Evaluating & Sorting Assignment Statements (using parse trees)

Module: Data Structures & Algorithms

CA2 Group Report

Kallen Ng Jia Jun (P2222556), Toh Kien Yu (P2222291)

Group: 5

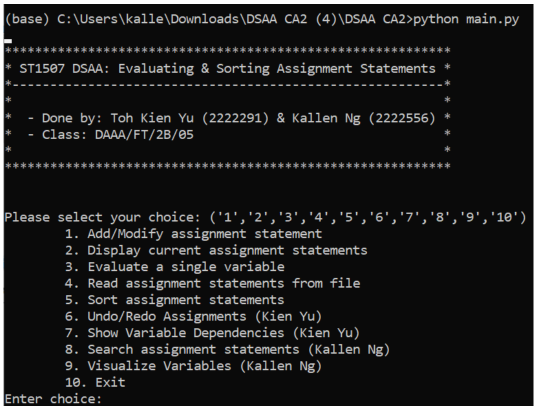
Class: DAAA/FT/2B/05

Introduction

This project involves applying the concepts we have learned thus far, including Data Structures, Algorithms, and Object-Oriented Programming (OOP) by developing an application capable of solving assignment statements with mathematical statements by utilizing parse trees. Parse trees are special binary trees that can be used to represent and solve mathematical expressions. The tree can be solved by starting at the leaves and then solving the subtree expressions first, afterwards it progressively works upwards until it reaches the root of the tree..

User Guideline

Open up Anaconda Prompt and switch to the project directory and run the program using:
python main.py

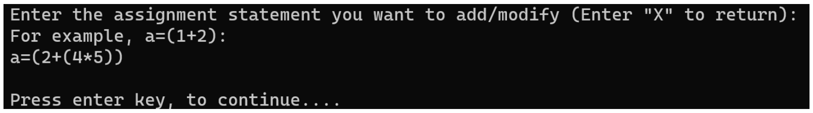


```
(base) C:\Users\kalle\Downloads\DSAA CA2 (4)\DSAA CA2>python main.py

***********************************************************
* ST1507 DSAA: Evaluating & Sorting Assignment Statements *
*---------------------------------------------------------*
*                                                         *
*  - Done by: Toh Kien Yu (2222291) & Kallen Ng (2222556) *
*  - Class: DAAA/FT/2B/05                                 *
*                                                         *
***********************************************************


Please select your choice: ('1','2','3','4','5','6','7','8','9','10')
        1. Add/Modify assignment statement
        2. Display current assignment statements
        3. Evaluate a single variable
        4. Read assignment statements from file
        5. Sort assignment statements
        6. Undo/Redo Assignments (Kien Yu)
        7. Show Variable Dependencies (Kien Yu)
        8. Search assignment statements (Kallen Ng)
        9. Visualize Variables (Kallen Ng)
        10. Exit
Enter choice:
```

Figure 1: Main Menu

The main menu will automatically appear and prompt the user to enter a choice between 1-10. [Figure 1] Upon entering a choice the program will subsequently request input which varies depending on the option the user has chosen. The user will be able to repeatedly select options from the menu, until he/she selects option 10 after which the application will terminate.

Option 1: Add/Modify Assignment Statements



```
Enter the assignment statement you want to add/modify (Enter "X" to return):
For example, a=(1+2):
a=(2+(4*5))

Press enter key, to continue....
```

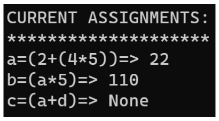
Figure 2: Adding / Modifying Assignment Statements

The purpose of Option 1 is to allow users to add assignments or modify existing assignments. [Figure 2] After the user enters the assignment, the application will store the assignment in the storage class for later usage. The user is only allowed to enter assignment statements that are correctly parenthesized and contain no spaces. For example if a user inputs a=(1+2+3), the application will request the user to enter it in the proper format, such as a=(1+(2+3)).

When a user inputs an assignment statement, the application stores/updates the information within the ‘Storage’ class. The information is stored in a dictionary where the variable is stored as the key and the value assigned to it is stored as the corresponding value.

Option 2: Display Current Assignment Statements

The user may for instance enter a=(2+(4*5)) followed by b=(a*5) then c= (a+d).



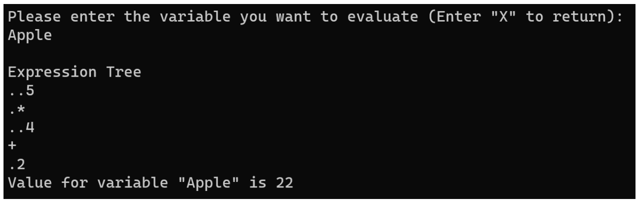
```
CURRENT ASSIGNMENTS:
********************
a=(2+(4*5))=> 22
b=(a*5)=> 110
c=(a+d)=> None
```

Figure 3: Display Current Assignment Statements

The purpose of Option 2 is to allow users to view the current assignment statements along with their evaluated results stored in the system. The statements are then evaluated using parse trees. If a statement includes expressions with variables that have not been previously defined, their values will be shown as ‘None’. Additionally, the statements are sorted alphabetically by their variable names.

The core functionality of option 2 is mainly implemented through the ‘Storage’ class, specifically leveraging techniques such as retrieving and displaying of assignments that are stored in the storage’s dictionary. This functionality is part of a larger program designed to manage assignments, evaluate expressions, and provide insights into variable dependencies

Option 3: Evaluate A Single Variable



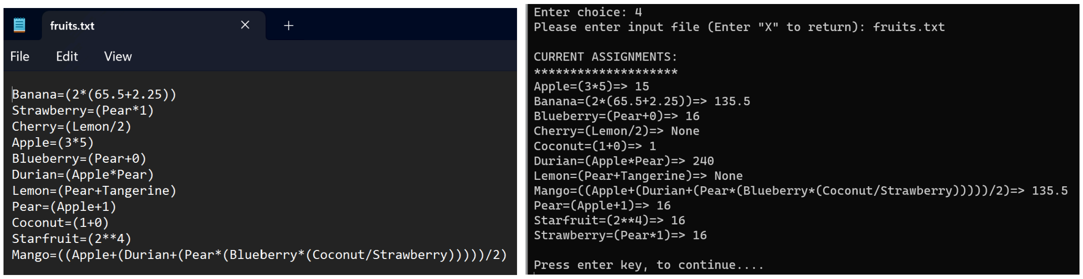
```
Please enter the variable you want to evaluate (Enter "X" to return):
Apple

Expression Tree
..5
.*
..4
+
.2
Value for variable "Apple" is 22
```

Figure 4: Evaluate A Single Variable

Option 3 allows the user to display the parse tree for a specific variable, with the tree being shown in in-order traversal. Figure 4 illustrates the outcome when the user inputs Apple=(2+(4*5)).

Option 3 is mainly constructed through the 'Storage', 'ParseTree' and 'BinaryTree' class. When a user inputs a variable, the 'Storage' class first verifies whether the variable exists within the storage. If the variable exists, it then retrieves the assignment corresponding to the variable and passes it to the 'ParseTree' class. The ParseTree class then constructs a parse tree and the BinaryTree class will then print the parse tree in the in-order traversal format.

Option 4: Read Assignment Statement From File



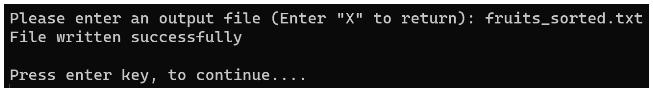
```
Banana=(2*(65.5+2.25))
Strawberry=(Pear*1)
Cherry=(Lemon/2)
Apple=(3*5)
Blueberry=(Pear+0)
Durian=(Apple*Pear)
Lemon=(Pear+Tangerine)
Pear=(Apple+1)
Coconut=(1+0)
Starfruit=(2**4)
Mango=((Apple+(Durian+(Pear*(Blueberry*(Coconut/Strawberry)))))/2)
```

```
Enter choice: 4
Please enter input file (Enter "X" to return): fruits.txt

CURRENT ASSIGNMENTS:
********************
Apple=(3*5)=> 15
Banana=(2*(65.5+2.25))=> 135.5
Blueberry=(Pear+0)=> 16
Cherry=(Lemon/2)=> None
Coconut=(1+0)=> 1
Durian=(Apple*Pear)=> 240
Lemon=(Pear+Tangerine)=> None
Mango=((Apple+(Durian+(Pear*(Blueberry*(Coconut/Strawberry)))))/2)=> 135.5
Pear=(Apple+1)=> 16
Starfruit=(2**4)=> 16
Strawberry=(Pear*1)=> 16

Press enter key, to continue....
```

Figure 5: Raw txt file & Assignments Read from file

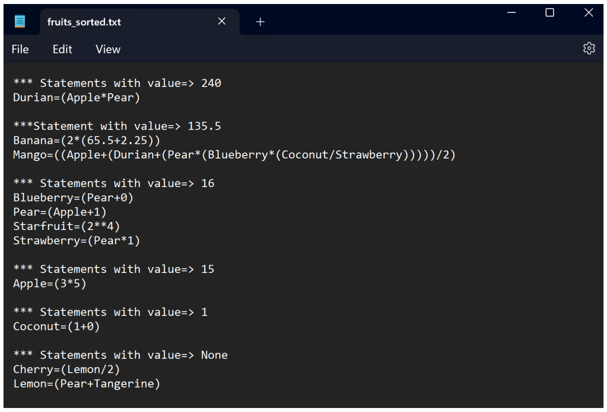
The purpose of Option 4 is to allow users to input a file they wish to read and process assignment statements from that file. [Figure 5] This option provides a means for users to import a collection of assignment statements stored in a file, extending the program's functionality beyond manual input. The application reads the input file using readFile function within the File class to read assignment statements before it parses and evaluates them using BinaryTree class. The assignment statements are then added to the storage. It also incorporates the sorting functionality upon reading and evaluating the statements from the file.

Option 5: Sort Assignment Statements



```
Please enter an output file (Enter "X" to return): fruits_sorted.txt
File written successfully

Press enter key, to continue....
```

Figure 6: Sort Assignment Statements



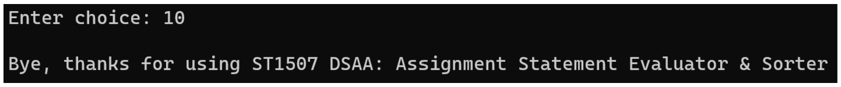
```
*** Statements with value=> 240
Durian=(Apple*Pear)

***Statement with value=> 135.5
Banana=(2*(65.5+2.25))
Mango=((Apple+(Durian+(Pear*(Blueberry*(Coconut/Strawberry)))))/2)

*** Statements with value=> 16
Blueberry=(Pear+0)
Pear=(Apple+1)
Starfruit=(2**4)
Strawberry=(Pear*1)

*** Statements with value=> 15
Apple=(3*5)

*** Statements with value=> 1
Coconut=(1+0)

*** Statements with value=> None
Cherry=(Lemon/2)
Lemon=(Pear+Tangerine)
```

Figure 7: Exported file with the fruits sorted

After assigning a set of assignment statements into the application, the user can organize them by choosing option 5 [Figure 6]. Option 5 will sort the statements and save them in an output file name of their choice. [Figure 7]

Option 5 utilizes the merge sort algorithm to rearrange the variables in the exported file in ascending order. The use of the merge sort algorithm separates the sorting logic from the main program which makes it maintainable in the long run. Upon completion of the sorting operation, it then writes the sorted statements into the output file.

Option 10 : Exit the program



```
Enter choice: 10

Bye, thanks for using ST1507 DSAA: Assignment Statement Evaluator & Sorter
```

Figure 8: Exit the program

Upon selecting option 10, the program will terminate.

Object-Oriented Programming

The project is implemented through Object-Oriented Programming (OOP), incorporating fundamental concepts such as encapsulation, polymorphism and abstraction.

Encapsulation

Encapsulation is a fundamental concept in object-oriented programming (OOP) that plays a crucial role in enhancing code organization, modularity, and security. At its core, encapsulation involves bundling data and the methods that operate on that data into a single unit known as a class. The primary objective is to encapsulate the internal workings of an object, providing a well-defined interface for interactions while hiding the implementation details.

Implementation

Class	Name	Type	Purpose
Menu	self.__length self.__list	Private Variables	The Menu class represents a menu in a program, encapsulating menu-related data and behavior.

	getLength(self)
	setLength(self)
	getList(self)
	setList(self)
		Getter and Setter	

Stack	self.__list	Private Variables	Hides the internal list from external code, preventing direct manipulation. This ensures that stack operations occur through the defined methods which helps maintain the integrity of the stack.
	get(self)	Getter and Setter	
Storage	self.__storage self.__undostack self.__redostack	Private Variables	External code cannot directly access or modify __storage, ensuring that interactions with the storage occur through controlled methods.

	getStorage(self)
	getUndoStack(self)
	getRedoStack(self)
		Getter and Setter	

Polymorphism

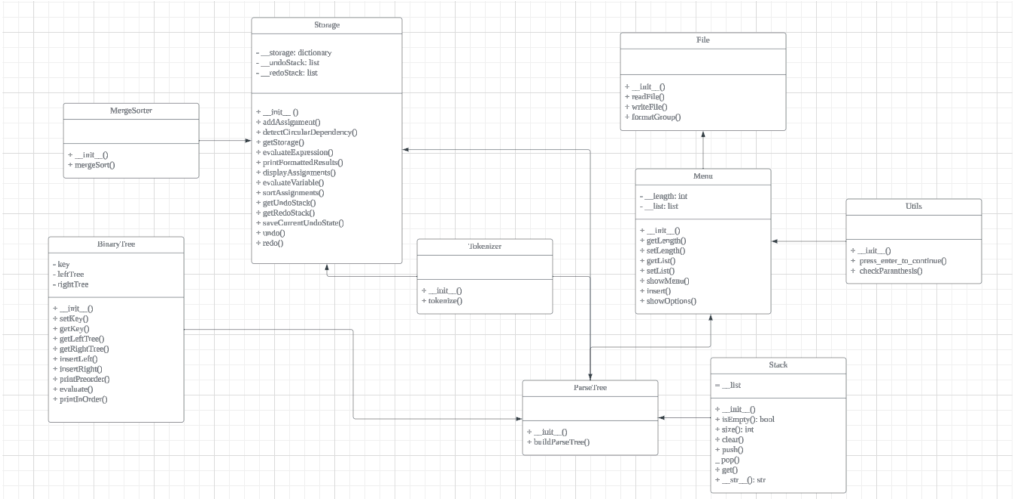
Polymorphism was done particularly in the evaluate method of the Binary Tree class, showcasing method overriding. This method allows various node types, including leaf and binary nodes, to respond uniquely to the common evaluate message. Leaf nodes handle individual values or variables, while binary nodes utilize a recursive approach for operators, emphasizing adaptability and code reusability.

Abstraction

Abstraction, a fundamental concept in object-oriented programming, involves hiding the intricate details of implementation while exposing only essential features or interfaces.

In the ParseTree class, abstraction is employed to conceal the complex processes of parsing mathematical expressions and constructing parse trees. Users simply need to use the buildParseTree() method with a mathematical expression, without worrying about the details of how expressions are tokenized or how nodes are inserted into the tree.

Class Diagram



Data Structures

List & Dictionaries:

We used lists and dictionaries to organize and process data efficiently in the context of parsing expressions, managing variable assignments, and implementing stack operations. They provide easy access and manipulation of information without the need for complex search or sorting algorithms.

List was used such as the tokens generated by the Tokenizer class or nodes within the parse tree. This way we are able to efficiently add,remove or access elements at any state, which is crucial for operations like token processing and construction of parse trees.

In storage.py, dictionaries such as .__storage was used to facilitate quick access to variables corresponding to their values. This approach enhances the overall efficiency of the code when handling variable assignments and related operations.

Stack:

A stack, implemented in the ParseTree class adhering to the Last In, First Out (LIFO) principle, serves as a fundamental data structure. Its role in our implementation involves orchestrating the construction of a parse tree, facilitating the parsing and evaluation of mathematical expressions in a methodical and efficient manner. This involves the stack's utilization for node tracking during parse tree construction, particularly in the parsing phase of mathematical expressions. The push and pop operations play a crucial role in maintaining the hierarchy and structure of the parse tree.

