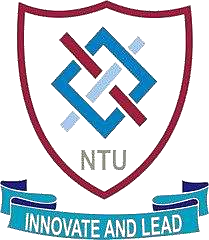
**Project Report: Source Code Analyzer and Summarizer Tool**



**Group 03 Members**

**Abdul Rafay**

21-NTU-CS-1197

**Arooba Zaman**

21-NTU-CS-1211

**Department of Computer Science**

National Textile University Faisalabad **27-05-2024**

Table of Contents

[**Problem Statement** 3](#_Toc167628011)

[**Introduction** 4](#_Toc167628012)

[**Purpose** 4](#_Toc167628013)

[**Project Components** 4](#_Toc167628014)

[**1. Tokenizer** 4](#_Toc167628015)

[**2. Categorizer** 4](#_Toc167628016)

[**3. Visualization** 4](#_Toc167628017)

[**4. Input Options** 4](#_Toc167628018)

[**5. Keyword Regex** 4](#_Toc167628019)

[**6. Operators Regex** 5](#_Toc167628020)

[**7. Identifiers and Comments** 5](#_Toc167628021)

[**8. Numeric and String Literals** 5](#_Toc167628022)

[**9. Helper Functions** 5](#_Toc167628023)

[**10. Main Function** 5](#_Toc167628024)

[**Libraries Used** 5](#_Toc167628025)

[**1)** **.re library** 5](#_Toc167628026)

[**2)** **matplotlib.pyplot** 5](#_Toc167628027)

[**3)** **keyword library** 5](#_Toc167628028)

[**Usage** 6](#_Toc167628029)

[**Applications** 6](#_Toc167628030)

[**1. Code Review** 6](#_Toc167628031)

[**2. Debugging** 6](#_Toc167628032)

[**3. Educational** 6](#_Toc167628033)

[**4. Static Code Analysis** 6](#_Toc167628034)

[**5. Code Metrics** 6](#_Toc167628035)

[**6. Refactoring** 6](#_Toc167628036)

[**7. Documentation** 7](#_Toc167628037)

[**Code Implementation** 7](#_Toc167628038)

[**Output:** 14](#_Toc167628039)

[**Visual Output:** 18](#_Toc167628040)

[**Conclusion** 19](#_Toc167628041)

[**References** 19](#_Toc167628042)

List of Figures

[**Figure 1** 18](#_Toc167464167)

[**Figure 2** 19](#_Toc167464168)

# **Problem Statement**

Develop a mini project that takes source program as input from user and distribute this source program into summarize information like keywords, relation operator, comments, identifiers etc.

# **Introduction**

This project involves creating a Python tool that analyzes source code and categorizes its components into keywords, relational operators, comments, identifiers, logical operators, airthematic operators, bitwise operators and other elements. The tool uses regular expressions to tokenize the input code and then sorts these tokens into their respective categories. It also visualizes the token distribution and frequency using pie charts and histograms.

# **Purpose**

The purpose of this project is to provide a comprehensive analysis of Python source code by breaking it down into key components. This helps in understanding the structure and usage of different elements within the code, making it easier to review and debug. It also provides a visual representation of the token distribution to help identify patterns and potential issues.

# **Project Components**

## **1. Tokenizer**

* Uses Regular expressions split source code into individual tokens by identifying patterns such as keywords, operators, identifiers, and literals. This process helps in lexical analysis, parsing, or syntax highlighting. Pythons re module provides tools to define and match these patterns efficiently.

## **2. Categorizer**

* Classifies tokens into categories like keywords, relational operators, comments, identifiers, logical operators, arithmetic operators, bitwise operators, numeric literals, string literals, and others.

## **3. Visualization**

* Provides pie charts and histograms to represent the distribution and frequency of different token categories.

## **4. Input Options**

* Allows users to choose between a sample program, custom input, or a file path to provide the source code.

## **5. Keyword Regex**

* Dynamically constructs a regex pattern for Python keywords, including additional ones like print and input.

