School of Computing (SoC)

Diploma in Applied AI and Analytics

AY 2021/2022 Semester 2

ST1507 Data Structures and Algorithms

Assignment Two

Expression Evaluator and Sorter (using Parse Trees)

Class:	DAAA/FT/2B/03
Lecturer:	Ms Hwee Shan Tay
Members:	Ethan Tan (P2012085)
	Reshma (P2011972)

TABLE OF CONTENTS

Description

This report entails the design, implementation, analysis and use of a program intended to evaluate simple arithmetic expressions.

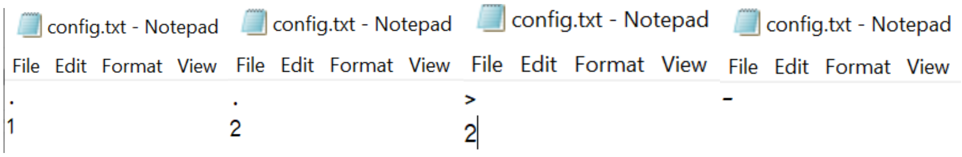
The program supports the following functions:

1. Evaluate and display the parse tree of a fully parenthesised valid arithmetic expression
2. Evaluate and display the parse tree of any valid arithmetic expression
3. Read, evaluate and sort arithmetic expressions from a specified text file and save the report to a specified output file
4. Fully parenthesize a valid arithmetic expression
5. Change the display mode for options 1 and 2
6. Register new operators to be used
7. Exit the program

User Guidelines

Configuration:

Users can select which separator and operator implementation group they want to use, through a configuration file, *config.txt*.



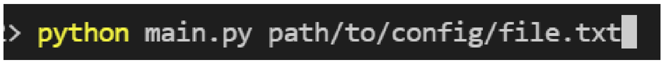
```
.
1
```

```
.
2
```

```
>
2
```

```
-
```

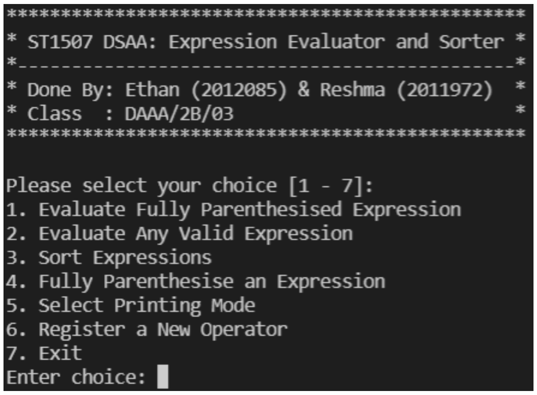
An alternative path can be specified through the command line.



```
> python main.py path/to/config/file.txt
```

The user can choose any valid single character for the separator (including space character) and can only choose from option 1 (Operator Group One) or 2 (Operator Group Two) for the operator implementation group.

The default options are '.' and 1 for the separator and operator implementation group respectively. These defaults apply if the configuration file cannot be found or if any option is omitted. The application loads the configuration(s) specified, and it cannot be changed afterwards.



```
**************************************************
* ST1507 DSAA: Expression Evaluator and Sorter *
*------------------------------------------------*
* Done By: Ethan (2012085) & Reshma (2011972)  *
* Class  : DAAA/2B/03                           *
**************************************************

Please select your choice [1 - 7]:
1. Evaluate Fully Parenthesised Expression
2. Evaluate Any Valid Expression
3. Sort Expressions
4. Fully Parenthesise an Expression
5. Select Printing Mode
6. Register a New Operator
7. Exit
Enter choice:
```

Start up

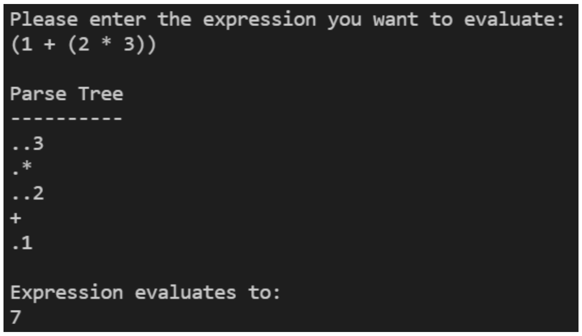
Upon start up, the program greets the user with a menu, displaying all the functions it supports.

The program will ignore invalid selections.

Option 1: Evaluate a Fully Parenthesised Expression

This option accepts a fully parenthesized valid arithmetic expression. It displays the parse tree used to evaluate the expression, along with the result the expression evaluates to.

The separator and base operator implementation group are the ones specified during configuration.



```
Please enter the expression you want to evaluate:
(1 + (2 * 3))

Parse Tree
----------
..3
.*
..2
+
.1

Expression evaluates to:
7
```

The orientation and traversal order used for printing can be configured in Option 5.

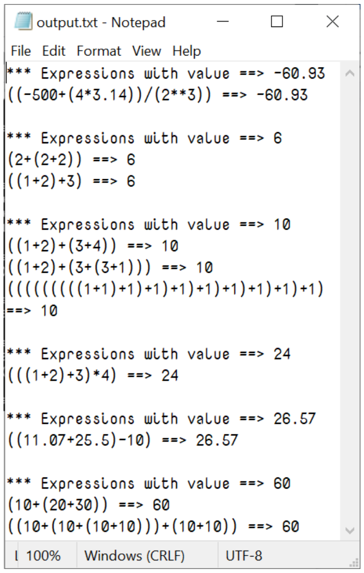
Option 3: Evaluate and Sort Expressions

In this option, the user will be prompted to enter an input and output file. Subsequently, the program will read the input file, evaluate and sort the expressions in the input file.

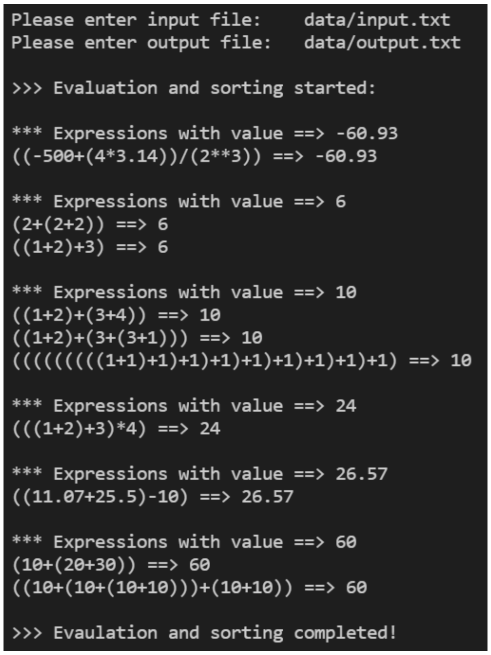
The sorting sorts the value of the expressions in ascending order, followed by their lengths in ascending order. If expressions have the same value and length, they will be sorted by the number of brackets in descending order.

The output will be displayed on the screen as well as written to the output file specified by the user.

The report will not be saved if no output file is specified.



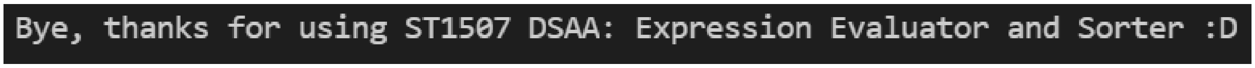
```
*** Expressions with value ==> -60.93
((-500+(4*3.14))/(2**3)) ==> -60.93

*** Expressions with value ==> 6
(2+(2+2)) ==> 6
((1+2)+3) ==> 6

*** Expressions with value ==> 10
((1+2)+(3+4)) ==> 10
((1+2)+(3+(3+1))) ==> 10
(((((((((1+1)+1)+1)+1)+1)+1)+1)+1)+1)
==> 10

*** Expressions with value ==> 24
(((1+2)+3)*4) ==> 24

*** Expressions with value ==> 26.57
((11.07+25.5)-10) ==> 26.57

*** Expressions with value ==> 60
(10+(20+30)) ==> 60
((10+(10+(10+10)))+(10+10)) ==> 60
```



```
Please enter input file:     data/input.txt
Please enter output file:    data/output.txt

>>> Evaluation and sorting started:

*** Expressions with value ==> -60.93
((-500+(4*3.14))/(2**3)) ==> -60.93

*** Expressions with value ==> 6
(2+(2+2)) ==> 6
((1+2)+3) ==> 6

*** Expressions with value ==> 10
((1+2)+(3+4)) ==> 10
((1+2)+(3+(3+1))) ==> 10
(((((((((1+1)+1)+1)+1)+1)+1)+1)+1)+1) ==> 10

*** Expressions with value ==> 24
(((1+2)+3)*4) ==> 24

*** Expressions with value ==> 26.57
((11.07+25.5)-10) ==> 26.57

*** Expressions with value ==> 60
(10+(20+30)) ==> 60
((10+(10+(10+10)))+(10+10)) ==> 60

>>> Evaulation and sorting completed!
```