Binary Tree:

The binary tree, implemented in the ParseTree class, is used to represent the hierarchical structure of mathematical expressions in the form of parse trees. Each node in the binary tree corresponds to an operator or operand, and the tree's structure mirrors the syntactic structure of the expressions. The binary tree facilitates both the construction of parse trees and the evaluation of expressions through recursive traversal.

Data Structures	**Use Case**
Lists	Used to maintain ordered collections of items of tokens and parse tree nodes.

Dictionaries	Used for mapping variable to expressions and values, providing efficient lookups and organization based on keys
Stack (Last-In-First-Out)	Used to keep track of nodes in the construction of parse tree
Binary Tree	Represent the hierarchical structure of mathematical expressions and evaluation in the form of a parse tree

Algorithms

Merge Sort

Merge Sort is a divide-and-conquer sorting algorithm that efficiently sorts an array by repeatedly dividing it into halves, sorting each half, and merging them back together. This algorithm aligns well with the hierarchical structure of parse trees, which represent syntactic relationships in programming languages. Merge Sort ensures optimal sorting efficiency with a time complexity of O(n log n)

In our application merge sort is used to organize the assignment statements by checking if the list of assignments is larger than a single element which indicates sorting needs to be done. It then divides the list into 2 halves, applying the same logic recursively until each sublist contains one element or is empty. It then merges the divided halves back together and compares the elements of each halves based on a specific order ascending or descending.

Tokenizer

We used a tokenizer to break down a sequence of characters, a string representing a program or expression into smaller units called tokens. These tokens serve as the fundamental building blocks for constructing a parse tree.

The tokenizer in our application takes in a mathematical expression and dissects it into a sequence of tokens, effectively transforming a continuous string of characters into distinct units. It starts off by taking in the entire expression as an input then iterates through the expression. The token then groups the characters into tokens based on their roles, for example alphanumeric characters and decimal values or identifiers for variables, while special characters representing operators and parentheses are an individual token. A key aspect of our tokenizer includes recognizing (‘**’) as an exponent operation and treated as a single token. Upon tokenizing the whole expression, it returns a list of tokens ready for use which lays down the base when using it to evaluate using Parse Trees.

Recursion

Recursion is used in our Binary Tree class within the 'evaluate' method to calculate mathematical expression. Recursion allows us to tackle complex problems using a divide-and-conquer approach. This method recursively evaluates the left and right subtrees for each node that represents an operation. This process continues until it reaches the leaf node .

Sorting and Evaluating

Algorithms	**Big (O)**	**Conclusion**
Merge Sort	O(n log n)	Excellent and often the best achievable time complexity for certain types of problems, especially in sorting and searching scenarios.
Tokenizer	O(n)	Very efficient,,it is representing an algorithm that has a linear relationship with the size of the input.
Recursion (BinaryTree evaluate function)	Balanced binary tree: O(log n) Unbalanced binary tree: O(n)	The algorithm is efficient when traversing and evaluating the trees. This efficiency is crucial when evaluating complex expressions.

Summary of Challenges & Takeaways

Challenges we faced:

We encountered several technical challenges during the project, particularly in developing an efficient program that effectively utilized classes and Object-Oriented Programming (OOP) technology as we found the concepts of OOP to be particularly challenging. We also struggled in implementing parse trees that are used for mathematical calculations as it was a newly taught concept which required us to figure things out along the way.

Furthermore, managing time as a group posed difficulties given our individual responsibilities beyond the project. Hence, we often find it difficult to make time to meet and work together on the project.

Key Takeaways:

This project helped us get better at managing our time. It also taught us to divide the work based on what we're good at, so we could do the project more efficiently.

While working on the application, we faced several problems that required debugging. This experience has taught us to effectively communicate with one another to swiftly resolve issues. Additionally, it instilled problem-solving skills such as breaking down the problem into simpler problems. This approach provides a clearer perspective, enabling us to evaluate various solutions and choose the most effective one.

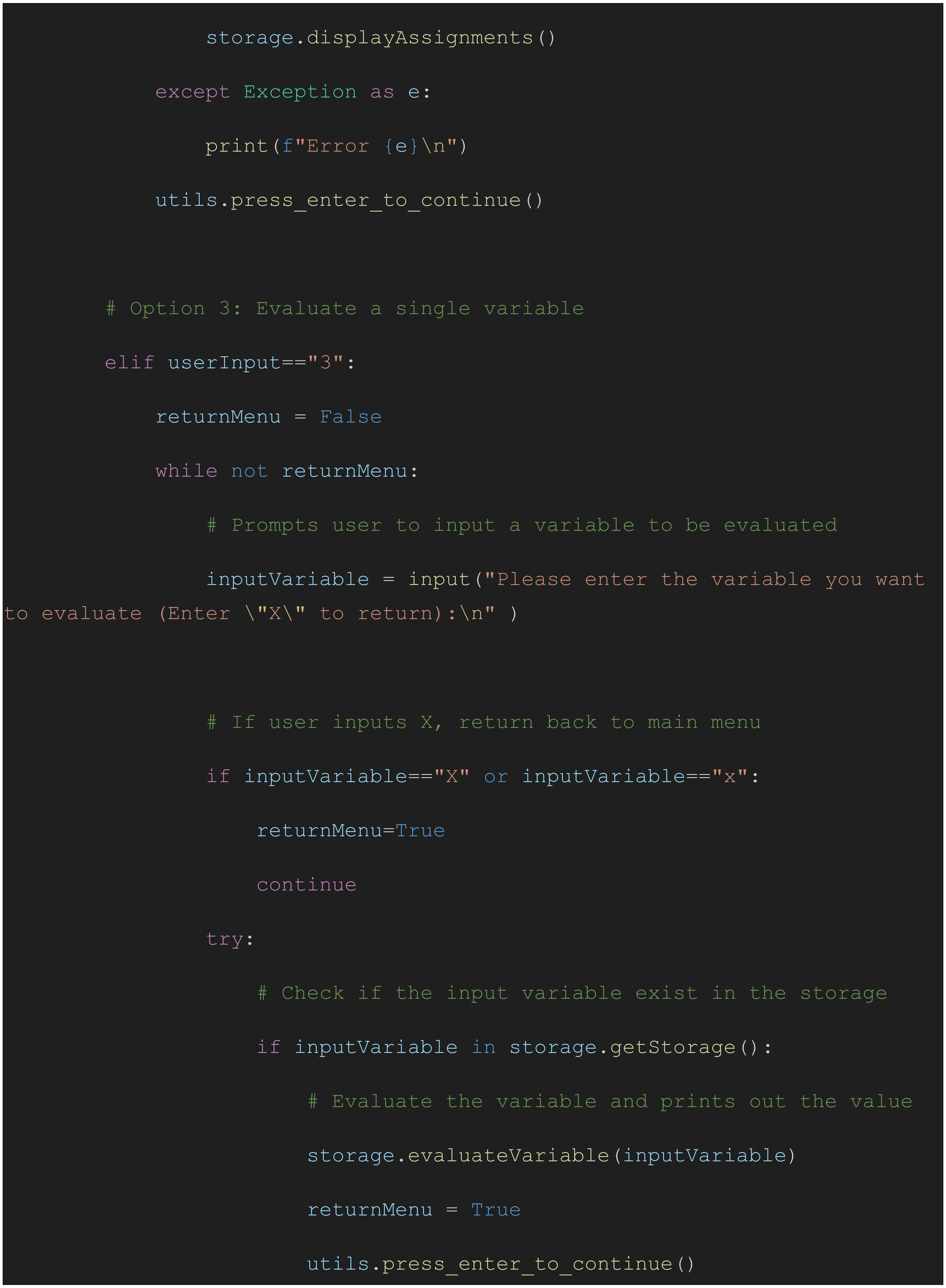
Contributions

Name	Tasks	Contribution
Kallen Ng Jia Jun	1. Read assignments statements from file 2. Sort assignment statements 3. Implement Merge Sort Algorithm 4. Error handling 5. Report 6. Write code comments	50%
Toh Kien Yu	1. Add/ Modify assignment statements 2. Building of Parse Tree 3. Display and evaluate current assignment statements 4. Evaluate a single variable 5. Error handling 6. Implement Tokenizer 7.Write comments	50%

Appendix

```
#-----------------------------------------------------
# Main Program
#-----------------------------------------------------
#
# Filename  : main.py
#
#-----------------------------------------------------
# To run: python main.py
#-----------------------------------------------------
# This is the main program for
# ST1507 DSAA: Evaluating & Sorting Assignment Statements.


from Menu.menu import Menu
from Storage.storage import Storage
from Tree.parseTree import ParseTree
from Tree.tokenizer import Tokenizer
from utils import Utils
from File.file import File
from Search.search import Search
```

```python
from Visualize.visualize import VariableBarPlot
import os


def main():
    # Initialize the classes
    menu = Menu(10)
    storage = Storage()
    utils = Utils()
    fileOperation = File()
    menu.showMenu()
    menu.insert(['Add/Modify assignment statement','Display current assignment statements','Evaluate a single variable','Read assignment statements from file','Sort assignment statements','Undo/Redo Assignments (Kien Yu)','Show Variable Dependencies (Kien Yu)','Search assignment statements (Kallen Ng)','Visualize Variables (Kallen Ng)','Exit',])
    exit = False


    # Main program
    while not exit:
        # Display main menu
        menu.showOptions()
        # User input
        userInput = input("Enter choice: ")


        # Validate user input
        if not userInput.isdigit() or int(userInput) < 1 or int(userInput) > menu.getLength():
            print(f"Invalid Input. Please enter a number between 1 and {menu.getLength()}\n")

        # Option 1: Add/Modify assignment statements
        elif userInput == "1":
            returnMenu = False
            while not returnMenu:
                # Prompts user to enter an assignment statement
                assignment = input("Enter the assignment statement you want to add/modify (Enter \"X\" to return):\nFor example, a=(1+2):\n")

                # If user inputs X, return back to main menu
                if assignment=="X" or assignment == "x":
                    returnMenu = True
                    continue
                try:
                    # Split assignment into a variable and value
                    variable,value = assignment.split("=")

                    # Check for spaces in variable name
                    if " " in variable or " " in value or not utils.checkParanthesis(value):
                        print("Invalid variable name. Spaces are not allowed. Please re-enter your assignment without spaces.\n")
                        continue
                    # Check if the assignment is correctly parenthesized
                    if not ((value.startswith("(") and value.endswith(")"))):
                        print("Invalid format. Ensure that the value is correctly enclosed in paranthesis.\n")
                        continue

                    # Add/update the assignment in the storage
                    success = storage.addAssignment(variable,value)
                    # If successfully add/update the assignment in the storage then return to main menu
                    if success:
                        returnMenu = True
                        utils.press_enter_to_continue()
                except:
                    print("Invalid Assignment. Please enter in the correct format.\n")

        # Option 2: Display current assignment statements
        elif userInput =="2":
            try:
                # Display all the assignments from the storage
```

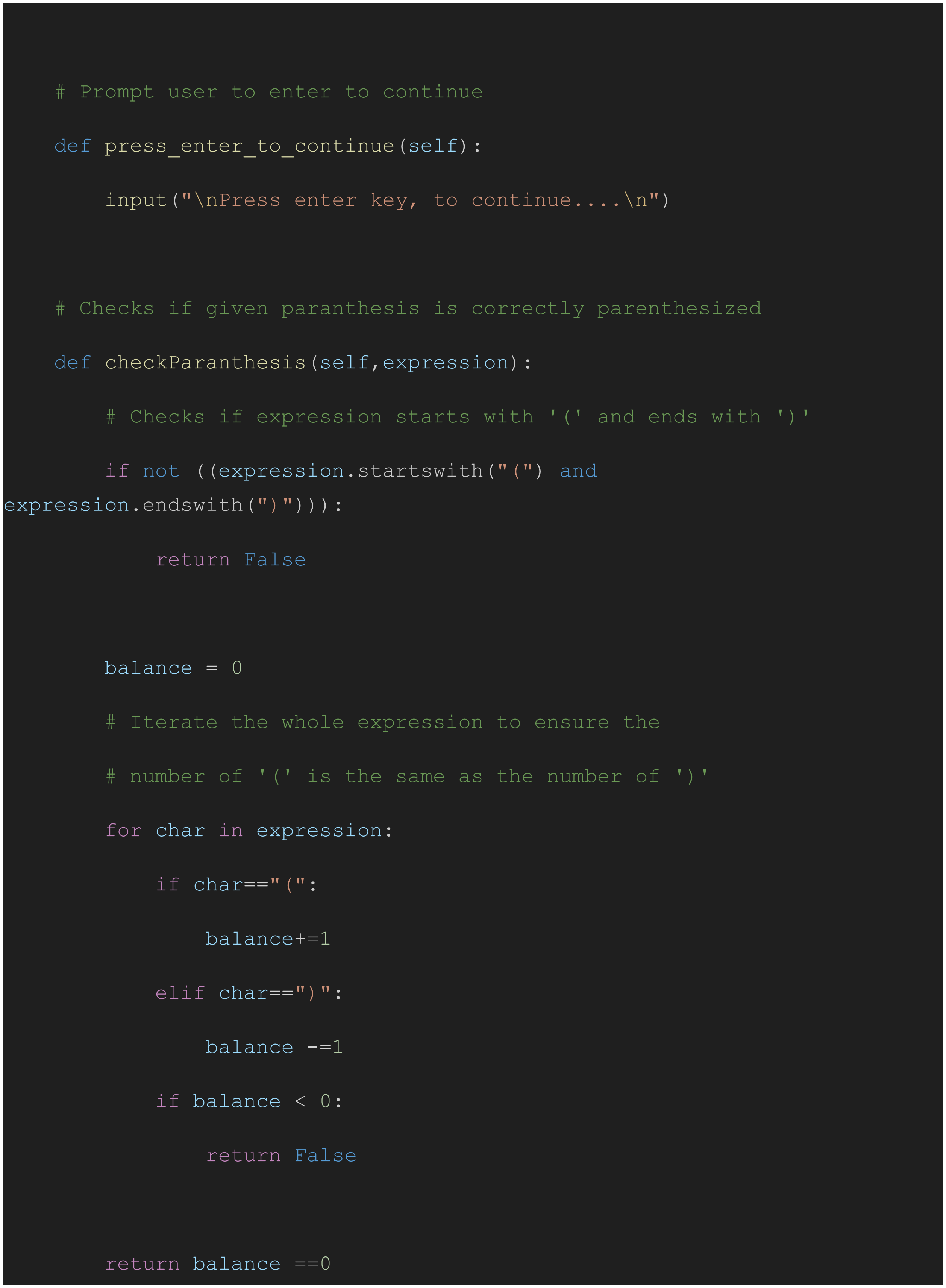


```
                storage.displayAssignments()
            except Exception as e:
                print(f"Error {e}\n")
            utils.press_enter_to_continue()


        # Option 3: Evaluate a single variable
        elif userInput=="3":
            returnMenu = False
            while not returnMenu:
                # Prompts user to input a variable to be evaluated
                inputVariable = input("Please enter the variable you want to evaluate (Enter \"X\" to return):\n" )


                # If user inputs X, return back to main menu
                if inputVariable=="X" or inputVariable=="x":
                    returnMenu=True
                    continue
                try:
                    # Check if the input variable exist in the storage
                    if inputVariable in storage.getStorage():
                        # Evaluate the variable and prints out the value
                        storage.evaluateVariable(inputVariable)
                        returnMenu = True
                        utils.press_enter_to_continue()
                    else:
                        print(f"Variable \"{inputVariable}\" not found\n")
                except Exception as e:
                    print(f"Error {e}")


        # Option 4: Read assignment statements from file
        elif userInput == "4":


            returnMenu = False
            while not returnMenu:
                # Prompts the user to enter the input file
                inputFile = input("Please enter input file (Enter \"X\" to return): ")
                # If user inputs X, return back to main menu
                if inputFile =="X" or inputFile=="x":
                    returnMenu=True
                    continue
                try:
                    # Create a File object
                    fileProcessor = File()
                    # Read assignment statement and add the assignment statements into the storage
                    fileProcessor.readFile(inputFile,storage)
                    utils.press_enter_to_continue()
                    returnMenu=True
                except FileNotFoundError:
                    # If file does not exist then print file not found
                    print("File not found.\n")
                except Exception as e:
                    print(f"Error: {e}")



        # Option 5: Sort assignment statements
        elif userInput == "5":
            returnMenu = False
            while not returnMenu:
                # Prompt user to enter an output file
                outputFile = input("\nPlease enter an output file (Enter \"X\" to return): ")

                # If user inputs X, return back to main menu
                if outputFile =="X" or outputFile=="x":
                    returnMenu=True
                # Check if user enters a .txt file
                elif not outputFile.endswith(".txt"):
                    print("File type not valid. Please enter a .txt file.")
                else:
                    # Checks if the name of the output file exists
                    if os.path.exists(outputFile):
                        # If file exists then prompt whether the user
wants to overwrite or create a new file name
                        choice = input("File already exists. Do you want
to overwrite it?: (Y/N): ").upper()
                        if choice == "Y":
                            try:
                                # Sort the assignments in storage
                                sortedAssignments =
storage.sortAssignments()
                                # Write the sorted assignments into the
output file

fileOperation.writeFile(sortedAssignments,outputFile)
                                print("File written successfully")
                                utils.press_enter_to_continue()
                                returnMenu = True
                            except Exception as e:
                                print(f"Error writing to file {e}")
                        elif choice=="N":
                            continue

                        else:
                            print("Invalid input. Returning to menu.\n")
                            returnMenu = True
                    else:
                        try:
                            # Sort the assignments in storage
                            sortedAssignments = storage.sortAssignments()
                            # Write the sorted assignments into the output
file

fileOperation.writeFile(sortedAssignments,outputFile)
                            print("File written successfully")
                            utils.press_enter_to_continue()
                            returnMenu = True
                        except Exception as e:
                            print(f"Error writing to file {e}")




        # Kien Yu's additional feature: Undo/Redo Assignments
        elif userInput == "6":
            returnMenu = False
            while not returnMenu:
                # Prompt user to enter an action ('D' for undo, 'R' for
redo, 'X' to return to main menu)
```

```
                action = input("\nEnter 'D' to remove the latest entry or 'R' to restore the most recently deleted entry (Enter \"X\" to return): ").upper()

                # If user inputs D, perform an undo operation on the storage
                if action == "D":
                    # Perform undo
                    storage.undo()
                    returnMenu = True
                    utils.press_enter_to_continue()
                # If user inputs R, perform a redo operation on the storage
                elif action == "R":
                    # Perform redo
                    storage.redo()
                    returnMenu = True
                    utils.press_enter_to_continue()

                # If user inputs X, return to main menu
                elif action == "X":
                    returnMenu = True
                else:
                    print("Invalid input. Please enter 'D' or 'R'")
```

```
        # Kien Yu's additional feature: Show Variable Dependencies
        elif userInput == "7":
            # Display the variable dependency
            storage.display_dependencies()

            while True:
                choice = input("\nDo you want to write these dependencies into a text file? (Y/N): ").upper()
                # If user enters 'Y', prompt user to enter output file name
                if choice=="Y":
                    while True:
                        fileName = input("\nEnter filename to write variable dependencies (Enter \"X\" to return): ")
                        # If user inputs X, return to main menu
                        if fileName.upper()=="X":
                            break

                        # Check if user enters a .txt file
                        if not fileName.endswith(".txt"):
                            print("File type not valid. Please enter a .txt file.")
                        else:
                            # Check if the file exists
                            if os.path.exists(fileName):
                                # If file exists, prompt user whether to overwrite the existing file
                                overwrite = input("File already exists. Do you want to overwite it? (Y/N): ").upper()
                                if overwrite == "Y":

                                    try:
                                        # Write variable dependency to file
                                        storage.writeDependencyToFile(fileName)
                                        print(f"Variable dependencies written to {fileName}")
                                        break
                                    except Exception as e:
                                        print(f"Error: {e}")
                                # If user chooses not to overwrite, user will enter a new file name
                                elif overwrite =="N":
                                    continue
                                else:
                                    print("Invalid input. Please enter 'Y' for yes, 'N' for no.")
                            else:
                                try:
                                    # Write variable dependency to file
```

