DS2030 Data Structures and Algorithms for Data Science
Lab 2 (In Person) Due on August 27, 5.00pm

Instructions

- You are to use Python as the programming language. Use may use Visual Studio Code (or any other editor you are comfortable with) as the IDE.
- You have to work individually for this lab.
- You are not allowed to share code with your classmates nor allowed to use code from the internet. You are encouraged engage in high level discussions with your classmates; however ensure to include their names in the report/code documentation. If you refer to any source on the Internet, include the corresponding citation in the report/code documentation. If we find that you have copied code from your classmate or from the Internet, you will get a straight fail grade in the course.
- The submission must be a zip file with the following naming convention - rollnumber.zip.
- Include appropriate comments to document the code. Include a `read me` file containing the instructions on for executing the code. The code should run on institute linux machines.
- Upload your submission to moodle by the due date and time. Do not email the submission to the instructor or the TA.

1 Introduction

This lab aims to guide you through implementing a simple interpreter for LISP-like arithmetic expressions using the stack data structure in Python. You will:

- Understand how to implement a stack using Python lists.
- Write functions within the stack class to perform essential operations.
- Implement a tokenizer function to parse LISP expressions.
- Implement the core functions for evaluating LISP expressions.
- Read test cases from an input text file and validate your interpreter.

2 LISP-Like Arithmetic Expressions

LISP (LISt Processing) is a family of programming languages known for its fully parenthesized prefix notation. In this lab, we will work with a simplified version of LISP-like arithmetic expressions.

2.1 Expression Format

LISP-like arithmetic expressions are written in prefix notation, where the operator comes before its operands. Each expression is enclosed in parentheses.

- **Operators**: The supported operators are `+`, `-`, `*`, and `/`, which correspond to addition, subtraction, multiplication, and division, respectively.
- **Operands**: Operands are integers (both positive and negative).

- **Expressions**: An expression is a combination of an operator followed by one or more operands, all enclosed in parentheses.

2.1.1 Examples of LISP-Like Expressions

- `( + 1 2 )` - This expression adds `1` and `2` to yield `3`.
- `( * 2 3 )` - This expression multiplies `2` and `3` to yield `6`.
- `( - 5 2 )` - This expression subtracts `2` from `5` to yield `3`.
- `( / 8 4 )` - This expression divides `8` by `4` to yield `2`.
- `( + 1 ( * 2 3 ) )` - This is a nested expression that multiplies `2` and `3` first, then adds `1` to the result, yielding `7`.

2.2 Handling Nested Expressions

LISP-like expressions can be nested, meaning that the operands themselves can be expressions. The innermost expressions are evaluated first, with the results passed up to the outer expressions.

2.2.1 Example of a Nested Expression

- `( * ( + 1 2 ) ( - 4 1 ) )` - In this expression, `( + 1 2 )` and `( - 4 1 )` are evaluated first to yield `3` and `3`, respectively. Then, `3` is multiplied by `3` to yield `9`.

Understanding this format is essential as you implement the interpreter to correctly parse and evaluate these expressions.

3 Stack Data Structure

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. In Python, stacks can be efficiently implemented using lists, where the `append()` and `pop()` methods allow adding and removing elements from the end of the list, respectively.

3.1 Implementing the Stack Class

You are required to implement a `Stack` class with the following methods:

- `push(item)`: Adds an item to the top of the stack.
- `pop()`: Removes and returns the item from the top of the stack.
- `peek()`: Returns the item at the top of the stack without removing it.
- `is_empty()`: Returns `True` if the stack is empty; otherwise, `False`.

3.1.1 Code Template for `stack.py`

Listing 1: stack.py

```
class Stack:
    def __init__(self):
        """Initialize an empty stack using a list."""
        self.items = []

    def is_empty(self):
        """Check if the stack is empty.

```

```python
        Returns:
            bool: True if the stack is empty, False otherwise.
        """
        # TODO: Implement the is_empty method
        pass

    def push(self, item):
        """Push an item onto the top of the stack.

        Args:
            item: The item to be pushed onto the stack.
        """
        # TODO: Implement the push method
        pass

    def pop(self):
        """Remove and return the item from the top of the stack.

        Returns:
            The item at the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        # TODO: Implement the pop method with proper error handling
        pass

    def peek(self):
        """Return the item at the top of the stack without removing it.

        Returns:
            The item at the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        # TODO: Implement the peek method with proper error handling
        pass
```

4 Tokenizing LISP Expressions

Before evaluating LISP-like expressions, they need to be tokenized—broken down into meaningful units like operators, operands, and parentheses.