## **6. Operators Regex**

* Constructs regex patterns for relational, logical, arithmetic, and bitwise operators.

## **7. Identifiers and Comments**

* Uses regex patterns to identify valid Python identifiers and comments by matching specific patterns. Identifiers are matched using r'\b[a-zA-Z\_] [a-zA-Z0-9\_] \*\b' and comments with r'#.\*'. This helps in parsing and analyzing Python code for various purposes.

## **8. Numeric and String Literals**

* Identifies numeric literals and string literals using regex patterns.

## **9. Helper Functions**

* Includes functions to check for complete string literals and handle partial tokens.

## **10. Main Function**

* Integrates all components and handles user input, tokenization, categorization, and visualization.

# **Libraries Used**

## **.re library**

* 1. **Purpose:** The re module provides support for regular expressions in Python. It is used to search, match, and manipulate strings based on regex patterns.
  2. **Usage in Code:** This module is used extensively to create regex patterns for tokenizing the source code and categorizing the tokens into different types like keywords, operators, identifiers, comments, numeric literals, and string literals.

## **matplotlib.pyplot**

* 1. **Purpose:** The matplotlib.pyplot module is a collection of functions that make matplotlib work like MATLAB. It is used for plotting and visualizing data.
  2. **Usage in Code:** This module is used to create visualizations of the token distribution in the form of pie charts and histograms, helping to understand the composition of the source code.

## **keyword library**

* 1. **Purpose:** The keyword module in Python provides a list of all the reserved keywords used in the language. It allows you to check if a string is a keyword and to retrieve the list of keywords. This module is useful for ensuring that variable names do not conflict with Python's reserved keywords.
  2. **Usage in Code:** This module is used to obtain the list of Python keywords, which are then included in the regular expression to identify keywords within the source.

# **Usage**

To use the tool, run the script and choose one of the input options. The script will then tokenize and categorize the source code, displaying the categorized tokens and their counts. Finally, it will generate visualizations to show the distribution of different token categories.

# **Applications**

## **1. Code Review**

* Helps reviewers understand the structure and key components of the source code.

## **2. Debugging**

* Identifies and categorizes different elements, making it easier to pinpoint issues.

## **3. Educational**

* Aids in teaching programming by showing how code is broken down into fundamental elements.

## **4. Static Code Analysis**

* Can be integrated into static code analysis tools to provide more detailed insights.

## **5. Code Metrics**

* Provides metrics on the usage of different tokens within the source code.

## **6. Refactoring**

* Assists in identifying parts of the code that may need refactoring by showing token distribution.

## **7. Documentation**

* Generates detailed information about the code, which can be used in documentation.

# **Code Implementation**

import re

from collections import Counter

import matplotlib.pyplot as plt

import keyword  # Import the keyword module

python\_keywords = keyword.kwlist

# Additional words to include in the keyword regex

extra\_keywords = ["print", "input"]

# Combine the existing Python keywords with additional ones

all\_keywords = python\_keywords + extra\_keywords

# Create a regular expression for all keywords

keywords\_regex = r'\b(?:' + '|'.join(all\_keywords) + r')\b'

# Define relational operators in Python

relational\_operators = ['==', '!=', '>', '<', '>=', '<=']  # All relational operators in Python

relation\_operators\_regex = r'|'.join([re.escape(op) for op in relational\_operators])  # Create a regex

# Define logical operators in Python

logical\_operators = ['and', 'or', 'not']  # All logical operators in Python

logical\_operators\_regex = r'\b(?:' + '|'.join(logical\_operators) + r')\b'  # Create a regex with word boundaries

# Define valid Python identifiers

identifiers\_regex = r'\b[a-zA-Z\_][a-zA-Z0-9\_]\*\b'

# Define regex for comments in Python

comments\_regex = r'#.\*'  # Match everything from '#' to the end of the line

# Define regex for numeric literals (integers and floats)

numeric\_literals\_regex = r'\b\d+(\.\d+)?\b'

numeric\_split\_regex = re.compile(r'(\d+(\.\d+)?)')  # Matches integers and floats