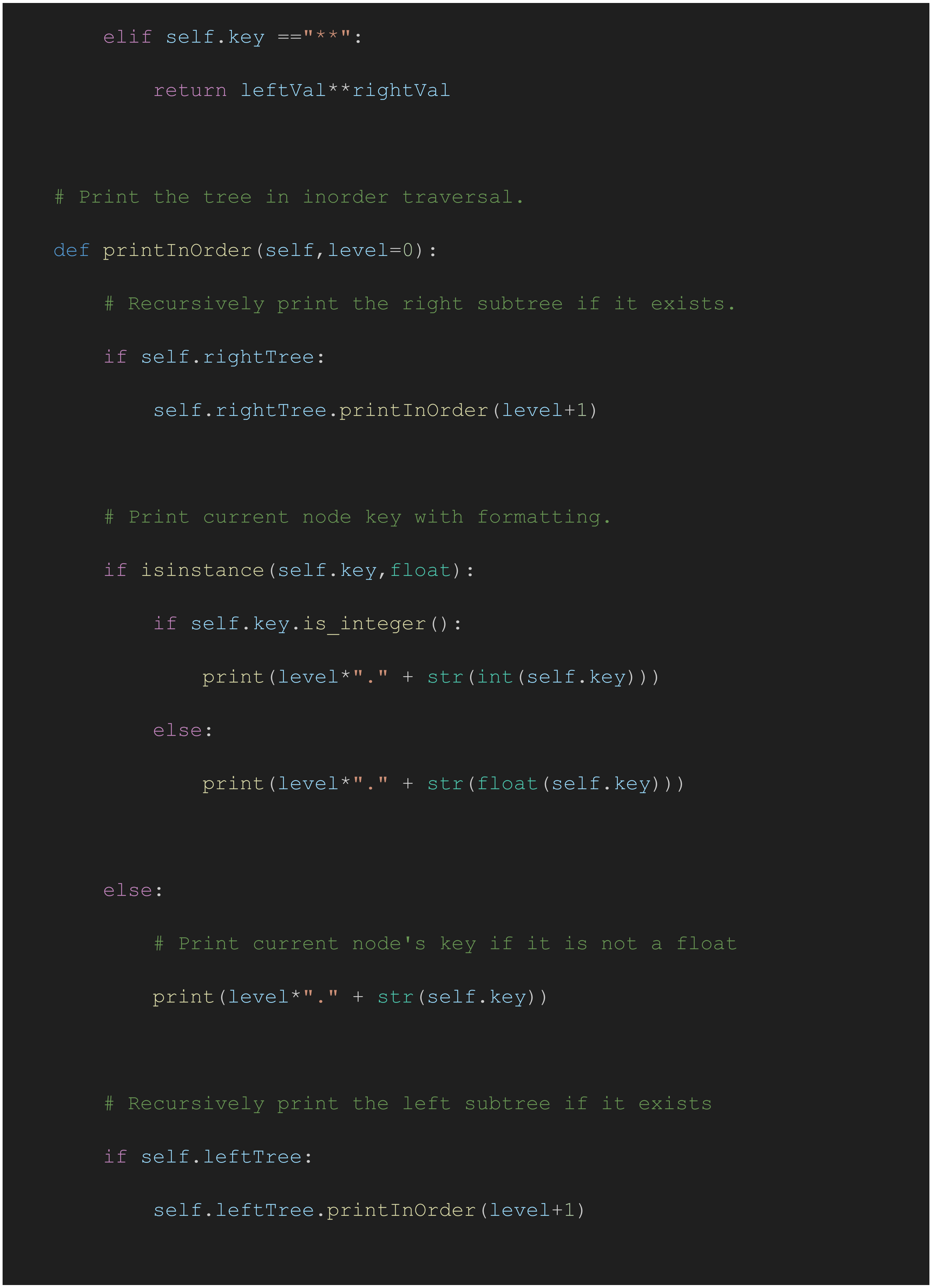
```
storage.writeDependencyToFile(fileName)
                                    print(f"Variable dependencies written
to {fileName}")
                                    break

                                except Exception as e:
                                    print(f"Error: {e}")
                    break
                elif choice=="N":
                    break
                else:
                    print("Invalid input. Please enter 'Y' for yes, 'N'
for no.")

            utils.press_enter_to_continue()

        # Kallen's additional feature: (range searching)
        elif userInput == "8":
            search = Search(storage)
            try:
                search.getRange()
                utils.press_enter_to_continue()
            except Exception as e:
                print(f"Error: {e}")
                utils.press_enter_to_continue()
                pass

        # Kallen's additional feature: (visualize variables using barplot)
        elif userInput == '9':
            try:
                assignments = storage.retrieveAllAssignments() # retrieve
all assignments stored in storage
                if not assignments:
                    print("No assignments found. To proceed, please assign
values to at least one variable. Returning to the main menu...")
                    utils.press_enter_to_continue()
                    returnMenu = True # returns user back to main menu
                    continue
                # Filtering assignments to exclude those with None
values."
                filtered_assignments = [(variable, values) for variable,
values in assignments if None not in values]
                variable_values = {variable: values for variable, values
in filtered_assignments}
                visualize = VariableBarPlot(variable_values) # plot the
barplot
                visualize.plot()
            except Exception as e:
                print(f"Error: {e}")
                utils.press_enter_to_continue()
        # Exit the program
        elif userInput == "10":
            print("\nBye, thanks for using ST1507 DSAA: Assignment Statement Evaluator & Sorter")
            exit = True




if __name__ == "__main__":
    main()

#-----------------------------------------------------
#
# Filename   : utils.py
#
#-----------------------------------------------------
# This file contains the utilities needed for our program.

class Utils:
    def __init__(self):
        pass
```



```
    # Prompt user to enter to continue
    def press_enter_to_continue(self):
        input("\nPress enter key, to continue....\n")

    # Checks if given paranthesis is correctly parenthesized
    def checkParanthesis(self,expression):
        # Checks if expression starts with '(' and ends with ')'
        if not ((expression.startswith("(") and
expression.endswith(")"))):
            return False

        balance = 0
        # Iterate the whole expression to ensure the
        # number of '(' is the same as the number of ')'
        for char in expression:
            if char=="(":
                balance+=1
            elif char==")":
                balance -=1
            if balance < 0:
                return False

        return balance ==0
```

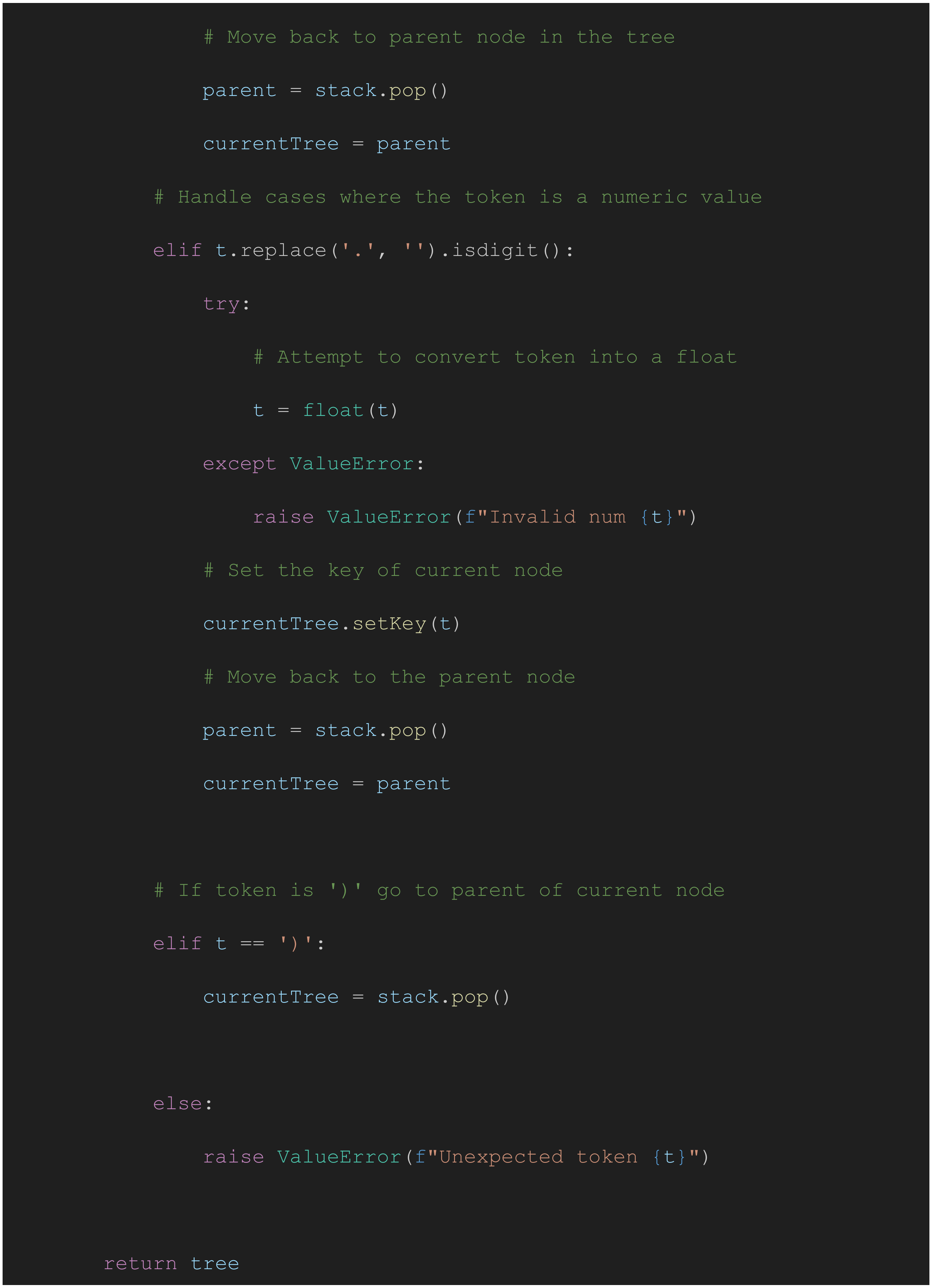
```
#-----------------------------------------------------
#
# Filename  : binaryTree.py
#
#-----------------------------------------------------
# This file contains binary class which represents the
# binary expression tree.


# The BinaryTree class represents a binary tree node and provides methods for tree manipulation and evaluation.
class BinaryTree:
    def __init__(self,key, leftTree = None, rightTree = None):
        self.key = key
        self.leftTree = leftTree
        self.rightTree = rightTree

    # Set the key value of the node.
    def setKey(self, key):
        self.key = key

    # Get the key value of the node.
    def getKey(self):
        return self.key

    # Get the left subtree of the node.
    def getLeftTree(self):
        return self.leftTree

    # Get the right subtree of the node.
    def getRightTree(self):
        return self.rightTree

    # Insert a new left subtree to the node.
    def insertLeft(self, key):
        # If there is no existing left subtree, create a new one with the
key
        if self.leftTree == None:
            self.leftTree = BinaryTree(key)
        else:
            t =BinaryTree(key)
            self.leftTree , t.leftTree = t, self.leftTree

    # Insert a new right subtree to the node.
    def insertRight(self, key):
        # If there is no existing right subtree, create a new one with the
key
```

```
        if self.rightTree == None:
            self.rightTree = BinaryTree(key)
        else:
            t =BinaryTree(key)
            self.rightTree , t.rightTree = t, self.rightTree

    # Print the tree in preorder traversal.
    def printPreorder(self, level):
        # Print the key of the current node based on tree levels
        print( str(level*'-') + str(self.key))
        # Recursively print the left subtree if it exists
        if self.leftTree != None:
            self.leftTree.printPreorder(level+1)

        # Recursively print the right subtree if it exists
        if self.rightTree != None:
            self.rightTree.printPreorder(level+1)

    # Evaluate the expression represented by the tree.
    def evaluate(self):
        if self.leftTree is None and self.rightTree is None:

            try:
                # If the current node has no children, try to convert its key to float
                return float(self.key)
            except ValueError:
                return None

        # Recursively evaluate the left and right subtrees and perform operation specified by the current node's key
        leftVal = self.leftTree.evaluate() if self.leftTree is not None else None
        rightVal = self.rightTree.evaluate() if self.rightTree is not None else None

        # Operations
        if self.key == '+':
            return leftVal + rightVal
        elif self.key == '-':
            return leftVal - rightVal
        elif self.key == '*':
            return leftVal * rightVal
        elif self.key == '/':
            if rightVal==0:
                raise ZeroDivisionError("Division by zero. Please enter a valid assignment")
            return leftVal / rightVal
```

