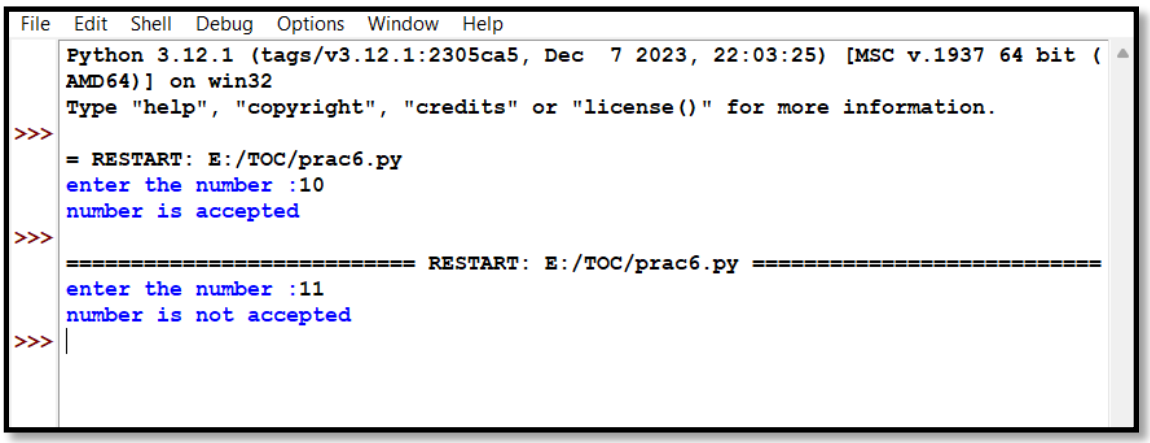
4 Example Regex:
- Pattern: "^[0-9]*[02468]$"
- Explanation: Matches any string consisting of digits (represented by "[0-9]*") ending with an even digit (represented by "[02468]").

5 Testing:
- Validate the FA or regex against sample numbers to ensure correct recognition of numbers divisible by 2.
- Test various scenarios, including edge cases, to confirm accuracy and reliability

Code:

```
n=int(input("enter the number"))
if(n%2==0):
    print("number is accepted")
else:
    print("number is not accepted ")
```

Output:


File Edit Shell Debug Options Window Help
Python 3.12.1 (tags/v3.12.1:2305ca5, Dec 7 2023, 22:03:25) [MSC v.1937 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: E:/TOC/prac6.py
enter the number :10
number is accepted
>>>
========== RESTART: E:/TOC/prac6.py ==========
enter the number :11
number is not accepted
>>>

Practical No. 6

Aim:- Design a program for creating a machine which accepts string having equal no. of 1's and 0's.

Introduction

1 Objective: Design a finite automaton (FA) or regular expression to recognize strings with an equal number of "1"s and "0"s.

2 Finite Automaton (FA):
- Define states representing different counts of "1"s and "0"s encountered in the input string.
- Transition between states based on encountering "1" or "0".
- Designate a final state that is reached only when the counts of "1"s and "0"s are equal.

3 Regular Expression (Regex):
- Construct a regex pattern to match strings with an equal number of "1"s and "0"s.
- Utilize regex syntax to ensure that the number of occurrences of "1"s and "0"s are the same.