4.1 Implementing the Tokenizer Function

Implement the `tokenize` function that takes a LISP expression as input and returns a list of tokens.

4.1.1 Code Template for `tokenizer.py`

Listing 2: tokenizer.py

```python
def tokenize(expression):
    """
    Tokenizes a LISP-like expression.

    Args:
        expression (str): The LISP expression to tokenize.

```

```
    Returns:
        list: A list of tokens.
    """
    tokens = []
    # TODO: Implement the tokenizer function
    return tokens
```

5 Evaluating LISP Expressions

Using the stack you've implemented and the tokenizer function, you will evaluate LISP-like arithmetic expressions.

5.1 Implementing the `apply_operator` Function

The `apply_operator` function is responsible for performing the arithmetic operation based on the operator and the operands provided.

5.1.1 Code Template for `apply_operator`

Listing 3: lisp_calculator.py

```
def apply_operator(op, operands):
    """
    Applies the operator to the list of operands.

    Args:
        op (str): The operator (e.g., '+', '-', '*', '/').
        operands (list): The list of operands (integers).

    Returns:
        int: The result of applying the operator.

    Raises:
        ValueError: If the operator is unknown.
    """
    # TODO: Implement the apply_operator function
    pass
```

5.2 Implementing the `evaluate_lisp_expression` Function

The `evaluate_lisp_expression` function will process the tokens and use the stack to compute the result of the expression. This function should handle basic arithmetic operations and nested expressions.

5.2.1 Code Template for `evaluate_lisp_expression`

Listing 4: lisp_calculator.py

```
from stack import Stack
from tokenizer import tokenize

def evaluate_lisp_expression(expression):
    """
    Evaluates a LISP-like arithmetic expression.

    Args:
        expression (str): The LISP expression to evaluate.

    Returns:
```

```python
        int: The result of the evaluation.

    Raises:
        ValueError: If the expression is invalid or contains unknown operators.
    """
    stack = Stack()
    tokens = tokenize(expression)
    # TODO: Implement the evaluate_lisp_expression(expression) function with proper error handling
```

6 Testing Your Interpreter

To validate your interpreter, you will read test cases from an input text file and verify the evaluated results.

6.1 Preparing the Test Cases File

Create a file named `test_cases.txt` with each line containing a LISP expression and its expected result, separated by a comma.

6.1.1 Example Content of `test_cases.txt`

```
( + 1 2 ),3
( * 2 3 ),6
( - 5 2 ),3
( / 8 4 ),2
( + 1 ( * 2 3 ) ),7
( * ( + 1 2 ) ( - 4 1 ) ),9
```

6.2 Implementing the Test Runner

6.2.1 Code Template for `test_lisp_calculator.py`

Listing 5: test_lisp_calculator.py

```python
from lisp_calculator import evaluate_lisp_expression

def run_tests(test_file):
    """
    Runs test cases from the specified file.

    Args:
        test_file (str): Path to the test cases file.
    """
    with open(test_file, 'r') as file:
        lines = file.readlines()

    for line in lines:
        expr, expected = line.strip().split(',')
        expected = int(expected) # Convert expected result to integer
        result = evaluate_lisp_expression(expr)
        assert result == expected, f"Test failed for expression {expr}. Expected {expected}, got {result}"

    print("All tests passed!")

if __name__ == "__main__":
    run_tests('test_cases.txt')
```

7 Lab Submission

Ensure that you complete the following before submission:

- Implement all the methods in the `Stack` class within `stack.py`.
- Implement the `apply_operator` function to correctly perform arithmetic operations.
- Implement the `evaluate_lisp_expression` function to evaluate LISP-like expressions.
- Verify that your `tokenize` function correctly tokenizes various LISP expressions.
- Test your interpreter using the provided test cases and ensure all tests pass.
- Add comments where necessary to explain the logic, especially in areas that might be non-obvious.

8 References

1. M Goodrich, R Tamassia, and M. Goldwasser, "Data Structures and Algorithms in Python", 1^{st} edition, Wiley, 2013.