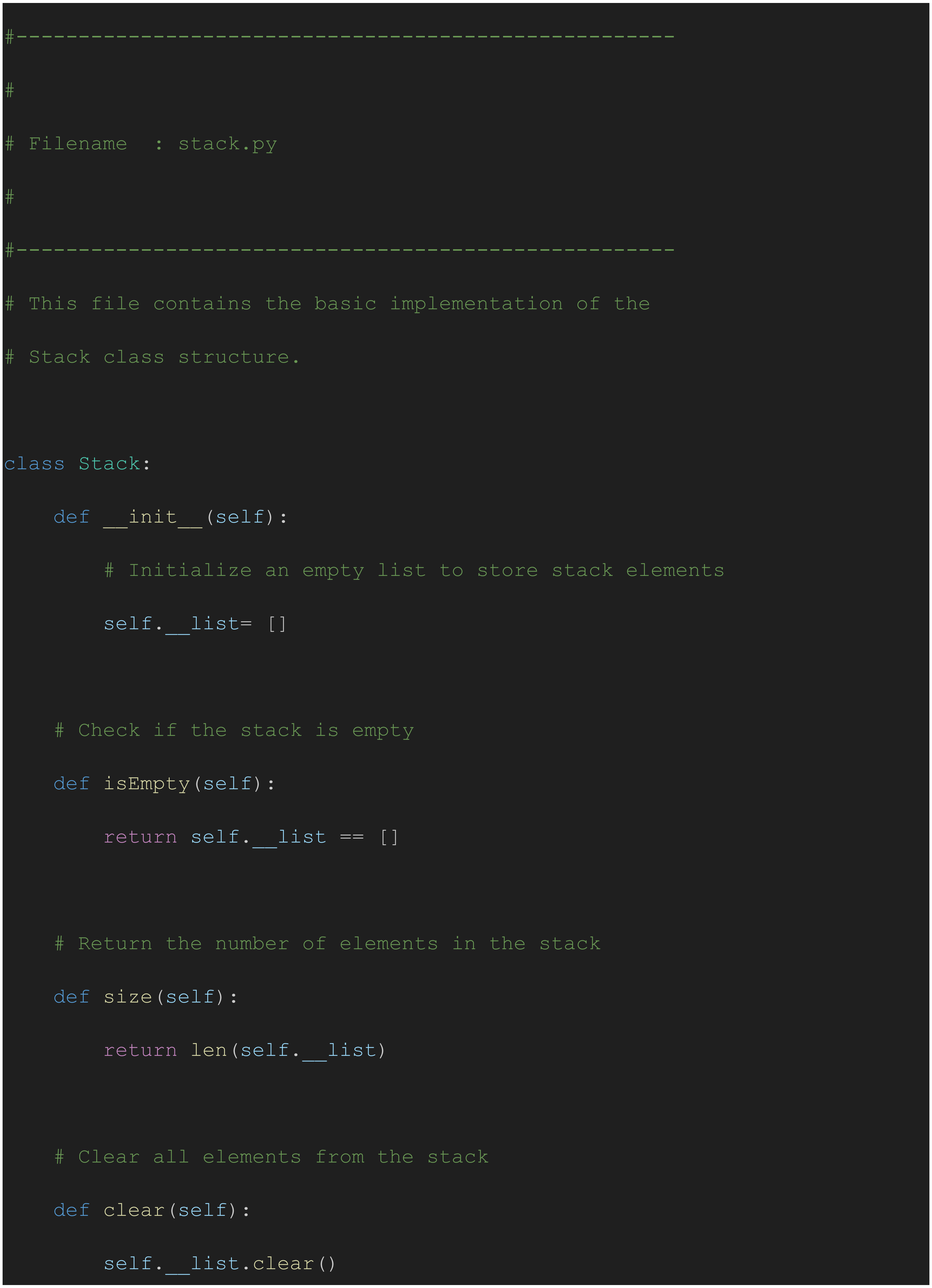


```python
        elif self.key =="**":
            return leftVal**rightVal

    # Print the tree in inorder traversal.
    def printInOrder(self,level=0):
        # Recursively print the right subtree if it exists.
        if self.rightTree:
            self.rightTree.printInOrder(level+1)

        # Print current node key with formatting.
        if isinstance(self.key,float):
            if self.key.is_integer():
                print(level*"." + str(int(self.key)))
            else:
                print(level*"." + str(float(self.key)))

        else:
            # Print current node's key if it is not a float
            print(level*"." + str(self.key))

        # Recursively print the left subtree if it exists
        if self.leftTree:
            self.leftTree.printInOrder(level+1)
```

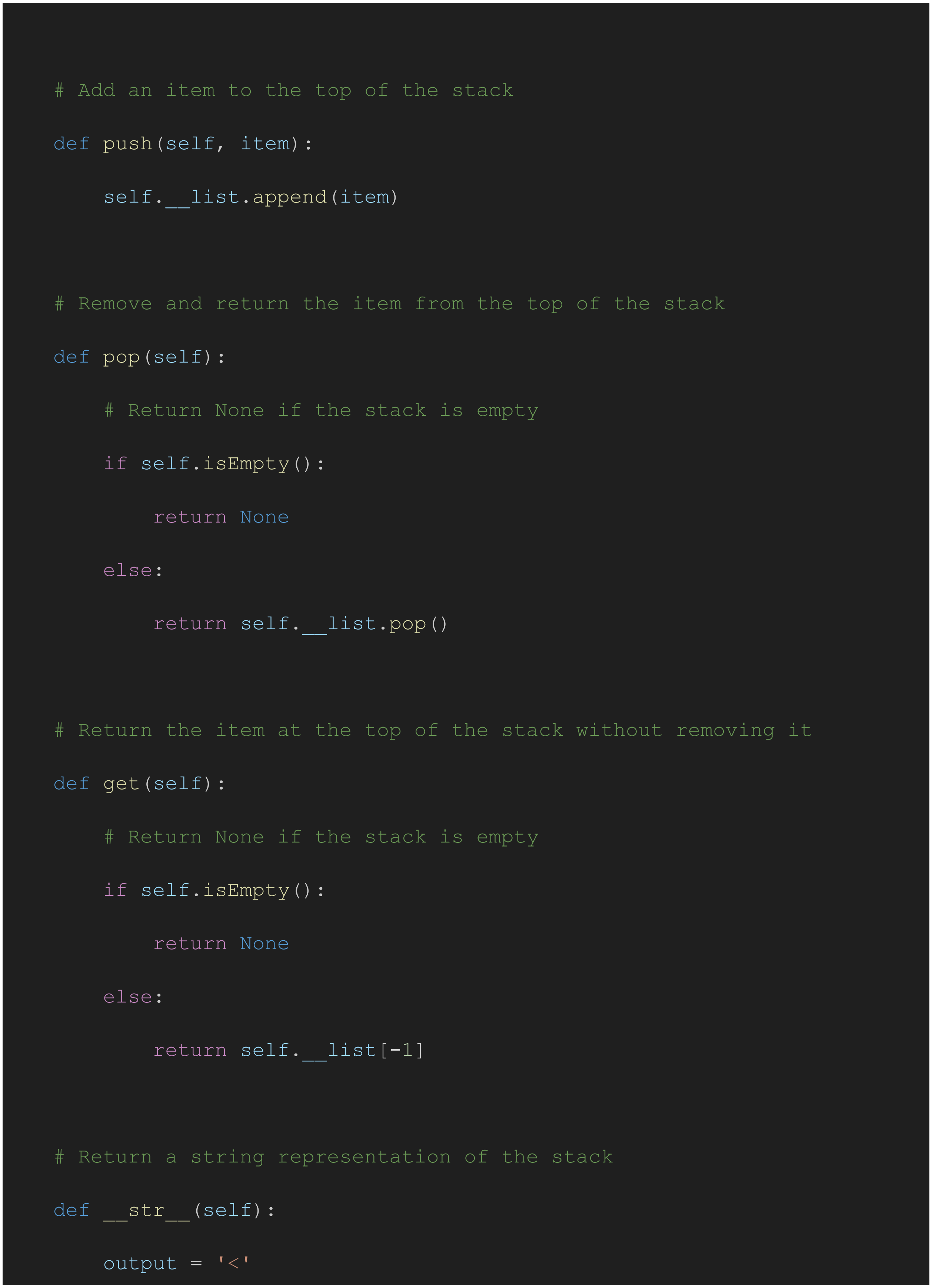
```python
#-----------------------------------------------------

#

# Filename  : parseTree.py

#

#-----------------------------------------------------

# This file contains the building of Parse Tree


from Tree.stack import Stack

from Tree.binaryTree import BinaryTree

from Tree.tokenizer import Tokenizer


# The ParseTree class represents a parse tree for mathematical expressions.

class ParseTree():

    # Constructor

    def __init__(self):

        pass


    # Build a parse tree from a mathematical expression.

    def buildParseTree(exp):

        # Tokenize the expression

        tokenizer = Tokenizer(exp)

        tokens = tokenizer.tokenize()
        # Check if the expression is in the form (number)
        if len(tokens) == 3 and tokens[1].replace('.', '').isdigit() and tokens[0] == "(" and tokens[2] == ")":
            return BinaryTree(float(tokens[1]))

        # Create a stack
        stack = Stack()
        tree = BinaryTree('?')
        stack.push(tree)
        currentTree = tree

        # Iterate through the tokens in the expression
        for t in tokens:
            # If token is '(' add a new node as left child
            # and descend into that node
            if t == '(':
                currentTree.insertLeft('?')
                stack.push(currentTree)
                currentTree = currentTree.getLeftTree()

            # If is alphabetic
            elif t.isalpha():
                # Set the key of the current node as the variable name
                currentTree.setKey(t)
                parent = stack.pop()
                currentTree = parent

            # If token is operator set key of current node
            # to that operator and add a new node as right child
            # and descend into that node
            elif t in ['+', '-', '*', '/', '**']:
                currentTree.setKey(t)
                currentTree.insertRight('?')
                stack.push(currentTree)
                currentTree = currentTree.getRightTree()

            # Handle cases where the token is enclosed within parenthesis but represents a float e.g (1,23)
            elif t[0] == '(' and t[-1] == ')' and t[1:-1].replace('.', '').isdigit():
                try:
                    # Attempt the convert token into a float
                    t = float(t[1:-1])
                except ValueError:
                    raise ValueError(f"Invalid num {t}")
                # Set key of current node
                currentTree.setKey(t)
```



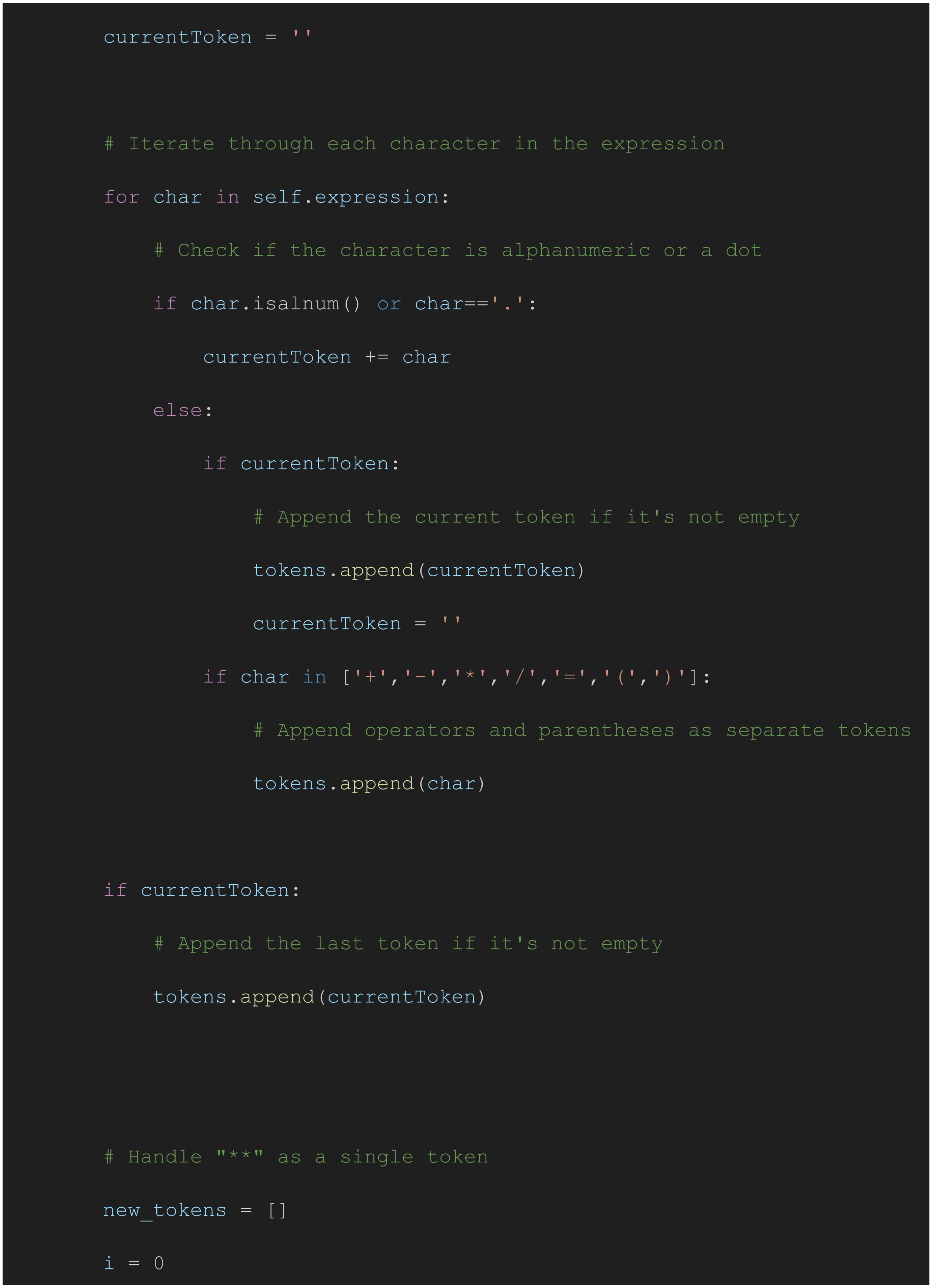
```python
            # Move back to parent node in the tree
            parent = stack.pop()
            currentTree = parent
        # Handle cases where the token is a numeric value
        elif t.replace('.', '').isdigit():
            try:
                # Attempt to convert token into a float
                t = float(t)
            except ValueError:
                raise ValueError(f"Invalid num {t}")
            # Set the key of current node
            currentTree.setKey(t)
            # Move back to the parent node
            parent = stack.pop()
            currentTree = parent

        # If token is ')' go to parent of current node
        elif t == ')':
            currentTree = stack.pop()

        else:
            raise ValueError(f"Unexpected token {t}")

    return tree
```