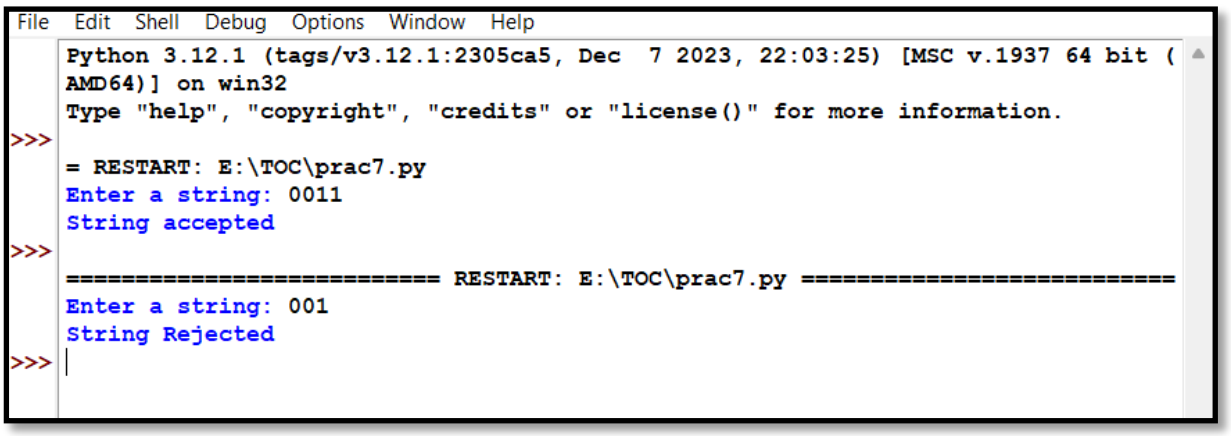
4 Example Regex:
- Pattern: "^(01)*$|^0*$|^1*$"
- Explanation: Matches strings consisting of an equal number of "01" pairs, or only "0"s, or only "1"s.

5 Testing:
- Validate the FA or regex against sample strings to ensure correct recognition of strings with equal counts of "1"s and "0"s.
- Test various scenarios, including strings of different lengths and edge cases, to confirm accuracy and reliability.

Code:

```
str = input("Enter a string: ")

ones = 0
zeroes = 0

for i in range (len(str)):
    if (str[i]) == '1' :
        ones+=1
    elif (str[i] == '0'):
          zeroes += 1

if(ones == zeroes):
    print ("String accepted")
else:
    print ("String Rejected")
```

Output:


File Edit Shell Debug Options Window Help
Python 3.12.1 (tags/v3.12.1:2305ca5, Dec 7 2023, 22:03:25) [MSC v.1937 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: E:\TOC\prac7.py
Enter a string: 0011
String accepted
>>>
========== RESTART: E:\TOC\prac7.py ==========
Enter a string: 001
String Rejected
>>>

Practical No. 7

Aim:- Design a program for creating a machine which count number of 1’s and 0’s in a given string.

Introduction

Here's the information in a point-wise format:

1 Objective: Design a finite automaton (FA) or algorithm to count the number of occurrences of "1"s and "0"s in a given string.

2 Finite Automaton (FA):
- Define states representing different counts of "1"s and "0"s encountered in the input string.
- Transition between states based on encountering "1" or "0".
- Designate a final state to mark the end of the string and output the counts of "1"s and "0"s.

3 Algorithm:
- Initialize counters for "1"s and "0"s.
- Iterate through each character in the input string.
- Increment the respective counter when encountering a "1" or "0".
- Output the counts of "1"s and "0"s after processing the entire string.

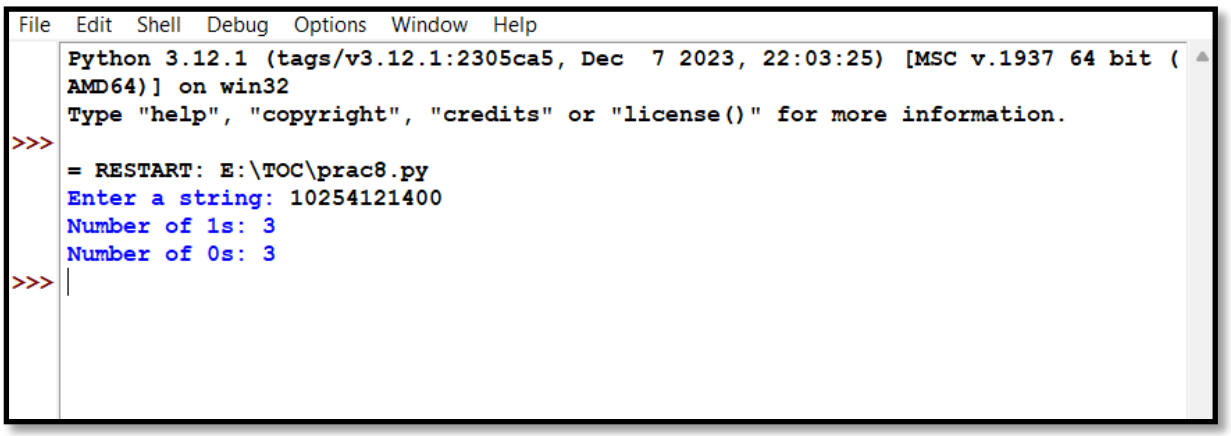
4 Example:
- Input: "110101"
- Output: Number of "1"s = 4, Number of "0"s = 2