# Define regex for string literals (single and double-quoted strings)

string\_literals\_regex = r"['\"].\*?['\"]"

# Define regex for operators (arithmetic, assignment, bitwise)

arithmetic\_operators\_regex = r'[-+\*/%=]'

bitwise\_operators\_regex = r'[&|^~]'

# Function to tokenize the source code

def tokenize\_source\_code(source\_code):

    # Tokenize the source code

    tokens = re.findall(rf'{keywords\_regex}|{relation\_operators\_regex}|{logical\_operators\_regex}|{identifiers\_regex}|{comments\_regex}|{string\_literals\_regex}|[^\s]+', source\_code)

    return tokens

# Function to categorize tokens

def categorize\_tokens(tokens):

    categories = {

    'Keywords': [],

    'Relational Operators': [],

    'Comments': [],

    'Identifiers': [],

    'Logical Operators': [],

    'Arithmetic Operators': [],

    'Bitwise Operators': [],

    'Numeric Literals': [],

    'String Literals': [],

    'Others': []

    }

    # Helper function to check if a token is a complete quoted string

    def is\_complete\_string\_literal(token):

        return (

            token[0] == token[-1]

            and token[0] in ["'", '"']

            and len(token) > 1

        )

    # Helper function to check if a token contains incomplete string

    def extract\_string\_parts(token):

        # Find the opening and closing quotes

        if token[0] in ["'", '"']:

            closing\_pos = token.find(token[0], 1)  # Find the corresponding closing quote

            if closing\_pos > 1:

                string\_literal = token[:closing\_pos + 1]

                remaining\_part = token[closing\_pos + 1:]  # Everything after the closing quote

                return string\_literal, remaining\_part

        return None, None

    for token in tokens:

        if re.match(keywords\_regex, token):  # Use the dynamic regex for Python keywords

            categories['Keywords'].append(token)

        elif re.match(relation\_operators\_regex, token):

            categories['Relational Operators'].append(token)

        elif re.match(comments\_regex, token):

            categories['Comments'].append(token)

        elif re.match(identifiers\_regex, token):

            categories['Identifiers'].append(token)

        elif re.match(logical\_operators\_regex, token):

            categories['Logical Operators'].append(token)

        elif re.match(arithmetic\_operators\_regex, token):

            categories['Arithmetic Operators'].append(token)

        elif re.match(bitwise\_operators\_regex, token):

            categories['Bitwise Operators'].append(token)

        # Check if the token contains a numeric literal with additional characters

        elif re.match(numeric\_literals\_regex, token):

            # Extract the numeric part and separate the rest

            match = numeric\_split\_regex.search(token)

            if match:

                numeric\_part = match.group(1)

                non\_numeric\_part = token[len(numeric\_part):]  # Remaining part after the number

                categories['Numeric Literals'].append(numeric\_part)

                if non\_numeric\_part:

                    categories['Others'].append(non\_numeric\_part)  # Add the non-numeric part to 'Others'

        elif is\_complete\_string\_literal(token):

            # If it's a complete string literal, add it to 'String Literals'

            categories['String Literals'].append(token)

        else:

            # If it's a partial or other token, check for split cases

            string\_literal, remaining\_part = extract\_string\_parts(token)

            if string\_literal:

                categories['String Literals'].append(string\_literal)  # Add the string literal

                if remaining\_part:

                    categories['Others'].append(remaining\_part)  # Add anything left as 'Others'

            else:

                categories['Others'].append(token)  # Default to 'Others' if no clear match

    return categories

# Function to plot token distribution

def plot\_token\_distribution(categories):

    labels = categories.keys()

    sizes = [len(tokens) for tokens in categories.values()]

    plt.figure(figsize=(8, 6))

    plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=140)

    plt.axis('equal')

    plt.title('Token Distribution')

    plt.show()