Option 7: Exit / Quit

This option terminates the program and exits, while bidding a friendly farewell to the user :D



```
Bye, thanks for using ST1507 DSAA: Expression Evaluator and Sorter :D
```

Implementation: Overview of Classes

Node (inherits ABC)

Description: Represents a single node data structure, with pointers to its parent and children
Purpose: Abstract base class for nodes
Implements: display, update_widths, __str__ and __repr__ methods, used for vertical printing
Challenges: Pointers are meant to be easily manipulated by Tree and ParseTree classes
Properties are indicated to be for internal use through single underscores

TempNode (inherits Node)

Description: Represents empty space useful for padding
Purpose: Provides padding for vertical printing
Implements: display (polymorphism) method, returns whitespace characters

MathNode (inherits Node)

Description: Represents a node in a parse tree, either an operand or operator
Purpose: Abstract base class for Operand and Operator subclasses
Implements: __call__ and __str__ (polymorphism) methods, used for evaluation and printing
Asserts: __lt__ and __gt__ (polymorphism) methods, used for parse tree construction

Operand (inherits MathNode)

Description: Represents a node in a parse tree, whose value is an operand
Purpose: Wrapper class for operands
Implements: __lt__, __gt__, get_priority, augment_priority and copy methods,
useful for parse tree construction

Operator (inherits MathNode)

Description: Represents a node in a parse tree, whose value is an operator
Purpose: Wrapper class for operators
Implements: __lt__, __gt__, get_priority, augment_priority and copy methods,
useful for parse tree construction

PrintOrientation (inherits Enum)

Description: Represents the collection of all valid print orientation options
Purpose: Ensures user-selected print orientations are valid
Members: HORIZONTAL, VERTICAL

TreeTraversalOrder (inherits Enum)

Description: Represents the collection of all valid tree traversal options
Purpose: Ensures user-selected tree traversals are valid
Members: IN_ORDER, PRE_ORDER, POST_ORDER

Tokenizer

Description: Represents a tokenizer object useful for tokenizing an expression
Purpose: Extract tokens from a string of characters
Implements: __combine and tokenize methods

Lexer

Description: Represents a lexer object useful for performing lexical analysis
Purpose: Creates a list of token objects from a list of string tokens
Implements: __lex_token and lex methods

Tree

Description: Represents a tree data structure
Purpose: Base class for a tree
Implements: print_tree and change_print_mode methods, with other utility methods, for different ways of printing the tree
Encapsulates: _root and _currentPointer, protected properties
__depth_symbol, __print_orientation & __print_traversal_order, private properties

ParseTree (inherits Tree)

Description: Represents a parse tree data structure
Purpose: Implement useful methods for parsing and evaluating arithmetic expressions
Implements: read, parse, build and __prepare methods for tree construction
__validate_parse_tree and __validate_fully_parenthesised methods for validation
print_tree, evaluate, evaluate_and_sort, reconstruct_expression and register_new_operator methods

Expression

Description: Represents a single arithmetic expression
Purpose: Wrapper class for an expression and its result
Implements: __lt__, __le__, __gt__, __ge__ and comparison_tuple methods for comparisons
get_result, __str__ and __repr__ methods for displaying

InvalidOptionError (inherits Exception)

Description: Represents an error whenever a user selects an invalid option
Purpose: Debugging

InvalidExpressionError (inherits SyntaxError)

Description: Represents an error whenever a user enters an invalid expression
Purpose: Debugging
Default: ‘Invalid Expression’

Implementation: Complexity Analysis

Tokenization

Space Complexity:	O(t) (space required to store tokens is proportional to number of tokens)
Time Complexity:	O(t) [Ideal] (ideally, the regex matching process should be linear)
Legend:	t = total number of tokens

Lexical Analysis

Space Complexity:	O(t) (space required to store objects is proportional to number of tokens)
Time Complexity:	O(t) (iterates over the list of tokens once)
Legend:	t = total number of tokens

Parse Tree Insertion

Space Complexity:	O(n) (extra space required is proportional to the number of nodes added)
Time Complexity:	O(n) [Amortised worst case] (amortised constant time to add each node)
Legend:	n = total number of nodes inserted

Parse Tree Construction

Space Complexity:	O(t) (all subprocesses have linear space complexity)
Time Complexity:	O(t) (all subprocesses have linear time complexity)
Legend:	t = total number of tokens

Parse Tree Evaluation

Space Complexity:	O(1) (evaluated in-place)
Time Complexity:	O(n) (node values are updated recursively)
Legend:	n = total number of nodes

Parse Tree Printing (DFS)

Space Complexity:	O(1) (no extra space required)
Time Complexity:	O(n) (visits every node once)
Legend:	n = total number of nodes

Parse Tree Printing (BFS)

Space Complexity:	O(m) (stores child nodes in a list)
Time Complexity:	O(n) (visits every node twice)
Legend:	m = total number of operands in the expression n = total number of nodes

Merge Sort

Space Complexity:	O(n log n) (creates sub lists with each partition)
Time Complexity:	O(n log n) (linear sorting with logarithmic partitions)
Legend:	n = total number of items

Implementation: Design

Merge Sort

Merge sort was used instead of other sorting algorithms like Bubble sort and Insertion sort due to efficiency. Merge sort has a time complexity of O(n log n), which is a vast improvement from O(n^2) (Bubble and Insertion sort). It does have a drawback of increased space complexity as our implementation does not perform sorting in-place.

Expression

The Expression class was introduced to facilitate the sorting of expressions. It acts as a wrapper around a single arithmetic expression and provides automatic predefined comparison methods which are utilised in sorting functions (in this case, merge sort).

Operator

The Operator class provides a simple interface for new operators to be registered, hence making the program extensible.

PrintOrientation / TreeTraversalOrder

Enums were used to contain all the supported options for printing orientation (horizontal/vertical) and tree traversal orders (pre-order/in-order/post-order) as comparing enum members is more robust and scalable as compared to comparing raw strings.

Custom values can also be specified for each member, which were used in displaying the active configuration.

Tree / Node

The edges (implemented as pointers) were delegated to the Node class as it seemed logical.

Parse Tree

The parsing and construction used a pointer (labelled currentPointer) to track the position of the last token object inserted. This is an efficient method of handling insertion and enables parsing non-fully parenthesised expressions as an added bonus.

Subtrees can be easily pruned as well by removing or re-assigning the pointer to the root of that sub-tree.

Implementation: OOP Concepts

Abstraction

Node and MathNode are abstract classes which are not meant to be instantiated from directly. Instead, they enforce a template for subclasses to follow and define what methods must be overridden in them.

Inheritance

The Parse Tree class inherits the Tree class

The Operand and Operator classes inherit the MathNode class, which inherits the generic Node class

Custom error classes inherit the Exception and SyntaxError classes

Methods and properties were grouped together in the highest logical common superclass

Polymorphism

Overloading

__lt__, __gt__methods were overloaded in Expression, Operator and Operand classes

get_priority, augment_priority and copy methods were defined differently in Operator and Operand classes

Overriding

__str__ and __repr__ methods were overridden in Expression, Node and MathNode classes

Encapsulation

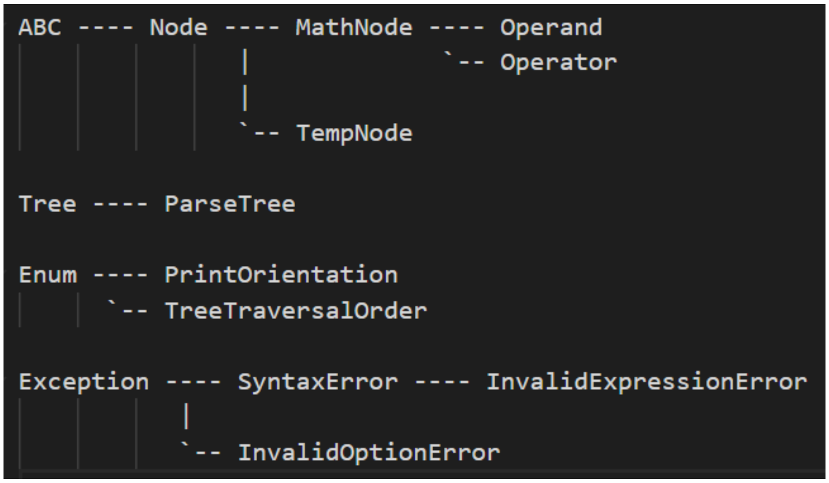
Private properties and methods in Expression, Lexer, Tokenizer, Operator, Tree and ParseTree classes were prefixed with double underscores