5 Testing:
- Validate the FA or algorithm against sample strings to ensure correct counting of "1"s and "0"s.
- Test various scenarios, including strings of different lengths and compositions, to confirm accuracy and reliability.

Code:

```
str = input("Enter a string: ")

count_1 = 0
count_0 = 0

for char in str:
    if char == '1':
        count_1 += 1
    elif char == '0':
        count_0 +=1

print(f"Number of 1s: {count_1}")
print(f"Number of 0s: {count_0}")
```

Output:


File Edit Shell Debug Options Window Help
Python 3.12.1 (tags/v3.12.1:2305ca5, Dec 7 2023, 22:03:25) [MSC v.1937 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: E:\TOC\prac8.py
Enter a string: 10254121400
Number of 1s: 3
Number of 0s: 3
>>>

Practical No. 8

Aim:- Introduction to Turin Machin.

5.1 Introduction to Turing Machine

- In 1936, Mr. Alan Turing introduced advanced mathematical model for modern digital computer which was known as Turing machine. Turing machine is advanced machine of FA and PDA.
- This simple mathematical model has no difference between input and output set. This mathematical model can be constructed to accept a given language or to carry out some algorithm.
- This model sometimes uses it's own output as input for further computation.
- This machine or model can select current location and also decides location of memory by moving left or right. This mathematical model known as Turing machine has become an effective procedure.

- In short Turing machine is equal to,

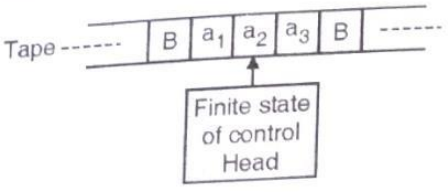
TM = FA + Tape

FA = Finite Automata

Tape

1. Infinite memory unit
2. Infinite cell (Each cell contains only one alphabet at one time)
3. Head of records move left or right.

- Turing machine's mathematical model consists of **Head** (moves right or left or stays in position), **infinite tape** and **finite set of states.**



Example 5.2.3 : Design Turing machine that accepts $\{a^n b^n \mid n \geq 1\}$

Solution :

- L = {ab, aabb, aaabbb....}
- 1st scan leftmost a and make it x using state q_0.
- State q_1 is used for scan b, change b to y and move backward
- Find rightmost X and repeat above steps in all characters change
- Check whether there are extra characters or not.

Turing machine is given below :

$$U = \{q_0, q_1, q_2, q_{chk}, q_{acc}\}$$

$$\Sigma = \{a, b\}$$

$$\Gamma = \{a, b, x, y, B\}$$

$$\delta = \text{Transition function}$$

Transition table

State	a	b	x	Y	B
q_0	(q_1, x, R)	-	-	(q_{chk}, y, R)	-
q_1	(q_1, a, R)	(q_2, y, L)	-	(q_1, y, R)	-
q_2	(q_2, a, L)	-	(q_0, x, R)	(q_2, y, L)	-
q_{chk}	-	-	-	(q_{chk}, y, R)	q_{acc}
q_{acc}	-	-	-	-	Final