# Function to plot a histogram

def plot\_token\_histogram(categories):

    labels = list(categories.keys())  # Get the category names

    counts = [len(tokens) for tokens in categories.values()]  # Count of tokens in each category

    plt.figure(figsize=(8, 6))  # Set figure size

    plt.bar(labels, counts)  # Plot the histogram

    # Set x and y axis labels

    plt.xlabel('Token Categories', fontsize=12)

    plt.ylabel('Number of Tokens', fontsize=12)

    # Rotate x-axis labels for better readability

    plt.xticks(rotation=90, fontsize=10)  # Rotate labels by 90 degrees (vertical)

    # Set the title for the histogram

    plt.title('Histogram of Token Occurrences', fontsize=14)

    plt.show()  # Display the histogram

# Main function

def main():

    # Ask user to choose between the sample program or custom input

    print("\t\t\tChoose an option:")

    print("===== Press 1 to Use the sample Python program ======")

    print("===== Press 2 to Enter your own source code ======")

    print("=== Press 3 to Enter path of your source program file ===")

    choice = int(input("Enter your choice (1 or 2 or 3): "))

    # Provide a default sample program if choice is 1

    if choice == 1:

        source\_code = """

        # This is a sample Python program

        def greet(name):

            print( "Hello" + name + "!" )

            # End of the function

        # Main code

        if \_\_name\_\_ == "\_\_main\_\_":

            greet( "World" )

        """

        print("\n\nUser Selection: Sample Program\n\n",source\_code,"\n")

    # Otherwise, take custom source code input from the user

    elif choice == 2:

        source\_code = input("Enter your source code:\n")

        print("\n\nUser Selection: Provide Program in input that is\n\n",source\_code,"\n")

    elif choice == 3:

        source\_code\_file\_path = input("Please enter the path to your source code file: ")

        with open(source\_code\_file\_path, 'r') as file:

            source\_code = file.read()

        print("\n\nUser Selection: Provide Program file\n\n",source\_code,"\n")

    # Tokenize the source code

    tokens = tokenize\_source\_code(source\_code)

    # Categorize tokens

    categories = categorize\_tokens(tokens)

    print("\*\*\*\*\*\*\*\* Information of Source Program \*\*\*\*\*\*\*\*\*\*\*\*\*")

    # Print token distribution with proper unpacking

    for category, tokens\_list in categories.items():

        # Display the category name and total count of tokens in this category

        print(f'\nTotal {category}: {len(tokens\_list)} tokens')

        # Display the tokens for each category

        print(f'{category} in program:')

        i=1

        for token in tokens\_list:

            print(f'{i}) \t{token}')  # Print each token within the category

            i=i+1

    # Plot token distribution

    plot\_token\_distribution(categories)

    # Plot the histogram of token occurrences

    plot\_token\_histogram(categories)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

# **Output:**

Choose an option:

======= Press 1 to Use the sample Python program ======

======== Press 2 to Enter your own source code ========

== Press 3 to Enter path of your source program file ==

User Selection: Provide Program file

# List of numbers from 1 to 10

numbers = list( range ( 1, 11 )) # This is a simple list from 1 to 10

# Function to calculate the sum of even numbers using bitwise and logical operators

def sum\_of\_even\_numbers( nums ):

total = 0

for num in nums:

# Bitwise operation for even check: number & 1 should be 0 if even

if ( num & 1 ) == 0: # Logical and bitwise operator combined

total += num # Add even number to the total

return total

# Main code block

if \_\_name\_\_ == "\_\_main\_\_":

# Calculate the sum of even numbers from the list

even\_sum = sum\_of\_even\_numbers( numbers )

# Output the sum of even numbers

print( "Sum of even numbers:", even\_sum)

# Check if a number is even or odd using bitwise operator

for num in numbers:

if ( num and 1 ) == 0: # If the number is even

print( "The", num, "is even")

else: # If the number is odd

print( "The", num, "is odd")