Protected (private within inheritance) properties and methods in Tree and ParseTree classes were prefixed with single underscores

Internal properties of Node and MathNode classes were prefixed with single underscores to denote internal use (i.e. use with care). They were not fully privatised to enable ease of manipulation within the Tree and ParseTree classes

Python’s property decorator was used in the encapsulation of the private __print_orientation and __print_traversal_order attributes of the Tree class. Custom validation was performed within the corresponding setter functions

Implementation: Inheritance Scheme



```
ABC ---- Node ---- MathNode ---- Operand
               |             `-- Operator
               |
               `-- TempNode

Tree ---- ParseTree

Enum ---- PrintOrientation
      `-- TreeTraversalOrder

Exception ---- SyntaxError ---- InvalidExpressionError
           |
           `-- InvalidOptionError
```

Implementation: Summary of Data Structures Used

Name	Type
dict	Built-in
list	Built-in
set	Built-in
tuple	Built-in
Node	Custom
Tree	Custom

Challenges faced

Technical

It was slightly difficult enabling the tokenizer to recognize negative numbers. In the end, the solution was to split up the tokenization process into 2 parts: extracting plain tokens and combining the negative prefix with their corresponding operands.

A considerable amount of debugging was required to make the parse tree's insert method work as expected. The implementation makes use of a pointer and recursive node floating to place nodes in their appropriate positions.

Group Work

There were no issues working together. All was well and there were no conflicts.

Key takeaways / Learning achievements

I learnt the significance of Binary Trees in computing. Binary Trees are used for searching and sorting as they provide a means of storing data in a hierarchical way. Through this project, I was also able to comprehend why we made use of a Binary Tree over other types of trees. Initially, I was not able to grasp the concept of the different tree traversal modes that we were taught in class as it didn't seem useful at first. However, incorporating the different tree traversal modes into this project aided me in understanding that the different ways of traversing trees are just as similar to how we access Linear Data Structures like arrays, stacks and lists. In a nutshell, this project has helped to deepen my knowledge of Trees in Data Structure.

Roles & Contributions

Roles

Ethan : Designer
Reshma : Scribe, Tester, Validator

Contributions

Ethan:

- io_utils.py
- exceptions.py
- lexer.py
- math_node.py
- node.py
- operand_.py
- operator_.py
- temp_node.py
- tokenizer.py

Reshma:

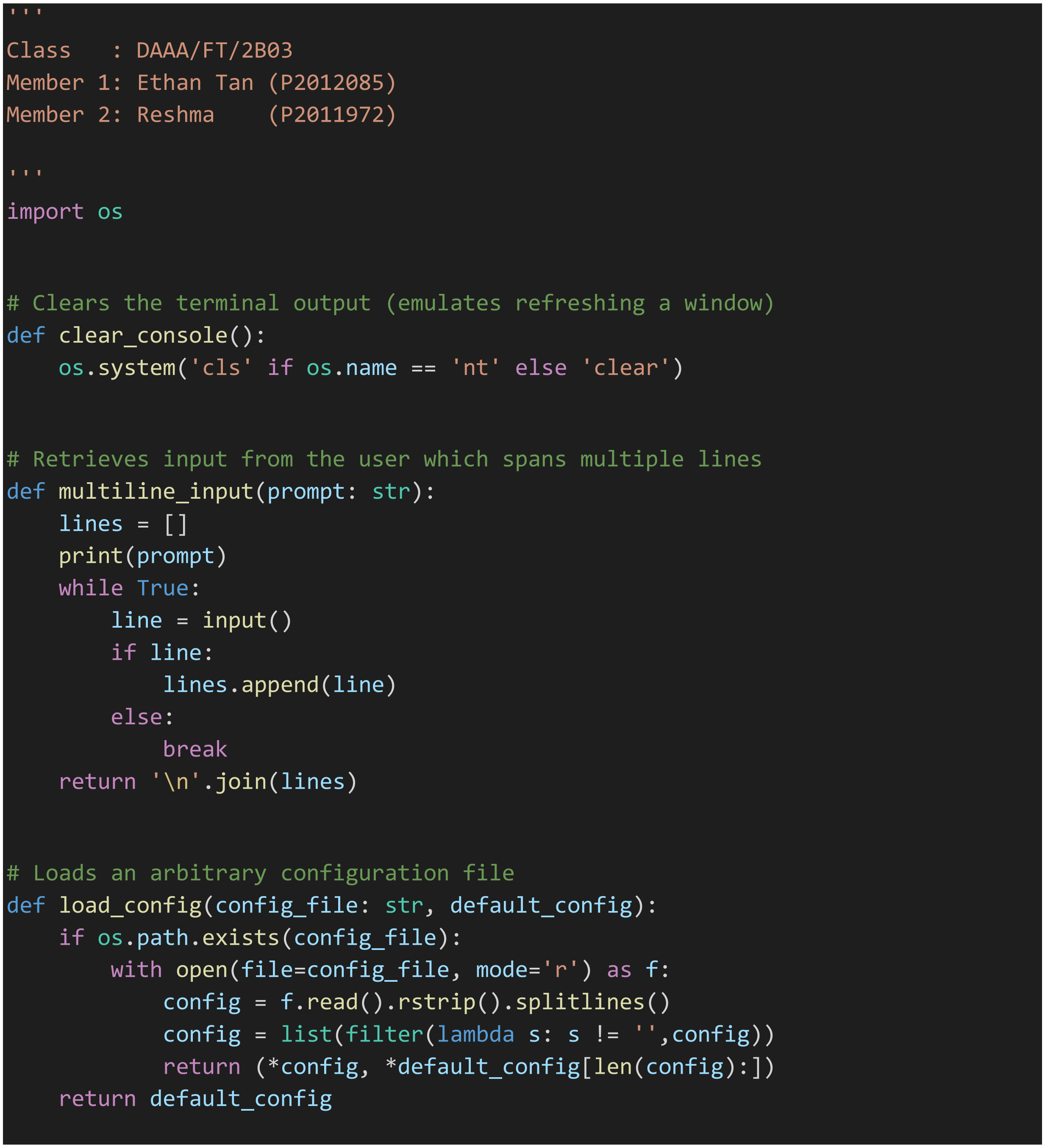
- mergesort.py
- expression.py

Together:

- print_orientation.py
- tree_traversal.py
- tree.py
- parse_tree.py
- main.py
- Report