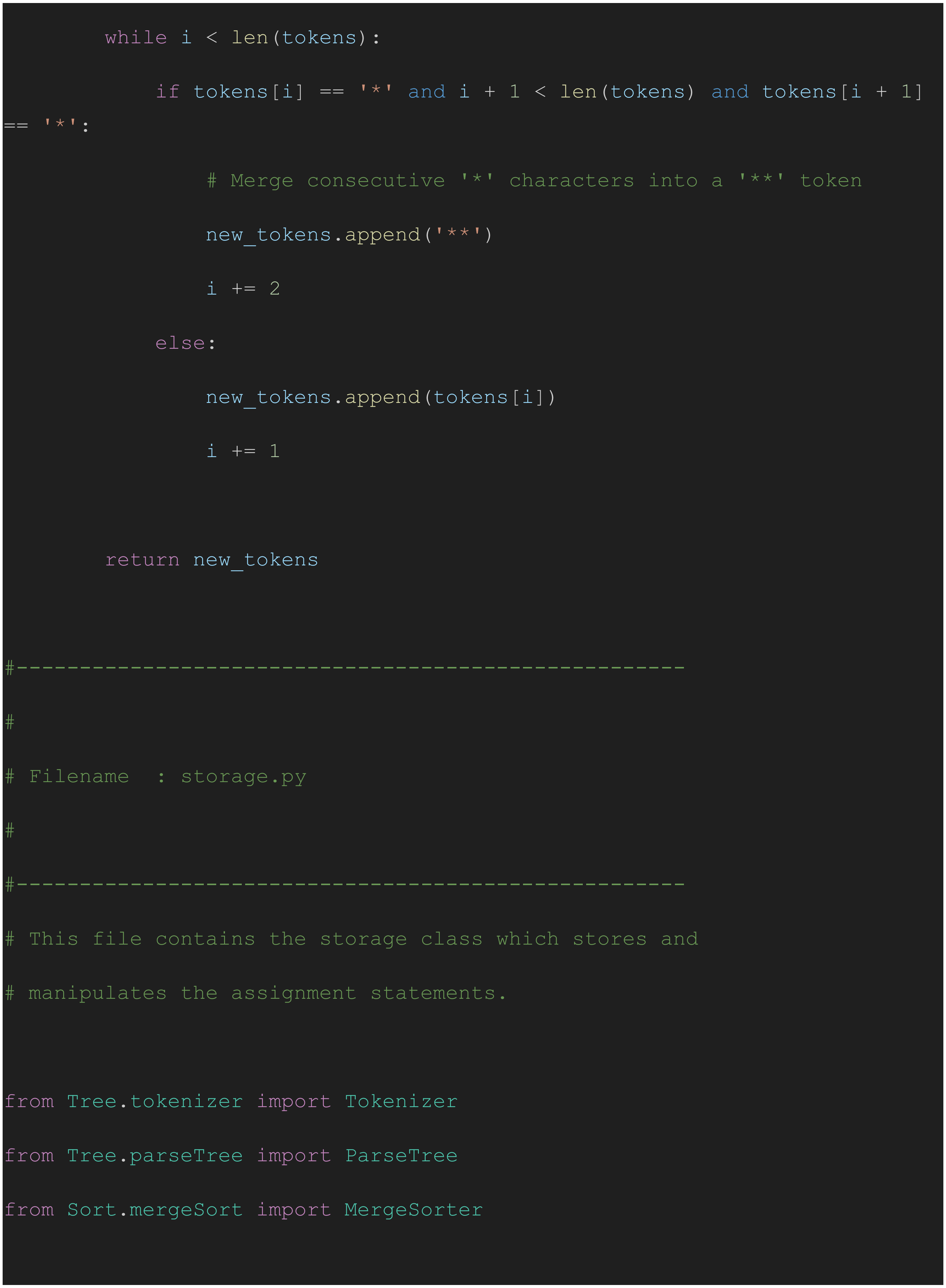
```python
#-----------------------------------------------------
#
# Filename  : stack.py
#
#-----------------------------------------------------
# This file contains the basic implementation of the
# Stack class structure.

class Stack:
    def __init__(self):
        # Initialize an empty list to store stack elements
        self.__list= []

    # Check if the stack is empty
    def isEmpty(self):
        return self.__list == []

    # Return the number of elements in the stack
    def size(self):
        return len(self.__list)

    # Clear all elements from the stack
    def clear(self):
        self.__list.clear()
```



```
    # Add an item to the top of the stack
    def push(self, item):
        self.__list.append(item)

    # Remove and return the item from the top of the stack
    def pop(self):
        # Return None if the stack is empty
        if self.isEmpty():
            return None
        else:
            return self.__list.pop()

    # Return the item at the top of the stack without removing it
    def get(self):
        # Return None if the stack is empty
        if self.isEmpty():
            return None
        else:
            return self.__list[-1]

    # Return a string representation of the stack
    def __str__(self):
        output = '<'
        for i in range( len(self.__list) ):
            item = self.__list[i]
            if i < len(self.__list)-1 :
                output += f'{str(item)}, '
            else:
                output += f'{str(item)}'
        output += '>'
        return output
#----------------------------------------------------
#
# Filename  : tokenizer.py
#
#----------------------------------------------------
# This file tokenizes the expression and returns a list
# of tokens

class Tokenizer:
    def __init__(self,expression):
        # Initialize the Tokenizer with the given expression
        self.expression = expression

    def tokenize(self):
        # Tokenize the expression and return a list of tokens
        tokens = []
```

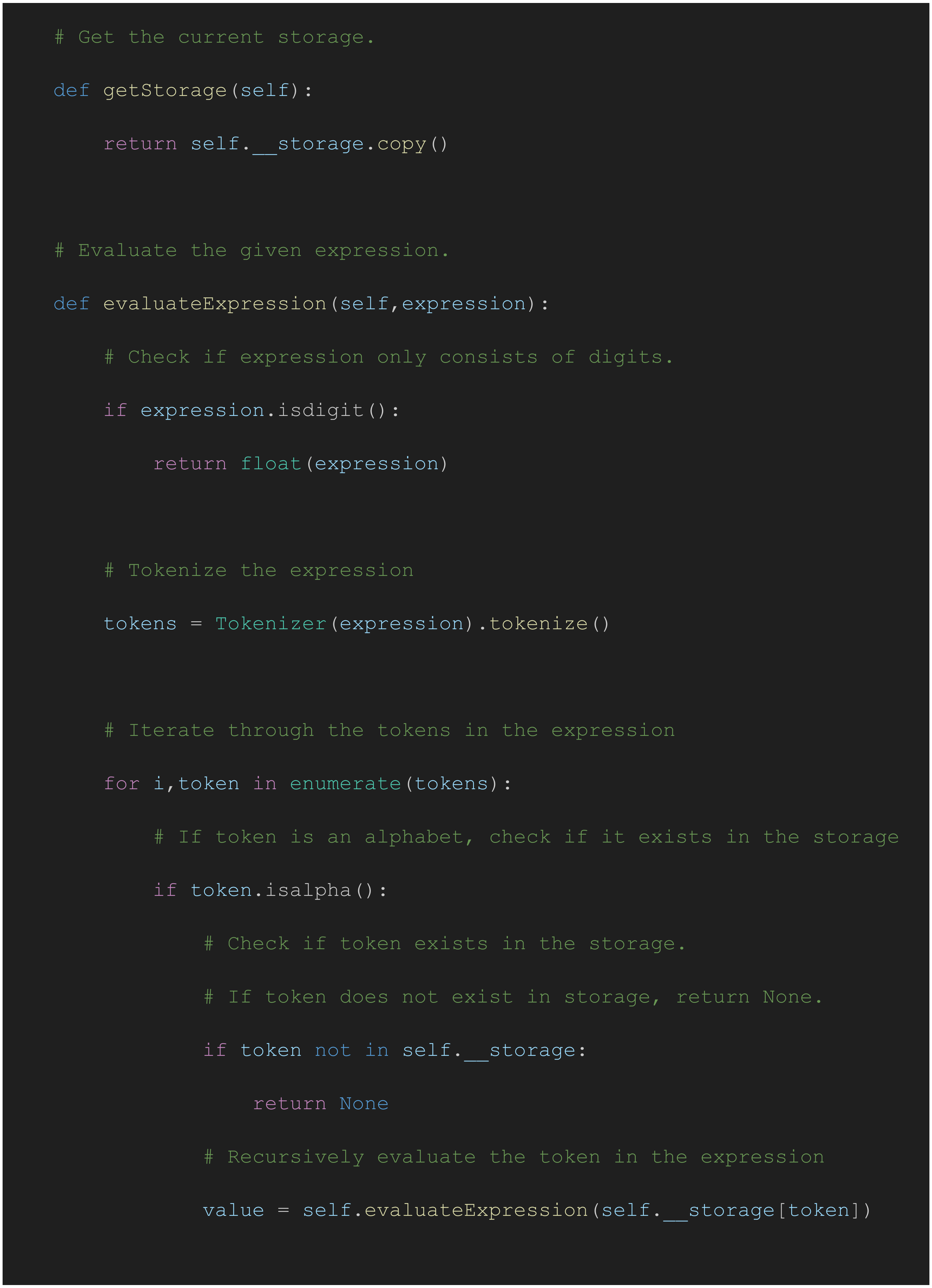


```
        currentToken = ''

        # Iterate through each character in the expression
        for char in self.expression:
            # Check if the character is alphanumeric or a dot
            if char.isalnum() or char=='.':
                currentToken += char
            else:
                if currentToken:
                    # Append the current token if it's not empty
                    tokens.append(currentToken)
                    currentToken = ''
                if char in ['+','-','*','/','=','(',')']:
                    # Append operators and parentheses as separate tokens
                    tokens.append(char)

        if currentToken:
            # Append the last token if it's not empty
            tokens.append(currentToken)


        # Handle "**" as a single token
        new_tokens = []
        i = 0
```



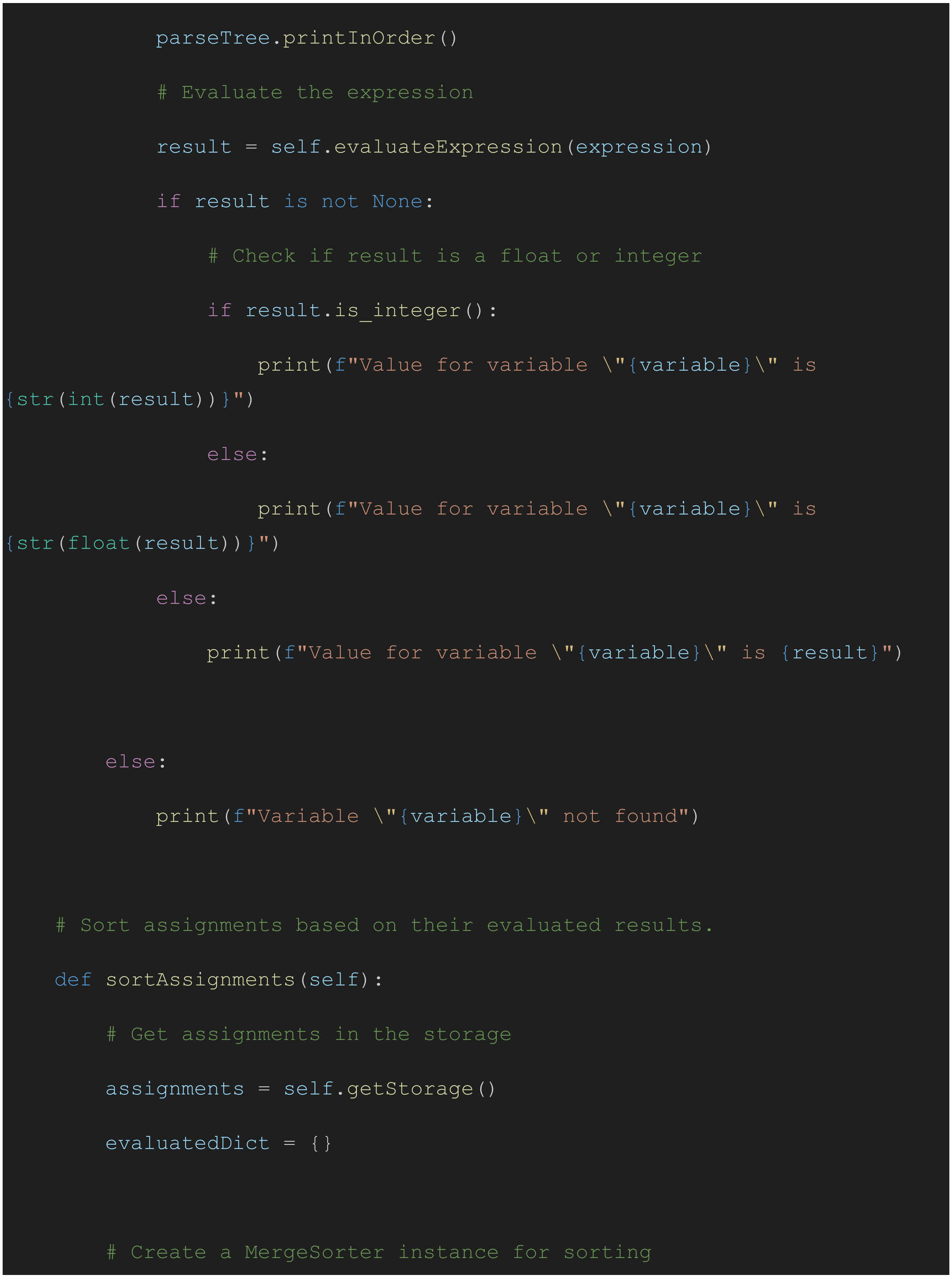
```
        while i < len(tokens):
            if tokens[i] == '*' and i + 1 < len(tokens) and tokens[i + 1]
== '*':
                # Merge consecutive '*' characters into a '**' token
                new_tokens.append('**')
                i += 2
            else:
                new_tokens.append(tokens[i])
                i += 1

        return new_tokens

#-----------------------------------------------------
#
#  Filename  : storage.py
#
#-----------------------------------------------------
#  This file contains the storage class which stores and
#  manipulates the assignment statements.

from Tree.tokenizer import Tokenizer
from Tree.parseTree import ParseTree
from Sort.mergeSort import MergeSorter
```

```
class Storage:
    # Initialize the storage, undo stack, and redo stack.
    def __init__(self):
        self.__storage = {}
        self.__undoStack=[]
        self.__redoStack=[]

    # Add a new assignment to the storage.
    def addAssignment(self,variable,value):
        # Save the current state of the storage for undo functionality
        self.saveCurrentUndoState()
        original_val = self.__storage.get(variable)
        success=True

        # Check for circular dependency before adding assignment
        if self.detectCircularDependency(variable,value):
            print(f"Assignment cannot be added. '{variable}' = '{value}'
due to Maximum Recursion Error.\n")
            return False
        try:
            # Attempt to evaluate the expression
            self.evaluateExpression(value)
            self.__storage[variable] = value
        except ZeroDivisionError:
            print(f"Assignment cannot be added. '{variable}' = '{value}' due to Zero Division Error.")
            success = False

        except Exception as e:
            print(f"Error: {e}")
            success=False
        # If assignment was unsuccessful, revert to the original state
        if not success:
            if original_val is not None:
                self.__storage[variable] = original_val

            else:
                del self.__storage[variable]
            self.__undoStack.pop()
        return success

    def detectCircularDependency(self,variable,value):
        # If variable is found in value, it means a circular dependency occurs.
        if variable in value:
            return True
        return False
```