\*\*\*\*\*\*\*\*\*\*\* Information of Source Program \*\*\*\*\*\*\*\*\*\*\*

Total Keywords: 14 tokens

Keywords in program:

1) def

2) for

3) in

4) if

5) return

6) if

7) print

8) for

9) in

10) if

11) and

12) print

13) else

14) print

Total Relational Operators: 3 tokens

Relational Operators in program:

1) ==

2) ==

3) ==

Total Comments: 12 tokens

Comments in program:

1) # List of numbers from 1 to 10

2) # This is a simple list from 1 to 10

3) # Function to calculate the sum of even numbers using bitwise and logical operators

4) # Bitwise operation for even check: number & 1 should be 0 if even

5) # Logical and bitwise operator combined

6) # Add even number to the total

7) # Main code block

8) # Calculate the sum of even numbers from the list

9) # Output the sum of even numbers

10) # Check if a number is even or odd using bitwise operator

11) # If the number is even

12) # If the number is odd

Total Identifiers: 22 tokens

Identifiers in program:

1) numbers

2) list

3) range

4) sum\_of\_even\_numbers

5) nums

6) total

7) num

8) nums

9) num

10) total

11) num

12) total

13) \_\_name\_\_

14) even\_sum

15) sum\_of\_even\_numbers

16) numbers

17) even\_sum

18) num

19) numbers

20) num

21) num

22) num

Total Logical Operators: 0 tokens

Logical Operators in program:

Total Arithmetic Operators: 4 tokens

Arithmetic Operators in program:

1) =

2) =

3) +=

4) =

Total Bitwise Operators: 1 tokens

Bitwise Operators in program:

1) &

Total Numeric Literals: 7 tokens

Numeric Literals in program:

1) 1

2) 11

3) 0

4) 1

5) 0

6) 1

7) 0

Total String Literals: 6 tokens

String Literals in program:

1) "\_\_main\_\_"

2) "Sum of even numbers:"

3) "The"

4) "is even"

5) "The"

6) "is odd"

Total Others: 29 tokens

Others in program:

1) (

2) (

3) ,

4) ))

5) (

6) ):

7) :

8) (

9) )

10) :

11) :

12) (

13) )

14) (

15) ,

16) )

17) :

18) (

19) )

20) :

21) (

22) ,

23) ,

24) )

25) :