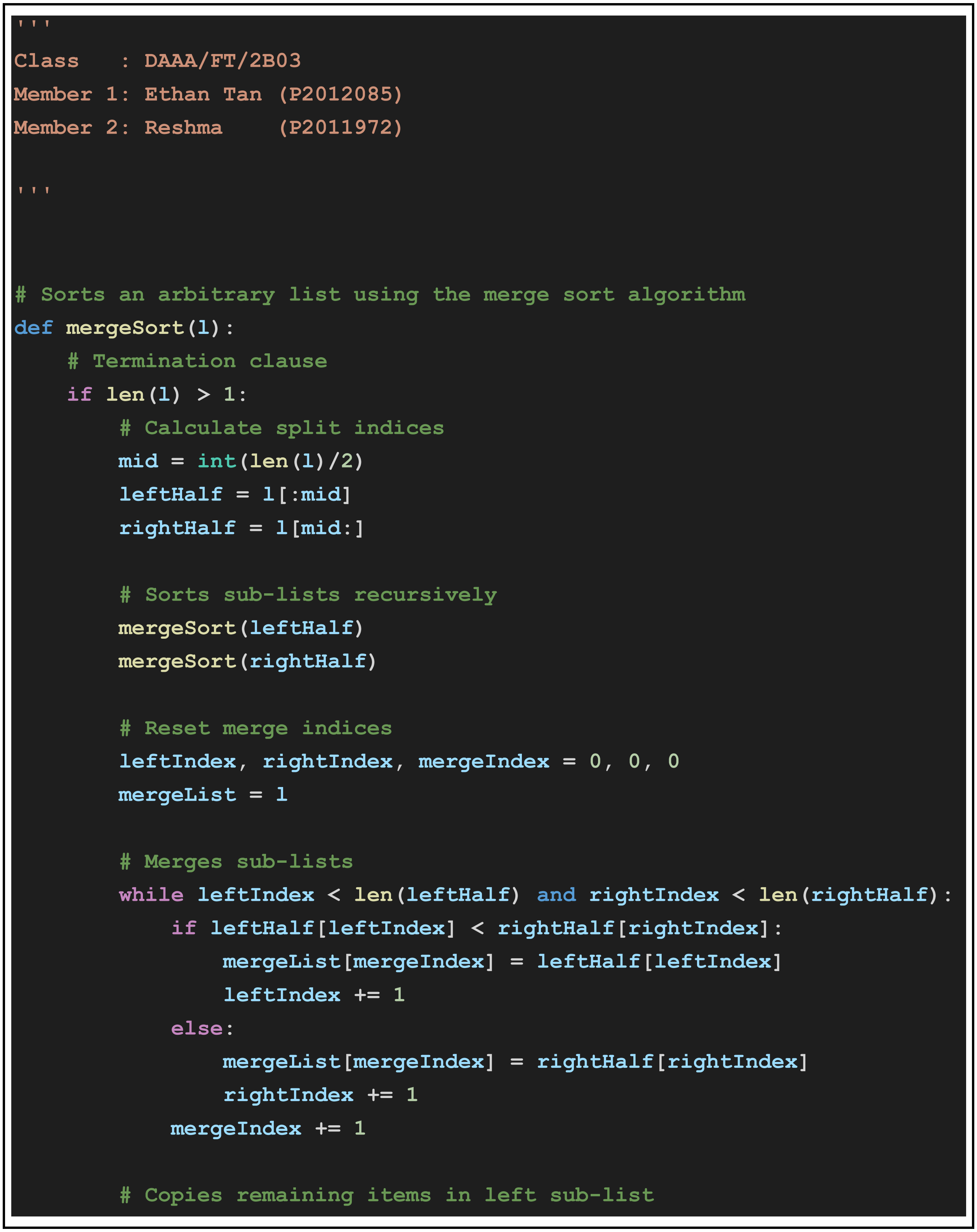
Appendix

src/utils/io_utils.py

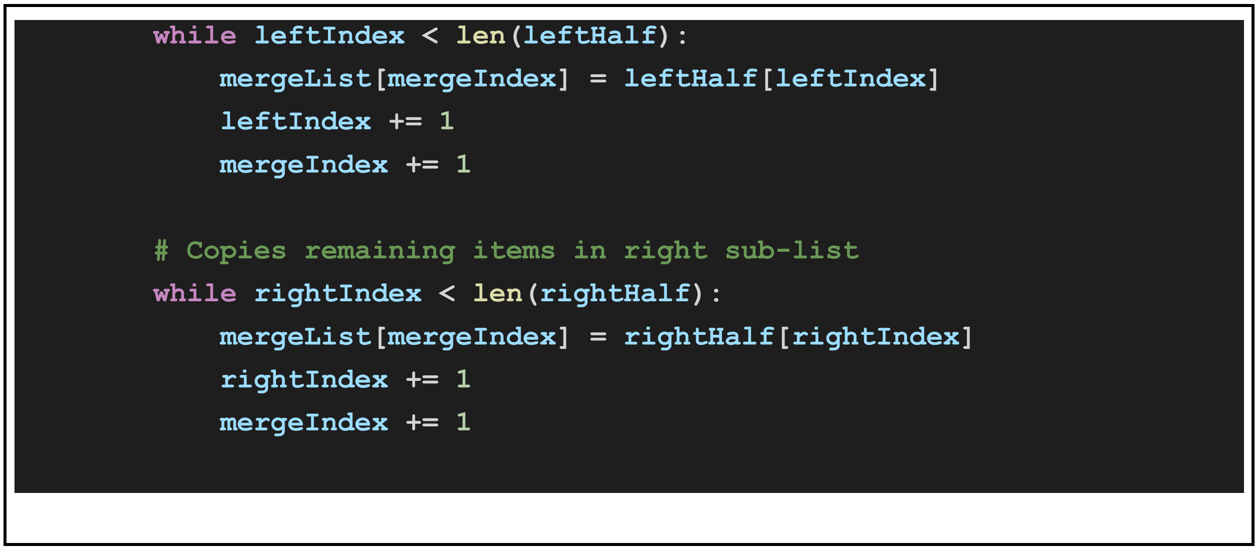


```python
'''
Class    : DAAA/FT/2B03
Member 1: Ethan Tan (P2012085)
Member 2: Reshma    (P2011972)

'''
import os


# Clears the terminal output (emulates refreshing a window)
def clear_console():
    os.system('cls' if os.name == 'nt' else 'clear')


# Retrieves input from the user which spans multiple lines
def multiline_input(prompt: str):
    lines = []
    print(prompt)
    while True:
        line = input()
        if line:
            lines.append(line)
        else:
            break
    return '\n'.join(lines)


# Loads an arbitrary configuration file
def load_config(config_file: str, default_config):
    if os.path.exists(config_file):
        with open(file=config_file, mode='r') as f:
            config = f.read().rstrip().splitlines()
            config = list(filter(lambda s: s != '',config))
            return (*config, *default_config[len(config):])
    return default_config
```

src/utils/mergesort.py

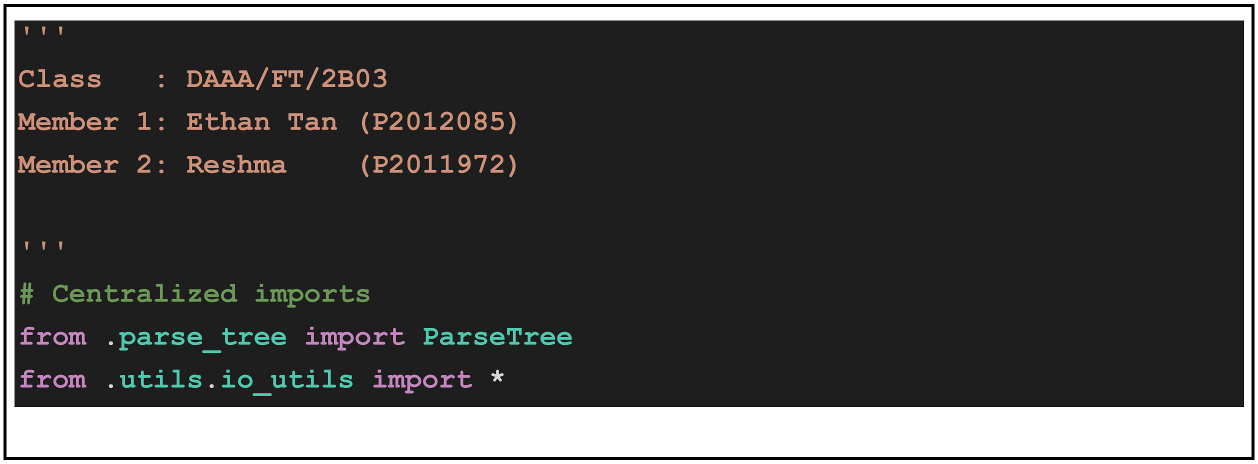


```
'''
Class   : DAAA/FT/2B03
Member 1: Ethan Tan (P2012085)
Member 2: Reshma    (P2011972)

'''


# Sorts an arbitrary list using the merge sort algorithm
def mergeSort(l):
    # Termination clause
    if len(l) > 1:
        # Calculate split indices
        mid = int(len(l)/2)
        leftHalf = l[:mid]
        rightHalf = l[mid:]

        # Sorts sub-lists recursively
        mergeSort(leftHalf)
        mergeSort(rightHalf)

        # Reset merge indices
        leftIndex, rightIndex, mergeIndex = 0, 0, 0
        mergeList = l

        # Merges sub-lists
        while leftIndex < len(leftHalf) and rightIndex < len(rightHalf):
            if leftHalf[leftIndex] < rightHalf[rightIndex]:
                mergeList[mergeIndex] = leftHalf[leftIndex]
                leftIndex += 1
            else:
                mergeList[mergeIndex] = rightHalf[rightIndex]
                rightIndex += 1
            mergeIndex += 1

        # Copies remaining items in left sub-list
```