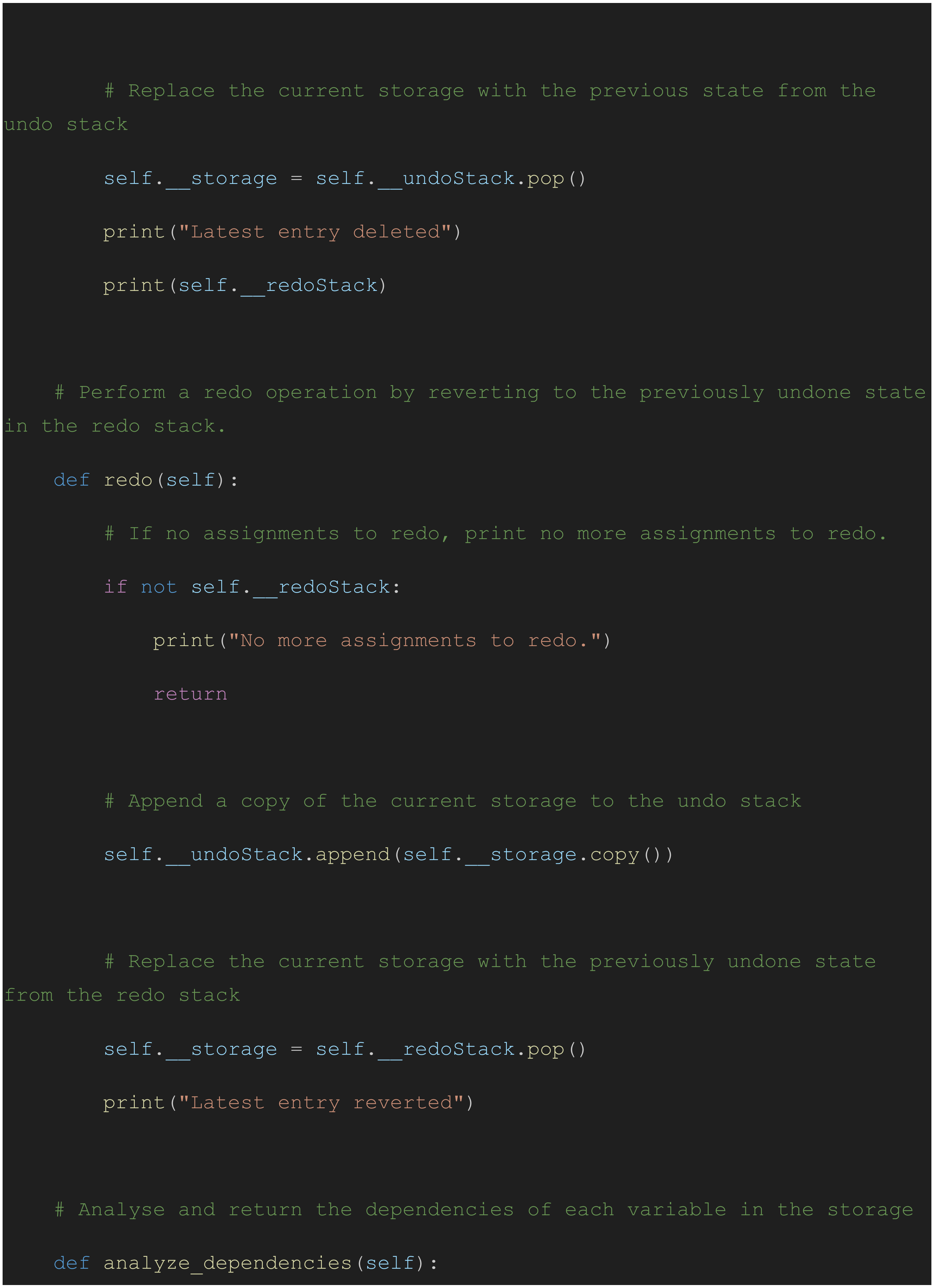


```
# Get the current storage.
def getStorage(self):
    return self.__storage.copy()

# Evaluate the given expression.
def evaluateExpression(self,expression):
    # Check if expression only consists of digits.
    if expression.isdigit():
        return float(expression)

    # Tokenize the expression
    tokens = Tokenizer(expression).tokenize()

    # Iterate through the tokens in the expression
    for i,token in enumerate(tokens):
        # If token is an alphabet, check if it exists in the storage
        if token.isalpha():
            # Check if token exists in the storage.
            # If token does not exist in storage, return None.
            if token not in self.__storage:
                return None
            # Recursively evaluate the token in the expression
            value = self.evaluateExpression(self.__storage[token])
```

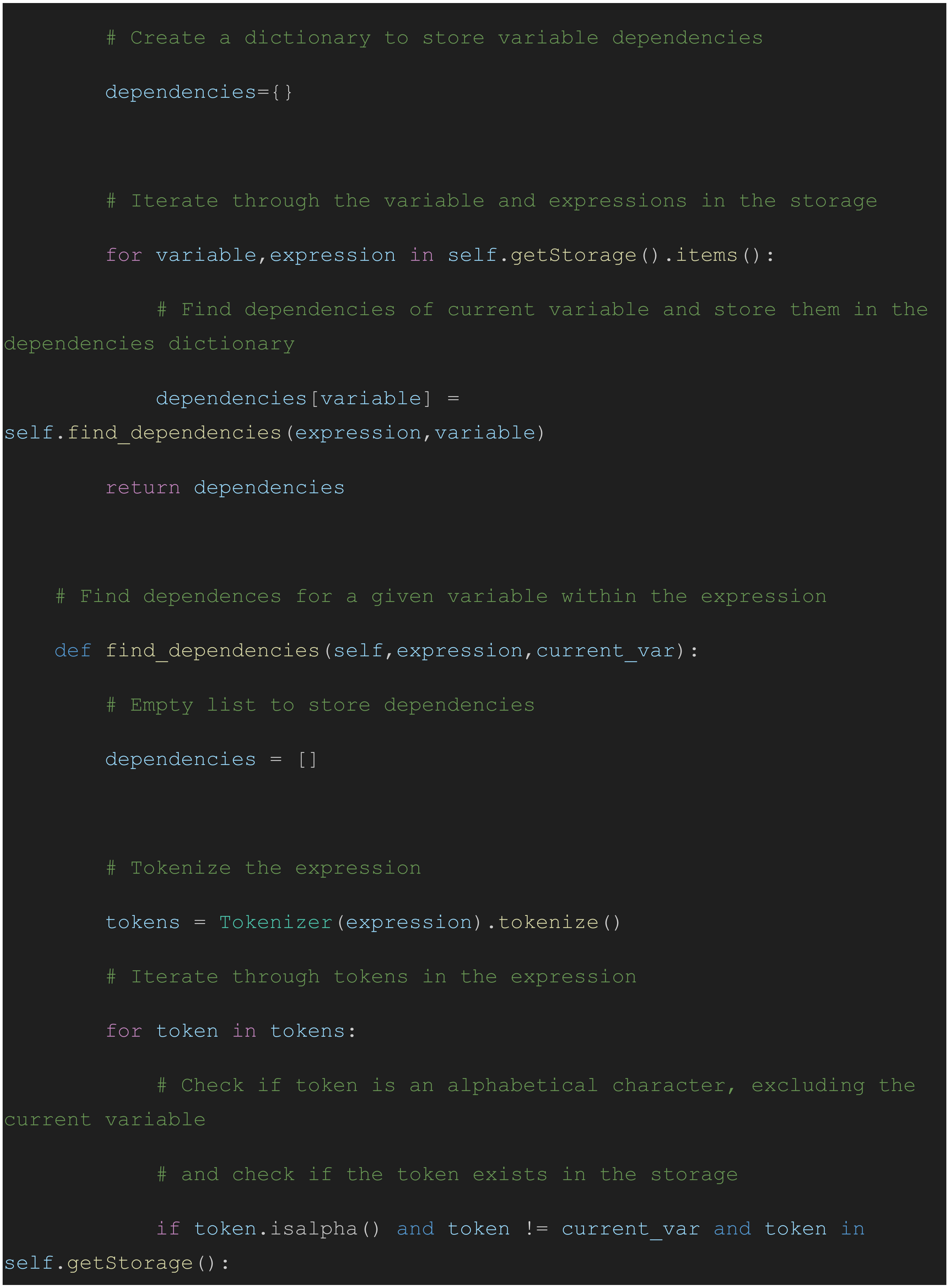
```
                # If evaluated results is None return None
                if value is None:
                    return None
                # Replace the token in the expression with its evaluated value
                tokens[i] = str(value)
        # Reconstruct the expression
        newExpression = ''.join(tokens)
        # Build a parse tree from the new expression
        tree = ParseTree.buildParseTree(newExpression)
        # Evaluate the parse tree and return the final result
        result = tree.evaluate()
        return result

    # Print the result of an assignment in a formatted manner
    def printFormattedResults(self,variable,expression,result):
        if result is not None:
            # Check if result is a float or integer
            if result.is_integer():
                print(f"{variable}={expression}=> {str(int(result))}")
            else:
                print(f"{variable}={expression}=> {str(float(result))}")
        else:
            print(f"{variable}={expression}=> {result}")
    # Display all current assignments and their results.
    def displayAssignments(self):
        assignments = self.getStorage()
        print("\nCURRENT ASSIGNMENTS:\n********************")
        for variable,originalExpression in sorted(assignments.items()):
            try:
                # Evaluate and print each assignment's result
                result = self.evaluateExpression(originalExpression)

self.printFormattedResults(variable,originalExpression,result)
            except Exception as e:
                print(f"Error: {e}")


    # Evaluate a specific variable and display its expression tree.
    def evaluateVariable(self,variable):
        # Check if variable exists in the storage
        if variable in self.getStorage():
            # Retrieve the variable in the storage
            expression = self.getStorage()[variable]
            print("\nExpression Tree")
            # Build parse tree from the expression
            parseTree = ParseTree.buildParseTree(expression)
            # Print the parse tree in-order travesal
```



```
            parseTree.printInOrder()
            # Evaluate the expression
            result = self.evaluateExpression(expression)
            if result is not None:
                # Check if result is a float or integer
                if result.is_integer():
                    print(f"Value for variable \"{variable}\" is {str(int(result))}")
                else:
                    print(f"Value for variable \"{variable}\" is {str(float(result))}")
            else:
                print(f"Value for variable \"{variable}\" is {result}")

        else:
            print(f"Variable \"{variable}\" not found")

    # Sort assignments based on their evaluated results.
    def sortAssignments(self):
        # Get assignments in the storage
        assignments = self.getStorage()
        evaluatedDict = {}

        # Create a MergeSorter instance for sorting
        sorter = MergeSorter()


        # Iterate through each variable and its corresponding expression

        for var, exp in assignments.items():


            # Evaluate the expression and get the results

            result = self.evaluateExpression(exp)


            # Check if result is in the evaluated dictionary

            if result not in evaluatedDict:

                evaluatedDict[result] = []


            # Append the variable and expression tuple to the result list
in the dictionary

            evaluatedDict[result].append((var, exp))


        # Create a dictionary to store the sorted assignments

        sorted_assignments = {result:
sorter.mergeSort(eval_group.copy(),'asc') for result, eval_group in
evaluatedDict.items()}

        return sorted_assignments




    # Kien Yu Redo/Undo

    # Save the current state of the storage for undo functionality.
    # Return the undo stack with previous states of the storage
    def getUndoStack(self):
        return self.__undoStack

    # Returns the redo stack with previously undone states of the storage
    def getRedoStack(self):
        return self.__redoStack

    # Save the current state of the storage in the undo state
    def saveCurrentUndoState(self):
        self.__undoStack.append(self.__storage.copy())

    # Perform an undo operation by reverting to the previous  state in the
undo stack.
    def undo(self):
        # If there are no more undo state, print no more assignments to
undo
        if not self.__undoStack:
            print("No more assignments to undo.")
            return

        # Append a copy of the current storage to the redo stack
        self.__redoStack.append(self.__storage.copy())
```



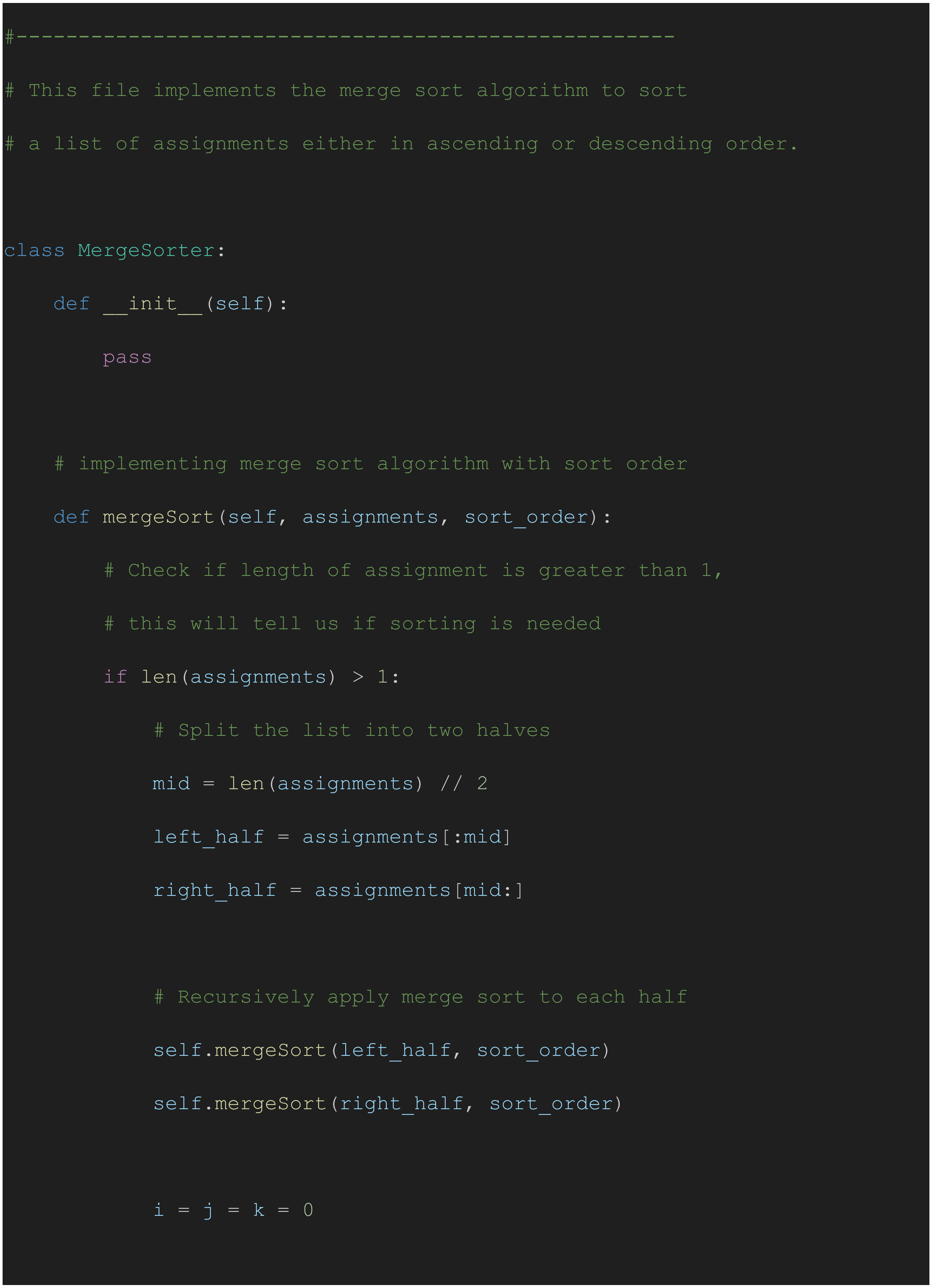
```
        # Replace the current storage with the previous state from the
undo stack
        self.__storage = self.__undoStack.pop()
        print("Latest entry deleted")
        print(self.__redoStack)

    # Perform a redo operation by reverting to the previously undone state
in the redo stack.
    def redo(self):
        # If no assignments to redo, print no more assignments to redo.
        if not self.__redoStack:
            print("No more assignments to redo.")
            return

        # Append a copy of the current storage to the undo stack
        self.__undoStack.append(self.__storage.copy())

        # Replace the current storage with the previously undone state
from the redo stack
        self.__storage = self.__redoStack.pop()
        print("Latest entry reverted")

    # Analyse and return the dependencies of each variable in the storage
    def analyze_dependencies(self):
```