Make it clear by transition diagram

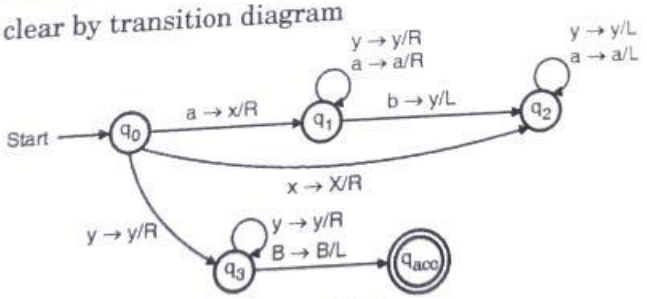


Fig. P. 5.2.3

Diagram explains that :

1. **State q_0 takes initiative to replace leftmost a by x and change the state to q_1.**

2. Function of q_1 is to search rightmost a's and y's until it finds leftmost b. If machine finds b, it changes it to y and enters state q_2.

3. State q_2 proceeds searching left for x and enters state q_0 upon finding it, move right to change a and the state.

4. All a's and b's are changes to x and y then q_{4bb} condition is satisfied. If B or x is encountered before b, then input is rejected either there is existence of a is moré than b or input $a^* b^*$.

5. From q_0 scanning y state q_3 is entered to scan over y's and check for number of b's remaining. If the y's are followed by a B, state q_{acc} is accepted otherwise rejected.

ID for aa bb → BaabbBB

→ Bq_0 aa bb BB

→ B x q_1 abb BB → xa q_1, bb BB → x q_2 ay b BB

→ B q_2 x a yb → Bx q_0 a y b BB → B xx q_1 yb BB

→ Bxx y q_1 b BB → B xx q_2 yy BB → B x q_2 x yy BB

→ Bxx q_0 yyBB → Bxx y q_3 yBB → B xx yy q_3 BB

→ Bxx yy B q_{acc} → q_{accept} → Final state

→ aabb is accepted.

Example 5.2.4 : Design Turing machine for language, $L = \{a^m b^n \mid m > n \times n > 0\}$

Solution :

$L = \{aab, aaab, aaaab, \ldots aaabb, aaaabb\}$

Turing machine,

$Q = \{q_0, q_1, q_2, q_{chk}, q_{acc}\}$

$\Sigma = (a, \{b\})$

$\Gamma = \{a, b, x, y, B\}$

δ = Transition function

Steps in this construction can be written as,

1. M starts in state q_0
2. If symbol a is read, then state q_0 changes to q_1 and replace symbol by x and move towards right then check for 1st occurrence of b, skip a and y.
3. b is replaced by y and enter the state q_2 and move towards left to search for x then move to right.
4. If symbol is b, then repeat above step, if symbol x is found out then move towards left until current symbol is not equal to a.
5. If current symbol is a that means there is at least one a is extra, then check blank on left side of the string.

Transition table

	a	b	x	y	B
q_0	(q_1, x, R)	-	-	-	-
q_1	(q_1, a, R)	(q_2, y, L)	-	(q_1, y, R)	(q_{chk}, y, L)
q_2	(q_2, a, L)	-	(q_0, x, R)	(q_2, y, L)	-
q_{chk}	(q_{chk}, a, L)	-	(q_{chk}, x, L)	(q_{chk}, y, L)	q_{acc}
q_{acc}	-	-	-	-	Final state

ID for aaab $\rightarrow$ BaaabBB

$\rightarrow B q_0 aaa b BB \rightarrow B x q_1 aa bB \rightarrow B x a q_1 a b B$

$\rightarrow B x aa q_1 b B \rightarrow B x aa q_2 y B \rightarrow x a q_2 a y B$

$\rightarrow B x q_2 a a y B \rightarrow B x q_0 aa y B \rightarrow B xx q_1 a y B$

$\rightarrow B xx a q_1 y B \rightarrow B xx a y q_1 B \rightarrow Bxx a y q_{chk} B$

$\rightarrow Bxx a q_{ch} k B$

$\rightarrow B xx q_{chk} ay \rightarrow Bx q_{chk} x a y B$

$\rightarrow B q_{chk} xx ay B \rightarrow q_{accept}$