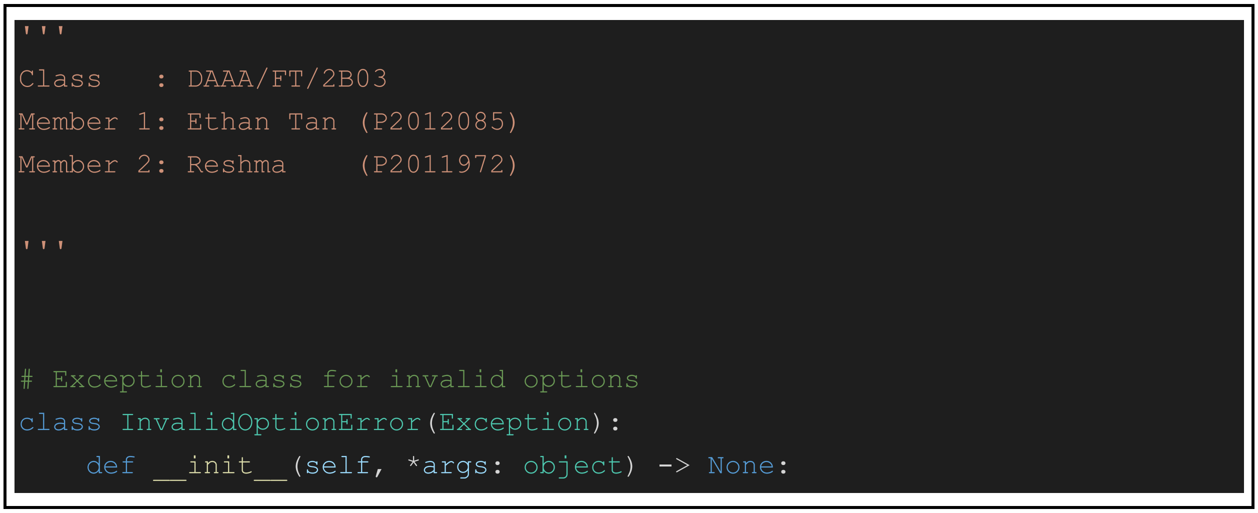
```
        while leftIndex < len(leftHalf):
            mergeList[mergeIndex] = leftHalf[leftIndex]
            leftIndex += 1
            mergeIndex += 1

        # Copies remaining items in right sub-list
        while rightIndex < len(rightHalf):
            mergeList[mergeIndex] = rightHalf[rightIndex]
            rightIndex += 1
            mergeIndex += 1
```

src/__init__.py

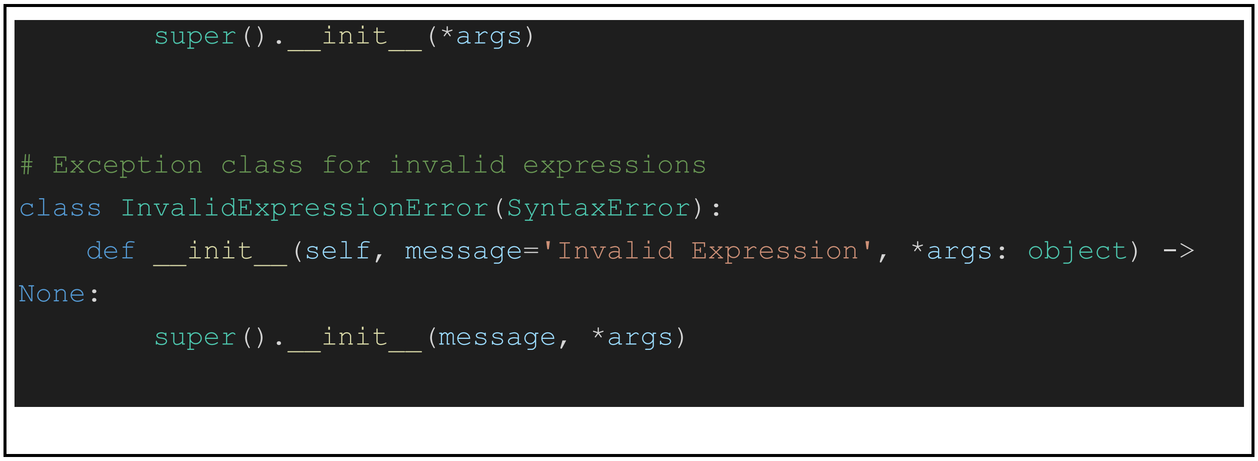


```
'''
Class    : DAAA/FT/2B03
Member 1: Ethan Tan (P2012085)
Member 2: Reshma    (P2011972)

'''
# Centralized imports
from .parse_tree import ParseTree
from .utils.io_utils import *
```

src/exceptions.py

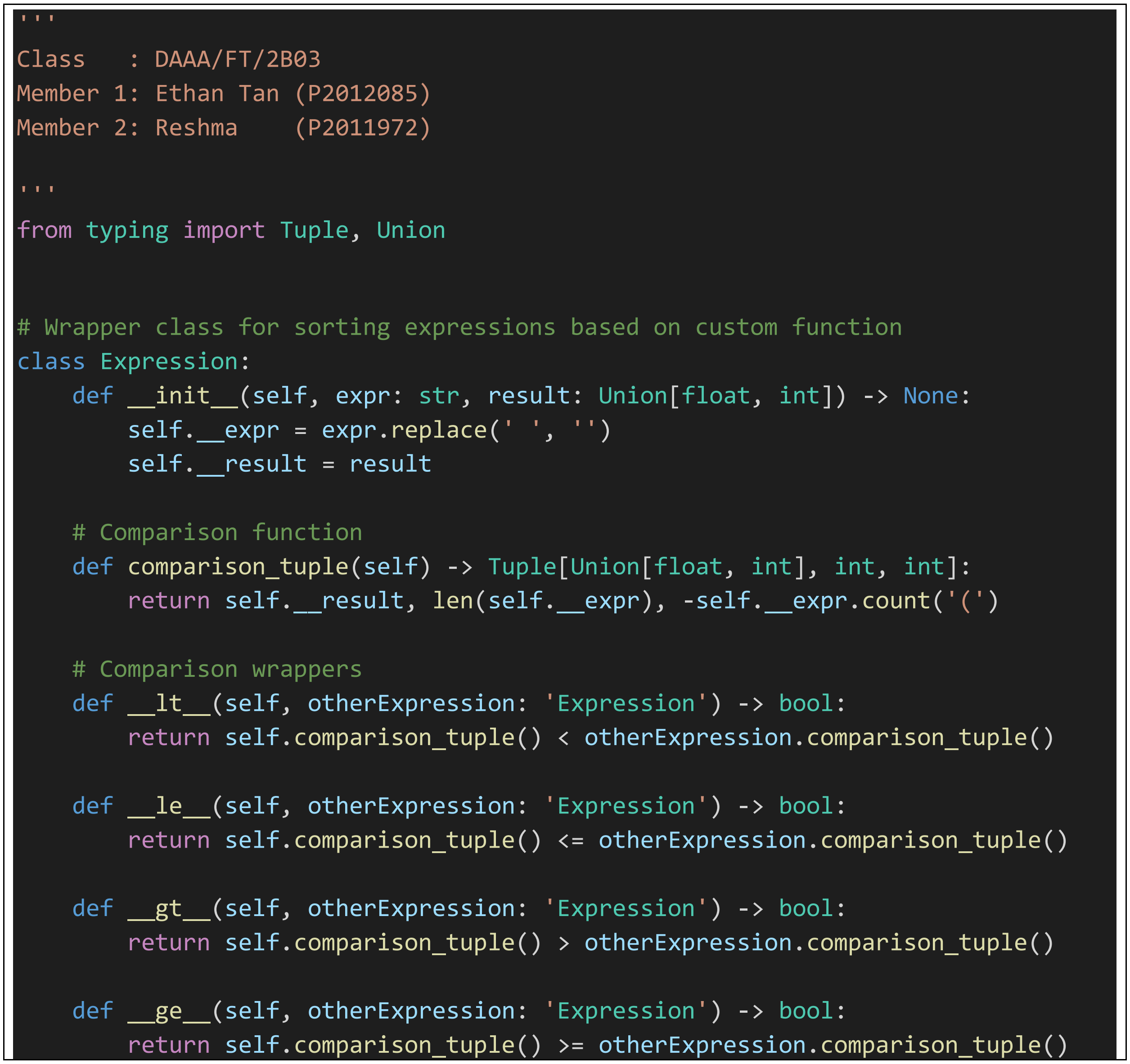


```
'''
Class    : DAAA/FT/2B03
Member 1: Ethan Tan (P2012085)
Member 2: Reshma    (P2011972)

'''



# Exception class for invalid options
class InvalidOptionError(Exception):
    def __init__(self, *args: object) -> None:
```