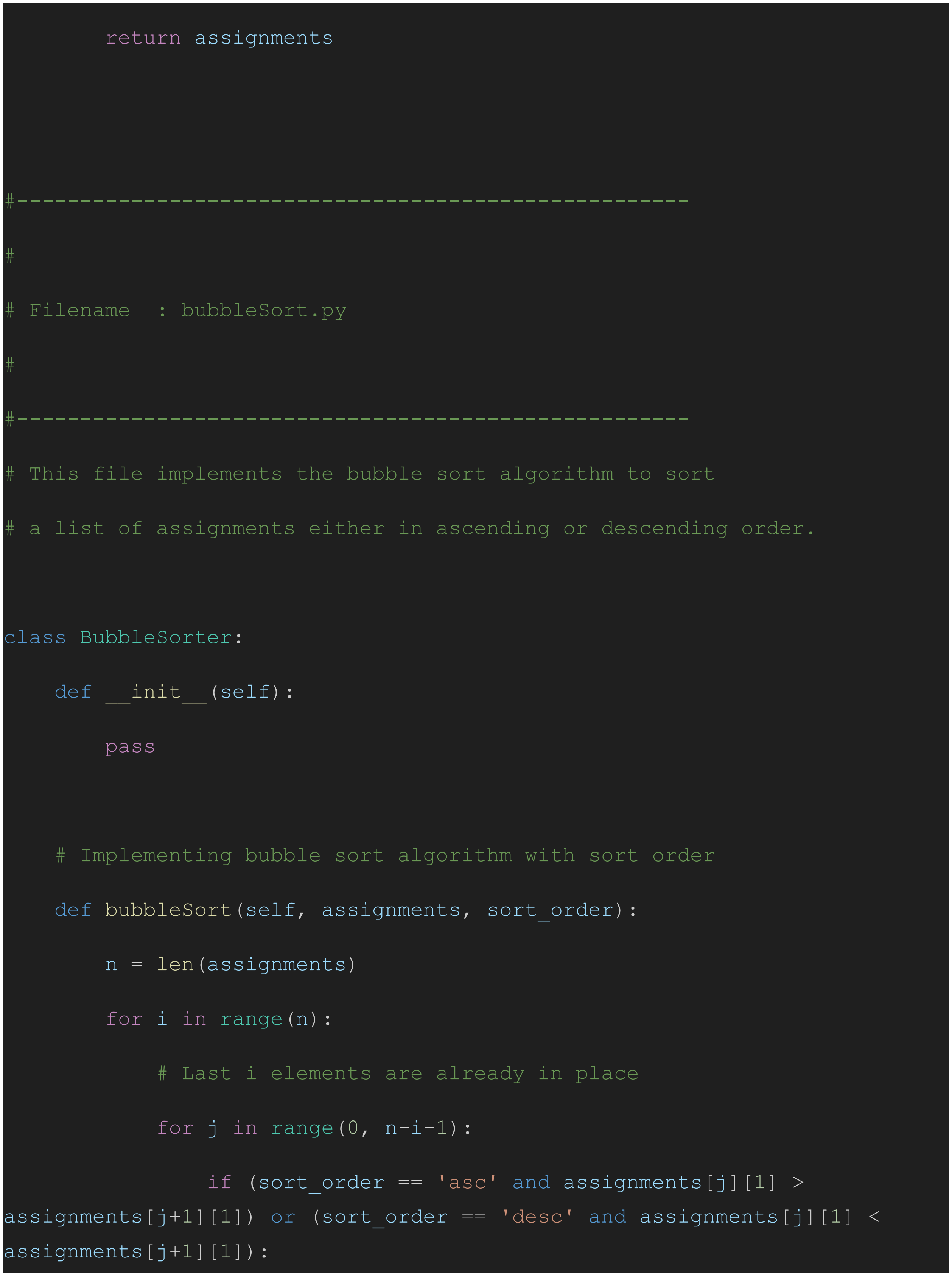
```
        # Create a dictionary to store variable dependencies
        dependencies={}

        # Iterate through the variable and expressions in the storage
        for variable,expression in self.getStorage().items():
            # Find dependencies of current variable and store them in the dependencies dictionary
            dependencies[variable] = self.find_dependencies(expression,variable)
        return dependencies

    # Find dependences for a given variable within the expression
    def find_dependencies(self,expression,current_var):
        # Empty list to store dependencies
        dependencies = []

        # Tokenize the expression
        tokens = Tokenizer(expression).tokenize()
        # Iterate through tokens in the expression
        for token in tokens:
            # Check if token is an alphabetical character, excluding the current variable
            # and check if the token exists in the storage
            if token.isalpha() and token != current_var and token in self.getStorage():
                dependencies.append(token)


        # Remove duplicates bu converting the list to a set then back to a list again

        return list(set(dependencies))


    # Display variable dependencies for each variable in the storage.

    def display_dependencies(self):

        # Analyze variable dependencies and store result in a dictionary

        dependencies = self.analyze_dependencies()


        # Get the assignments from storage

        assignments = self.getStorage()


        # Print header

        print("\nVARIABLE DEPENDENCIES\n*********************")


        # Iterate through variables and their expression in a sorted order

        for variable,originalExpression in sorted(assignments.items()):

            try:

                # Evaluate the expression to obtain result

                result=self.evaluateExpression(originalExpression)


                # Print formatted results for the variable
```

```
self.printFormattedResults(variable,originalExpression,result)


                # Get variable dependencies for the current variable

                variable_dependencies = dependencies.get(variable,[])


                # Print variable dependences or if there are no
dependencies

                if variable_dependencies:

                    print(f"Variable '{variable}' depends on: {',
'.join(variable_dependencies)}\n")


                else:

                    print(f"Variable '{variable}' depends on: No
Dependencies.\n")


            except Exception as e:

                print(f"Error {e}")


    def writeDependencyToFile(self,outputFile):

        # Analyze variable dependencies and store result in a dictionary

        dependencies = self.analyze_dependencies()

        # Get the assignments from storage

        assignments = self.getStorage()
        with open(outputFile,'w') as file:
            file.write("VARIABLE DEPENDENCIES\n*********************\n")

            for variable,originalExpression in
sorted(assignments.items()):
                try:
                    # Evaluate the expression to obtain result
                    result=self.evaluateExpression(originalExpression)
                    # Format as integer if its a whole number, else keep
it as a float
                    formattedResult = int(result) if result is not None
and result.is_integer() else result
                    outputStr = f"{variable}={originalExpression}=>
{formattedResult if result is not None else 'None'}\n"
                    file.write(outputStr)

                    # Get variable dependencies for the current variable
                    variable_dependencies = dependencies.get(variable,[])

                    if variable_dependencies:
                        file.write(f"Variable '{variable}' depends on: {',
'.join(variable_dependencies)}\n\n")
                    else:
                        file.write(f"Variable '{variable}' depends on: No
Dependencies.\n\n")
```

```
                except Exception as e:
                    print(f"Error {e}")


    # Retrieve all current assignments and their results. (Kallen's additional feature)
    def retrieveAllAssignments(self):
        assignments = self.getStorage()
        formatted_assignments = []

        for variable, originalExpression in sorted(assignments.items()):
            try:
                result = self.evaluateExpression(originalExpression)
                formatted_assignments.append((variable, [result]))  # Ensure result is in a list
            except Exception as e:
                print(f"Error: {e}")

        return formatted_assignments

#-----------------------------------------------------
#
# Filename  : mergeSort.py
#
```

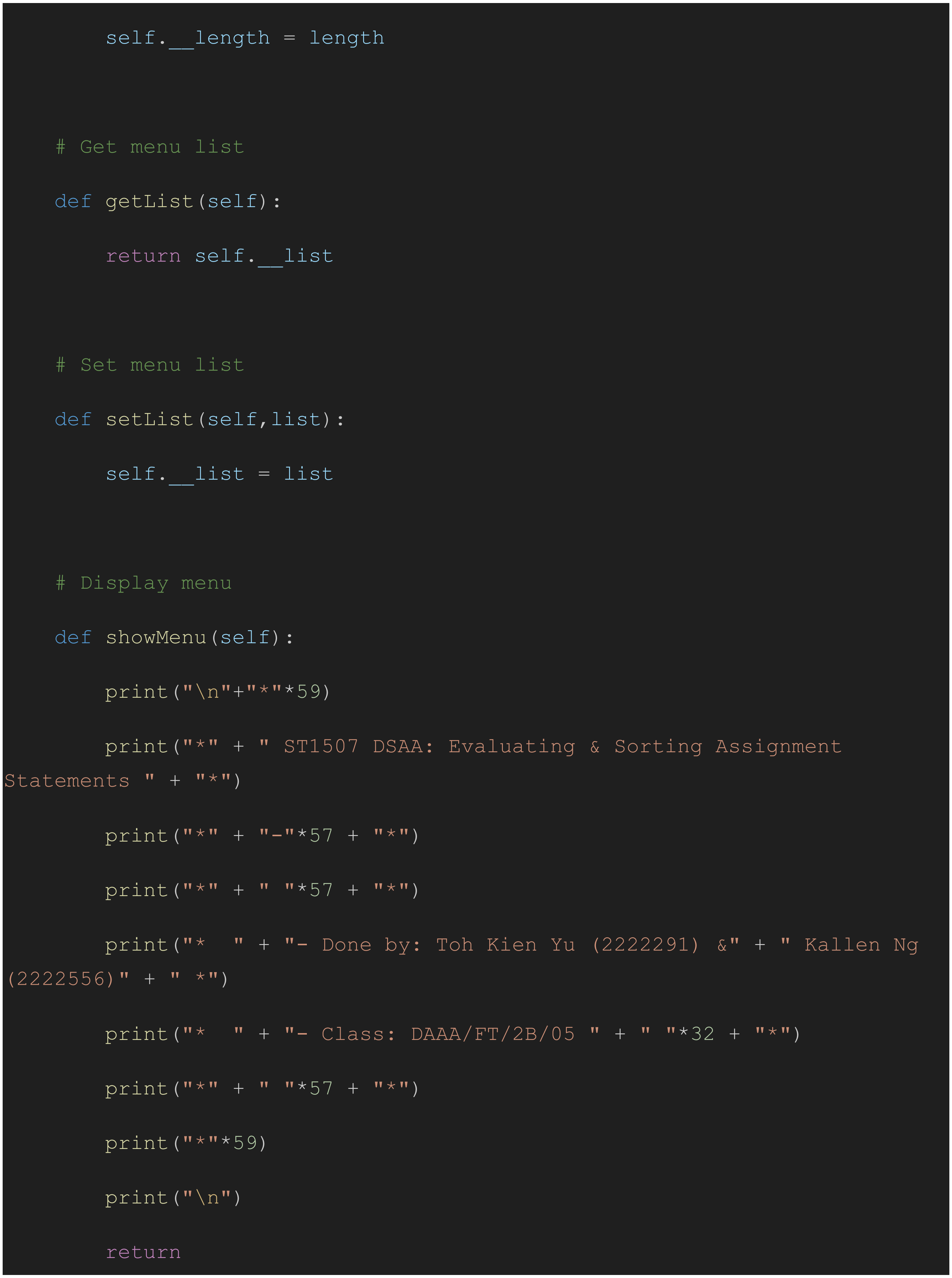


```python
#-----------------------------------------------------

# This file implements the merge sort algorithm to sort

# a list of assignments either in ascending or descending order.


class MergeSorter:

    def __init__(self):

        pass


    # implementing merge sort algorithm with sort order

    def mergeSort(self, assignments, sort_order):

        # Check if length of assignment is greater than 1,

        # this will tell us if sorting is needed

        if len(assignments) > 1:

            # Split the list into two halves

            mid = len(assignments) // 2

            left_half = assignments[:mid]

            right_half = assignments[mid:]


            # Recursively apply merge sort to each half

            self.mergeSort(left_half, sort_order)

            self.mergeSort(right_half, sort_order)


            i = j = k = 0
            # Merge the two halves back into a sorted list
            while i < len(left_half) and j < len(right_half):
                # Compare and merge based on sort order, either ascending or descending
                if (sort_order == 'asc' and left_half[i][0] < right_half[j][0]) or (sort_order == 'desc' and left_half[i][0] > right_half[j][0]):
                    assignments[k] = left_half[i]
                    i += 1
                else:
                    assignments[k] = right_half[j]
                    j += 1
                k += 1
            # Handle any remaining elements in the left half
            while i < len(left_half):
                assignments[k] = left_half[i]
                i += 1
                k += 1

            # Handle any remaining elements in the right half
            while j < len(right_half):
                assignments[k] = right_half[j]
                j += 1
                k += 1
```

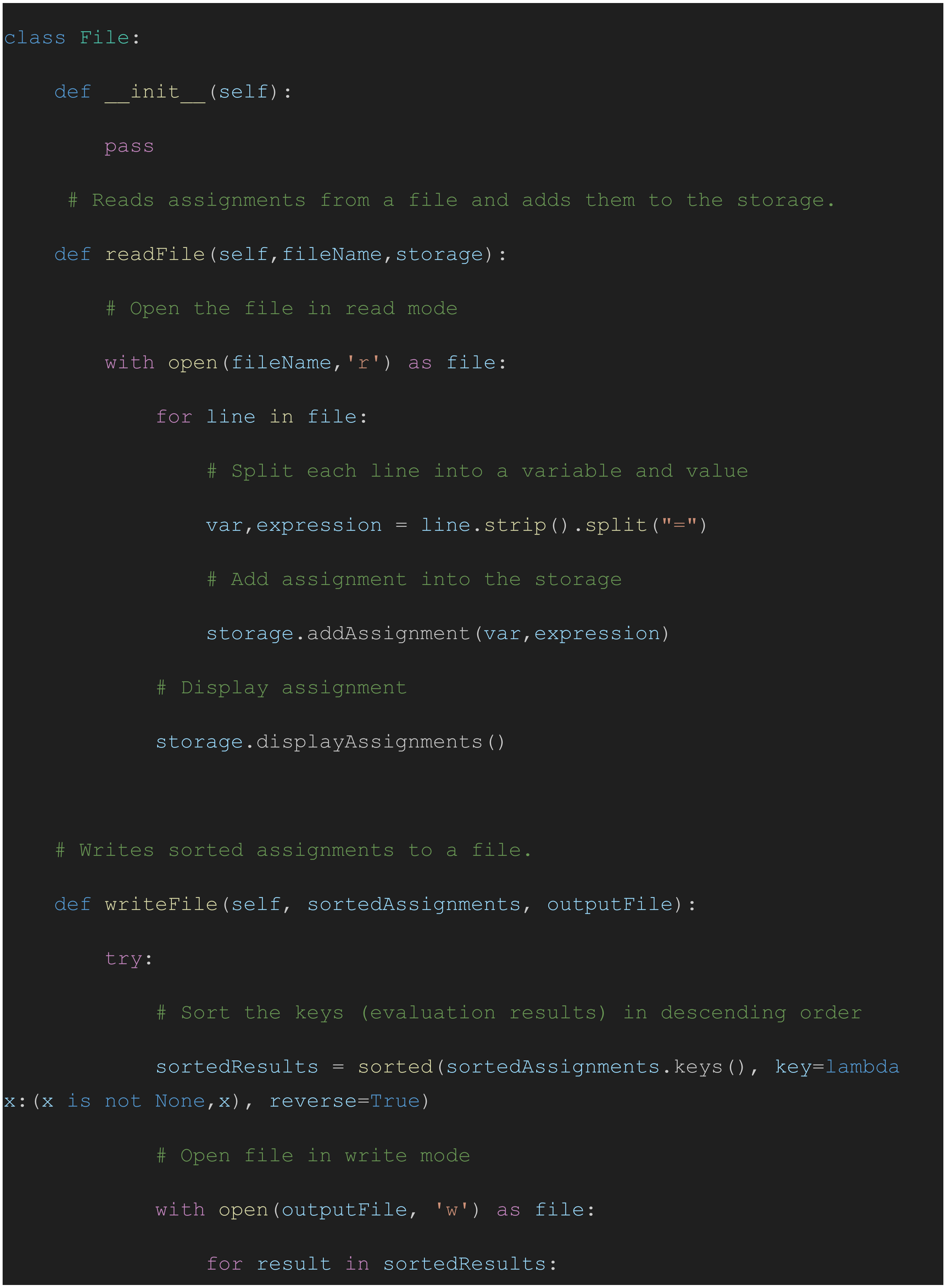


```
        return assignments


#-----------------------------------------------------
#
# Filename  : bubbleSort.py
#
#-----------------------------------------------------
# This file implements the bubble sort algorithm to sort
# a list of assignments either in ascending or descending order.

class BubbleSorter:
    def __init__(self):
        pass

    # Implementing bubble sort algorithm with sort order
    def bubbleSort(self, assignments, sort_order):
        n = len(assignments)
        for i in range(n):
            # Last i elements are already in place
            for j in range(0, n-i-1):
                if (sort_order == 'asc' and assignments[j][1] >
assignments[j+1][1]) or (sort_order == 'desc' and assignments[j][1] <
assignments[j+1][1]):
```