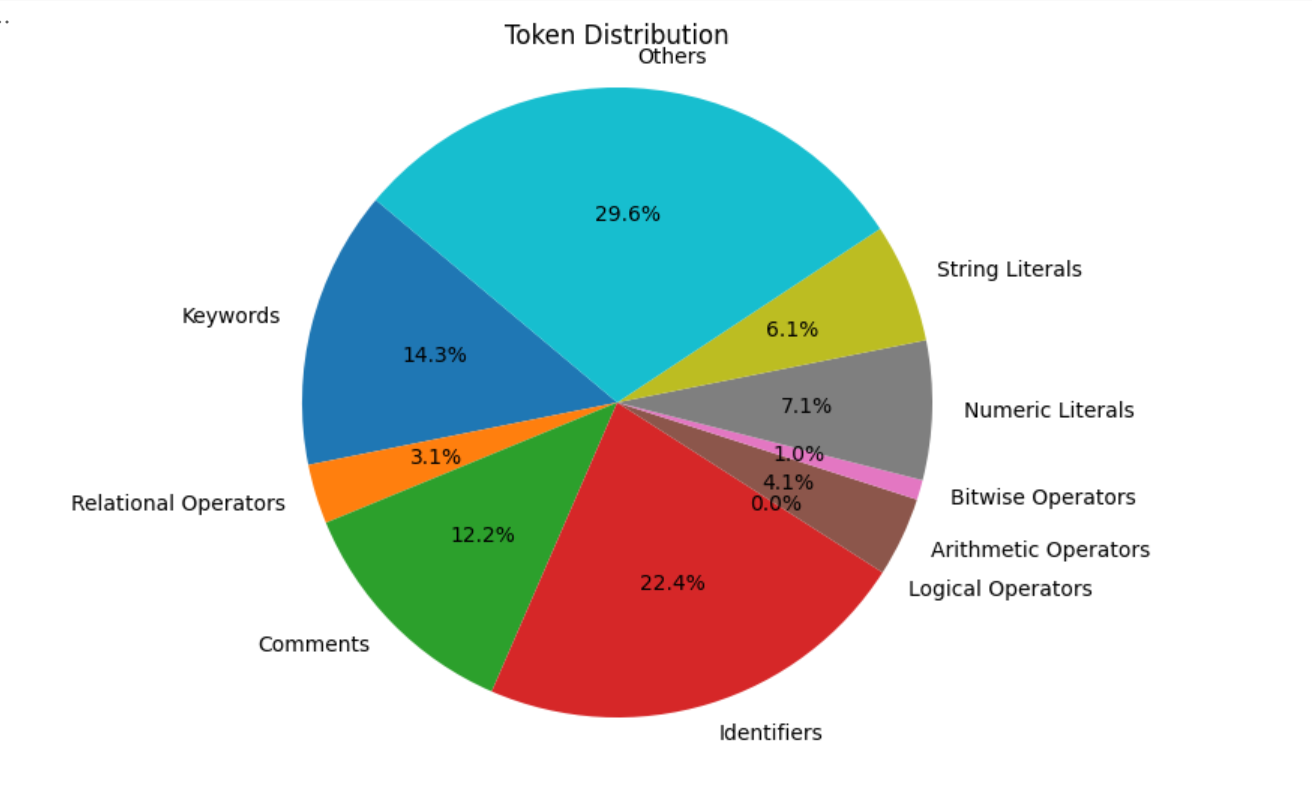
26) (

27) ,

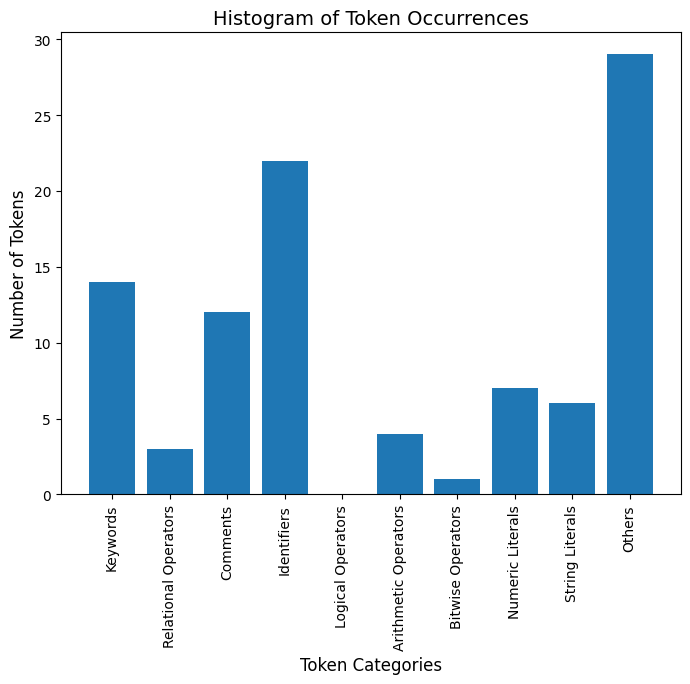
28) ,

29) )

# **Visual Output:**



**Figure 1**



**Figure 2**

# **Conclusion**

This project provides a comprehensive tool for analyzing and categorizing Python source code. By breaking down the code into key components and visualizing their distribution, it aids in understanding, reviewing, and improving the code. The modular structure and flexibility in input methods make it a versatile tool for various applications.

# **References**

1. (Documentation, Python Regular Expressions, n.d.)
2. (Documentation, Matplotlib;, n.d.)
3. (Python Keywords Module, n.d.)