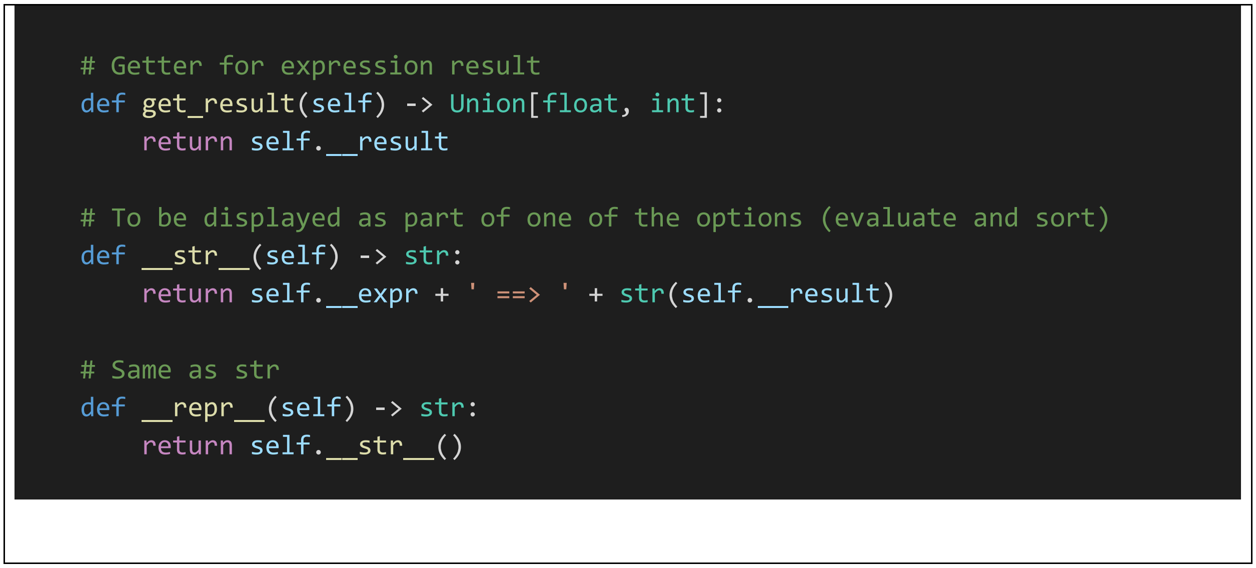


```
        super().__init__(*args)


# Exception class for invalid expressions
class InvalidExpressionError(SyntaxError):
    def __init__(self, message='Invalid Expression', *args: object) ->
None:
        super().__init__(message, *args)
```

src/expression.py

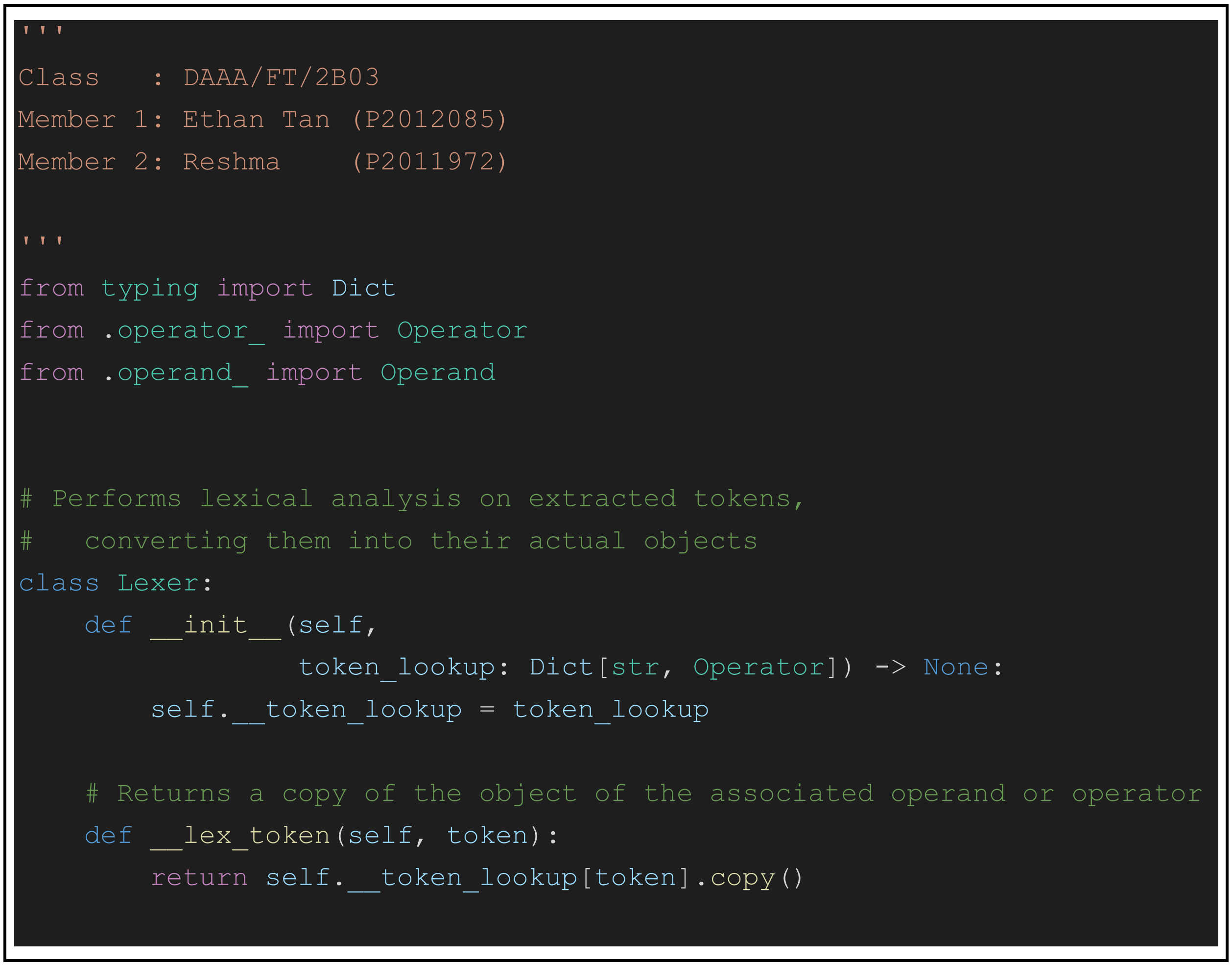


```
'''
Class    : DAAA/FT/2B03
Member 1: Ethan Tan (P2012085)
Member 2: Reshma    (P2011972)

'''
from typing import Tuple, Union


# Wrapper class for sorting expressions based on custom function
class Expression:
    def __init__(self, expr: str, result: Union[float, int]) -> None:
        self.__expr = expr.replace(' ', '')
        self.__result = result

    # Comparison function
    def comparison_tuple(self) -> Tuple[Union[float, int], int, int]:
        return self.__result, len(self.__expr), -self.__expr.count('(')

    # Comparison wrappers
    def __lt__(self, otherExpression: 'Expression') -> bool:
        return self.comparison_tuple() < otherExpression.comparison_tuple()

    def __le__(self, otherExpression: 'Expression') -> bool:
        return self.comparison_tuple() <= otherExpression.comparison_tuple()

    def __gt__(self, otherExpression: 'Expression') -> bool:
        return self.comparison_tuple() > otherExpression.comparison_tuple()

    def __ge__(self, otherExpression: 'Expression') -> bool:
        return self.comparison_tuple() >= otherExpression.comparison_tuple()
```



```
    # Getter for expression result
    def get_result(self) -> Union[float, int]:
        return self.__result

    # To be displayed as part of one of the options (evaluate and sort)
    def __str__(self) -> str:
        return self.__expr + ' ==> ' + str(self.__result)

    # Same as str
    def __repr__(self) -> str:
        return self.__str__()
```