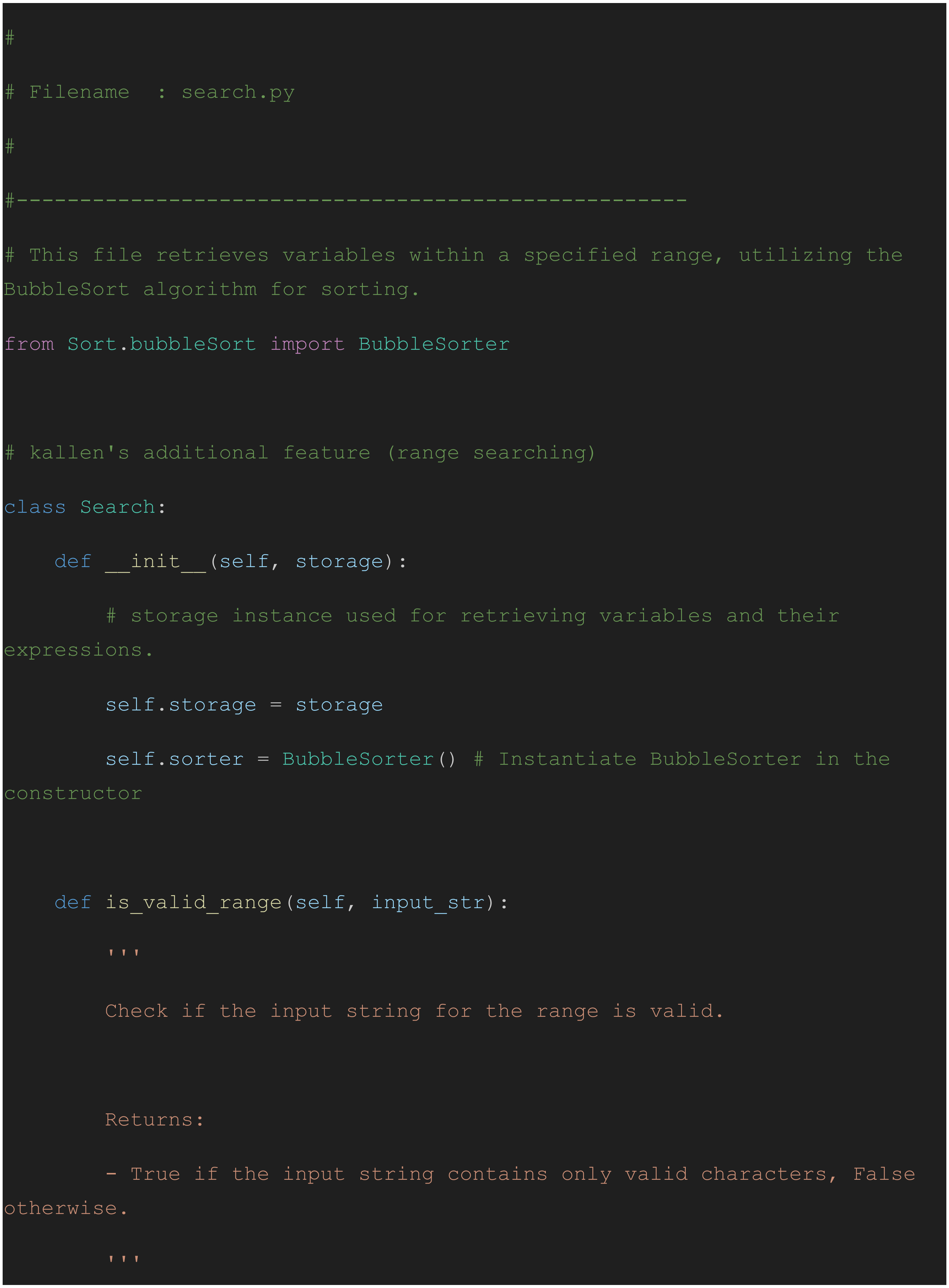
```
                    assignments[j], assignments[j+1] = assignments[j+1], 
assignments[j]
        return assignments
#----------------------------------------------------
#
# Filename  : menu.py
#
#----------------------------------------------------
# This file handles the displaying of main menu such
# as inserting items into the menu and displaying of options.

class Menu:
    # Constructor
    def __init__(self,options):
        self.__length = 0
        self.__list = []
        self.options = options
        return
    # Get menu length
    def getLength(self):
        return self.__length

    # Set menu length
    def setLength(self,length):
```



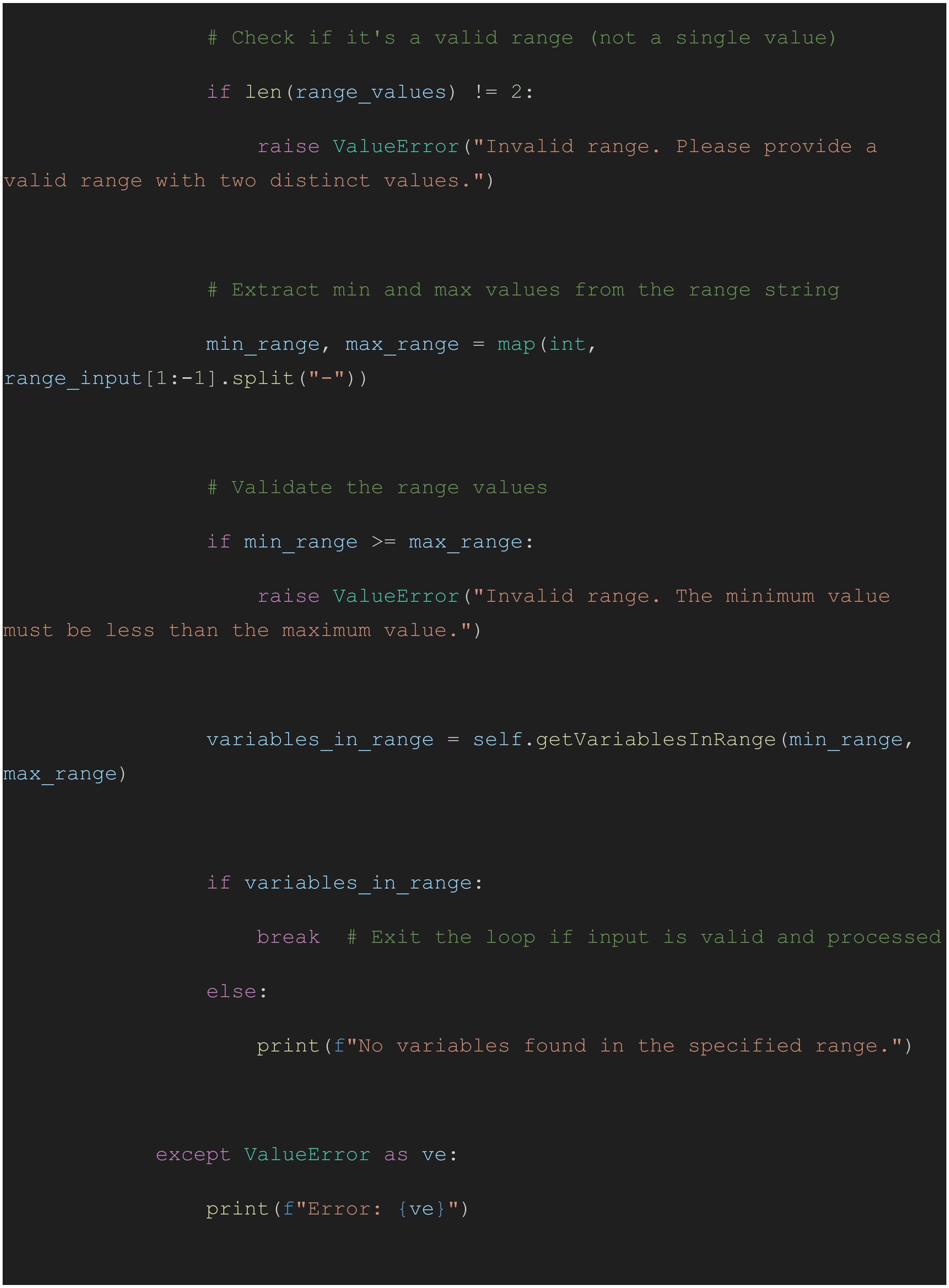
```
        self.__length = length

    # Get menu list
    def getList(self):
        return self.__list

    # Set menu list
    def setList(self,list):
        self.__list = list

    # Display menu
    def showMenu(self):
        print("\n"+"*"*59)
        print("*" + " ST1507 DSAA: Evaluating & Sorting Assignment Statements " + "*")
        print("*" + "-"*57 + "*")
        print("*" + " "*57 + "*")
        print("*  " + "- Done by: Toh Kien Yu (2222291) &" + " Kallen Ng (2222556)" + " *")
        print("*  " + "- Class: DAAA/FT/2B/05 " + " "*32 + "*")
        print("*" + " "*57 + "*")
        print("*"*59)
        print("\n")
        return
    # Insert items into menu
    def insert(self,list):
        for i in list:
            self.__list.append(i)
            self.__length +=1
        return

    # Show menu options
    def showOptions(self):
        choices = ','.join(f"'{i}'" for i in range(1,self.options+1))
        print(f"Please select your choice: ({choices})")
        # Loop through the list and display the options
        for i in range(len(self.__list)):
            print(f"\t{i+1}. {self.__list[i]}")
        return
#-----------------------------------------------------
#
# Filename  : file.py
#
#-----------------------------------------------------
# This file provides functinality for file processing
# such as reading a file and writing to a file
```



```
class File:
    def __init__(self):
        pass
     # Reads assignments from a file and adds them to the storage.
    def readFile(self,fileName,storage):
        # Open the file in read mode
        with open(fileName,'r') as file:
            for line in file:
                # Split each line into a variable and value
                var,expression = line.strip().split("=")
                # Add assignment into the storage
                storage.addAssignment(var,expression)
            # Display assignment
            storage.displayAssignments()

    # Writes sorted assignments to a file.
    def writeFile(self, sortedAssignments, outputFile):
        try:
            # Sort the keys (evaluation results) in descending order
            sortedResults = sorted(sortedAssignments.keys(), key=lambda x:(x is not None,x), reverse=True)
            # Open file in write mode
            with open(outputFile, 'w') as file:
                for result in sortedResults:
                    evalGroup = sortedAssignments[result]
                    # Format and write the result and evaluated group to
the output file
                    file.write(self.formatGroup(result,evalGroup))
        except Exception as e:
            print(f"Error writting file {e}")

    # Format how the information is displayed
    def formatGroup(self,result,evaluatedGroup):
        formatString = ""
        # If result is not None, format it to either a float or integer
        if result is not None:
            resultStr = f"*** Statements with value=> {int(result)}\n" if
result.is_integer() else f"***Statement with value=> {float(result)}\n"
        else:
            # If result is None just write None
            resultStr = f"*** Statements with value=> {result}\n"
        formatString += resultStr
        # Add each variable and its corresponding expression to
formatString
        for var,exp in evaluatedGroup:
            formatString += f"{var}={exp}\n"
        formatString += "\n"
        return formatString
#-----------------------------------------------------
```



```
#
# Filename  : search.py
#
#-----------------------------------------------------
# This file retrieves variables within a specified range, utilizing the BubbleSort algorithm for sorting.
from Sort.bubbleSort import BubbleSorter


# kallen's additional feature (range searching)
class Search:
    def __init__(self, storage):
        # storage instance used for retrieving variables and their expressions.
        self.storage = storage
        self.sorter = BubbleSorter() # Instantiate BubbleSorter in the constructor


    def is_valid_range(self, input_str):
        '''
        Check if the input string for the range is valid.


        Returns:
        - True if the input string contains only valid characters, False otherwise.
        '''
        allowed_chars = set("0123456789-()")
        return all(char in allowed_chars for char in input_str)

    def getVariablesInRange(self, min_value, max_value):
        # Retrieve variables within the specified range.
        variables_in_range = []

        # Iterate over variables in storage
        for variable, expression in self.storage.getStorage().items():
            result = self.storage.evaluateExpression(expression)
            # Check if the result is within the specified range
            if result is not None and min_value <= result <= max_value:
                variables_in_range.append(variable)

        return variables_in_range

    def getRange(self):
        '''
        Get user input for the range, validate it, and retrieve variables within the range.
        Additionally, prompt the user for sorting options and display the sorted variables.
        '''
        while True:
            try:
                # Prompt user for the range in the format (min-max)
                range_input = input("\nEnter the range you wish to search
(min-max), or enter 'X' to return to the main menu:\nFor example,
(1-10)\n")

                # Check if the user wants to return to the main menu
                if range_input.lower() == 'x':
                    print("Returning to the main menu.")
                    return  # Exit the function

                # Validate the format of the range
                if not range_input.startswith('(') or not
range_input.endswith(')') or ' ' in range_input:
                    raise ValueError("Invalid range format. Please use
brackets and avoid spacing.")

                # Check if only allowed characters are entered
                if not self.is_valid_range(range_input):
                    raise ValueError("Invalid characters in the range.
Please enter only numbers, brackets, and hyphens.")

                # Extract min and max values from the range string
                range_values = range_input[1:-1].split("-")
```



```
                # Check if it's a valid range (not a single value)
                if len(range_values) != 2:
                    raise ValueError("Invalid range. Please provide a
valid range with two distinct values.")

                # Extract min and max values from the range string
                min_range, max_range = map(int,
range_input[1:-1].split("-"))

                # Validate the range values
                if min_range >= max_range:
                    raise ValueError("Invalid range. The minimum value
must be less than the maximum value.")

                variables_in_range = self.getVariablesInRange(min_range,
max_range)

                if variables_in_range:
                    break  # Exit the loop if input is valid and processed
                else:
                    print(f"No variables found in the specified range.")

            except ValueError as ve:
                print(f"Error: {ve}")
```

```
        # Loop for sorting order input
        while True:
            try:
                # Prompt user for sorting
                sort_order = input("\nHow do you want to arrange the
variables? Type 'asc' for ascending order or 'desc' for descending
order:").lower()

                # Validate the sorting order
                if sort_order not in ['asc', 'desc']:
                    raise ValueError("Invalid sorting order. Please enter
'asc' for ascending or 'desc' for descending.")

                sorted_assignments =
self.sortVariables(variables_in_range, sort_order)

                order_text = "ascending" if sort_order == "asc" else
"descending"

                # Display both variable names and evaluated values
                formatted_assignments = ', '.join([f"{var}:
{self.storage.evaluateExpression(value)}" for var, value in
sorted_assignments])
                print(f"\nVariables in the range ({min_range}-{max_range})
sorted in {order_text} order: {formatted_assignments}")
```

```
                break  # Exit the loop if input is valid and processed

            except ValueError as ve:
                print(f"Error: {ve}")

    def sortVariables(self, variables, sort_order):
        assignments = [(variable, self.storage.getStorage()[variable]) for variable in variables]
        sorted_assignments = self.sorter.bubbleSort(assignments, sort_order) # sort using BubbleSort
        return sorted_assignments

#-----------------------------------------------------
#
# Filename  : visualize.py
#
#-----------------------------------------------------
# This file retrives all variables in the storage and plots a barplot

class VariableBarPlot:
    def __init__(self, variable_values):
        self.variable_values = variable_values

    def plot(self):
        max_value = max(max(values) for values in self.variable_values.values())
        scale_factor = 10  # Scale of the chart

        for variable, values in self.variable_values.items():
            print(f'{variable}:')
            for value in values:
                scaled_value = int(round(value * scale_factor / max_value))
                bar = '*' * scaled_value
                print(f'    {value: 5.2f} | {bar}')
            print()
```