src/lexer.py



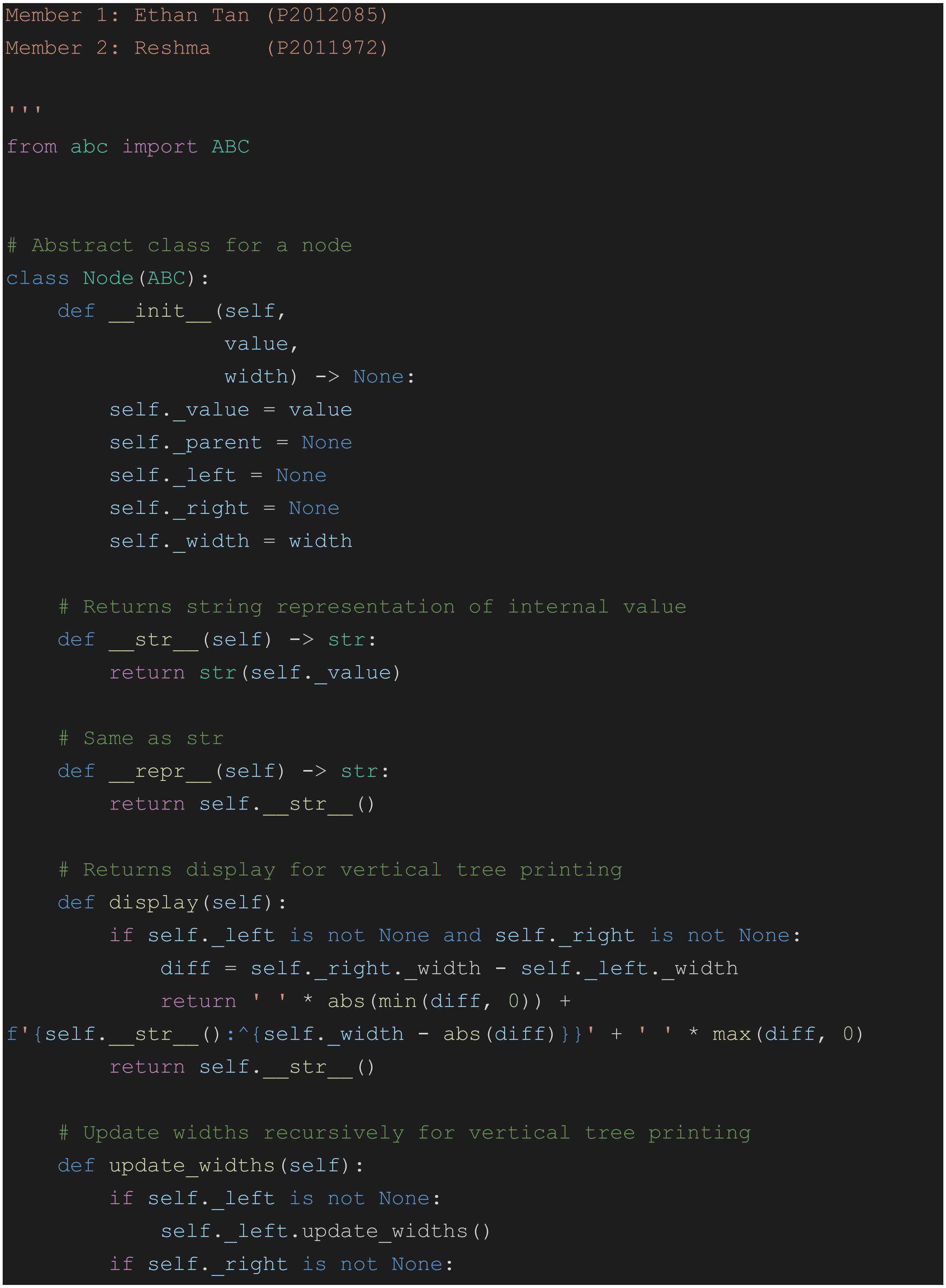
```
'''
Class   : DAAA/FT/2B03
Member 1: Ethan Tan (P2012085)
Member 2: Reshma    (P2011972)

'''
from typing import Dict
from .operator_ import Operator
from .operand_ import Operand


# Performs lexical analysis on extracted tokens,
#   converting them into their actual objects
class Lexer:
    def __init__(self,
                 token_lookup: Dict[str, Operator]) -> None:
        self.__token_lookup = token_lookup

    # Returns a copy of the object of the associated operand or operator
    def __lex_token(self, token):
        return self.__token_lookup[token].copy()
    # Iterates through a given list of tokens, and
    #   conducts lexical analysis
    def lex(self, tokens):
        lexed_token_list = []

        for token in tokens:
            if token in '()':
                lexed_token_list.append(token)
            elif token in self.__token_lookup.keys():
                lexed_token_list.append(self.__lex_token(token=token))
            else:
                lexed_token_list.append(Operand(value=float(token) if
'.' in token else int(token)))

        return lexed_token_list
```

src/math_node.py

```
'''
Class   : DAAA/FT/2B03
Member 1: Ethan Tan (P2012085)
Member 2: Reshma    (P2011972)

'''
from typing import Any
from abc import abstractmethod
from .node import Node


# Abstract superclass for operands and operators
class MathNode(Node):
    def __init__(self, value, symbol, func) -> None:
        super().__init__(value=value, width=len(str(value)))
        self._symbol = symbol
        self._func = func

    # Calls sub-trees recursively before updating its value
    def __call__(self) -> Any:
        if self._left is not None and self._right is not None:
            self._left()
            self._right()
            self._value = self._func(self._left._value,
self._right._value)

    # Returns symbol
    def __str__(self) -> str:
        return str(self._symbol)

    # Abstract comparison methods
    @abstractmethod
    def __lt__(self, otherNode) -> bool:
        pass

    @abstractmethod
    def __gt__(self, otherNode) -> bool:
        pass

    # Abstract methods
    @abstractmethod
    def get_priority(self) -> int:
        pass

    @abstractmethod
    def augment_priority(self) -> None:
        pass

    @abstractmethod
    def copy(self):
        pass
```

src/node.py

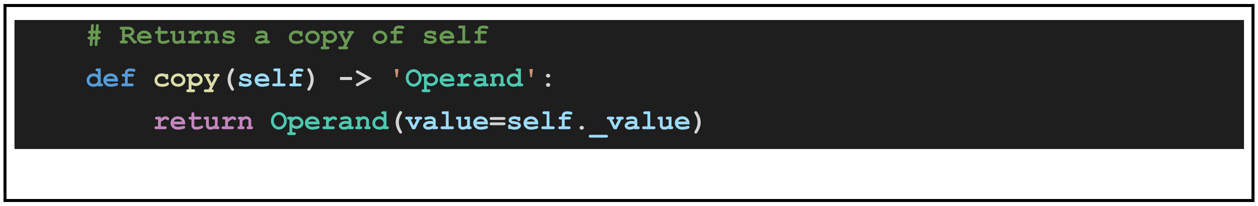
```
'''
Class    : DAAA/FT/2B03
```



```
Member 1: Ethan Tan (P2012085)
Member 2: Reshma    (P2011972)

'''
from abc import ABC


# Abstract class for a node
class Node(ABC):
    def __init__(self,
                 value,
                 width) -> None:
        self._value = value
        self._parent = None
        self._left = None
        self._right = None
        self._width = width

    # Returns string representation of internal value
    def __str__(self) -> str:
        return str(self._value)

    # Same as str
    def __repr__(self) -> str:
        return self.__str__()

    # Returns display for vertical tree printing
    def display(self):
        if self._left is not None and self._right is not None:
            diff = self._right._width - self._left._width
            return ' ' * abs(min(diff, 0)) +
f'{self.__str__():^{self._width - abs(diff)}}' + ' ' * max(diff, 0)
        return self.__str__()

    # Update widths recursively for vertical tree printing
    def update_widths(self):
        if self._left is not None:
            self._left.update_widths()
        if self._right is not None:
            self._right.update_widths()
        if self._left is not None and self._right is not None:
            self._width = self._left._width + 1 + self._right._width
```

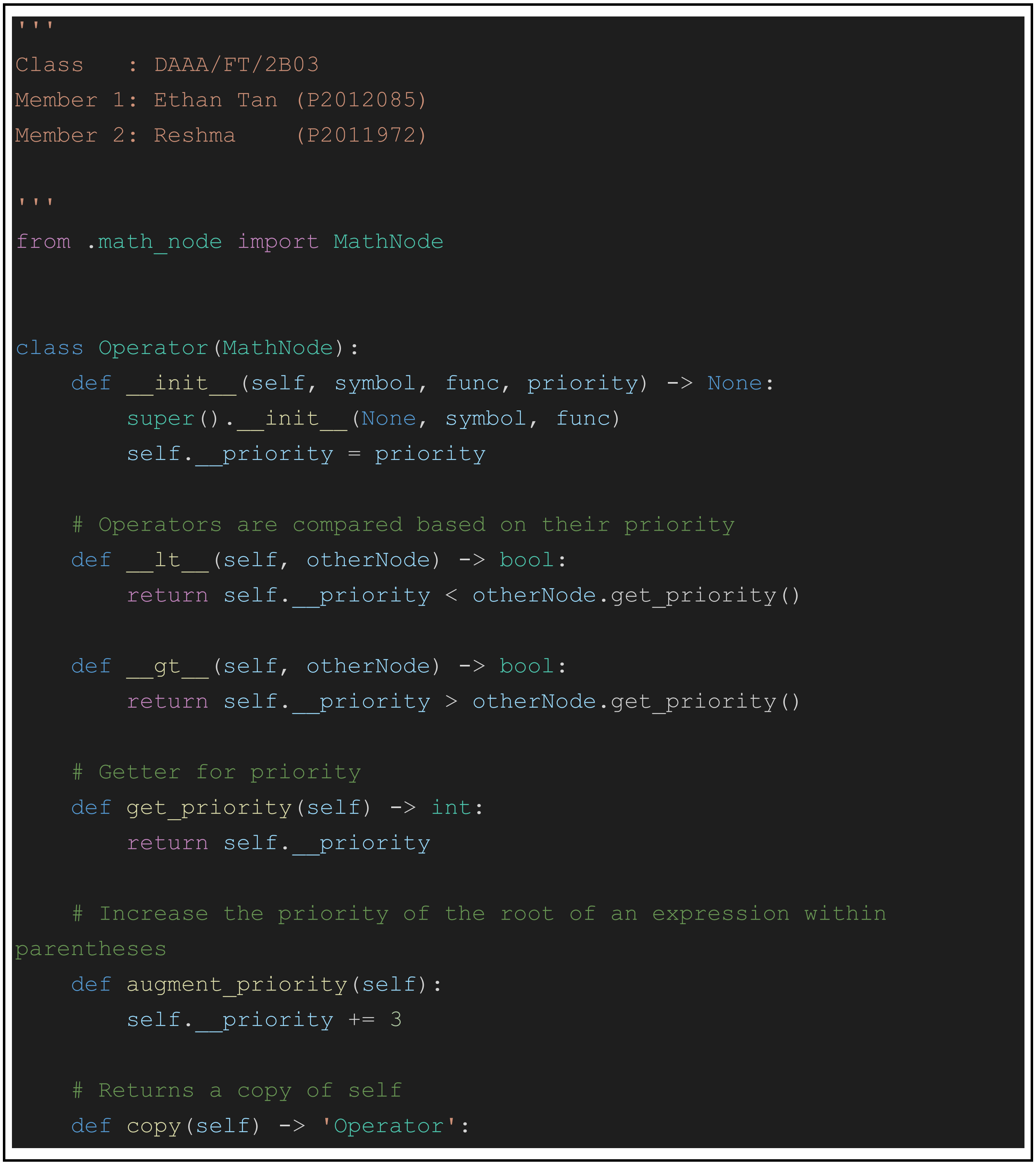
src/operand_.py

```
'''
Class   : DAAA/FT/2B03
Member 1: Ethan Tan (P2012085)
Member 2: Reshma    (P2011972)

'''
from .math_node import MathNode
from .exceptions import InvalidExpressionError


# Class for operands (numbers)
class Operand(MathNode):
    def __init__(self, value) -> None:
        super().__init__(value, value, lambda _, __: self._value)

    # Operands have the highest priority
    def __lt__(self, _) -> bool:
        return False

    def __gt__(self, _) -> bool:
        return True

    # Method should not be called if expression is valid
    def get_priority(self):
        raise InvalidExpressionError()

    # Increase the priority of the root of an expression within
parentheses
    def augment_priority(self):
        pass
```



```
    # Returns a copy of self
    def copy(self) -> 'Operand':
        return Operand(value=self._value)
```

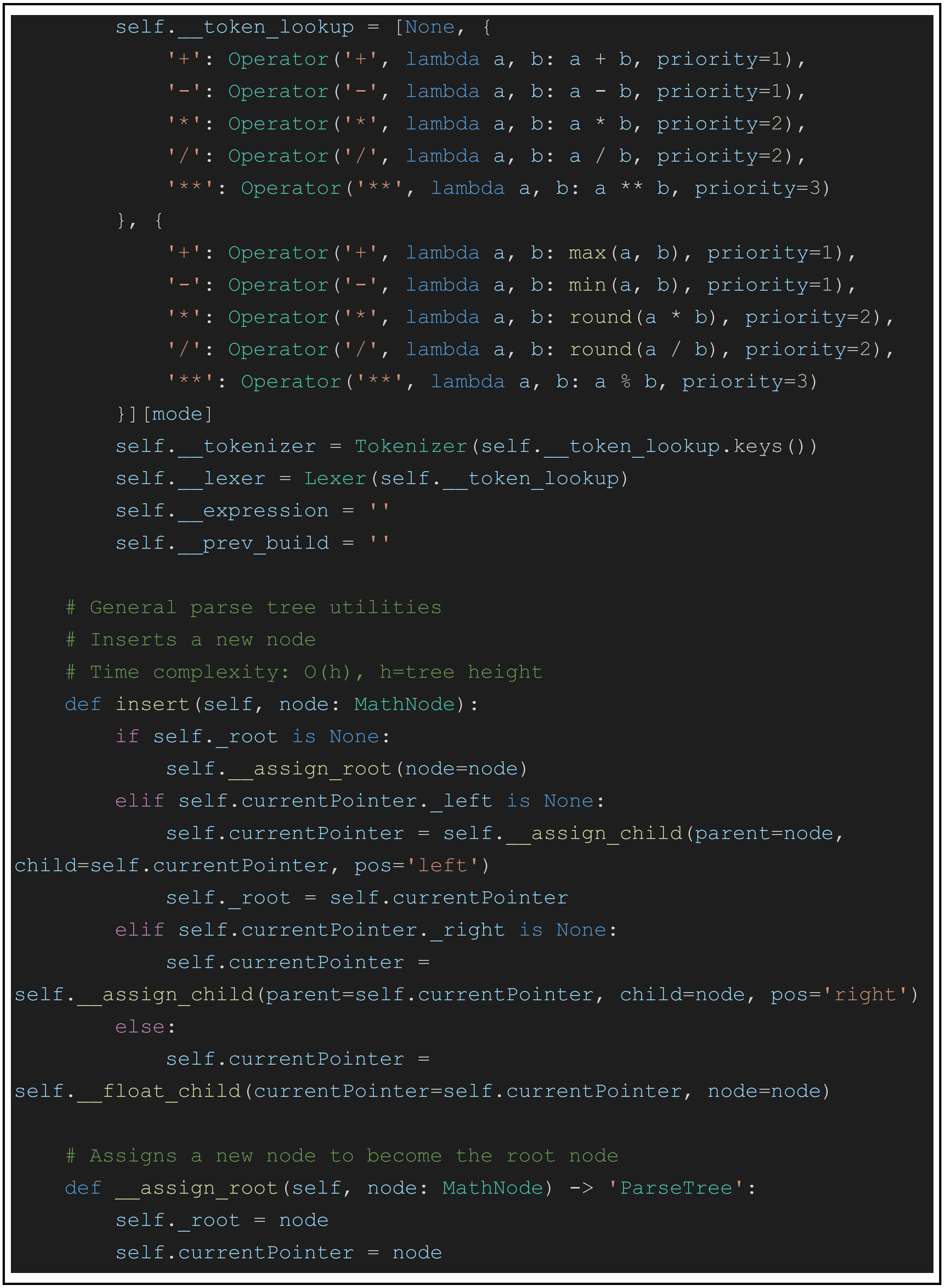
src/operator_.py



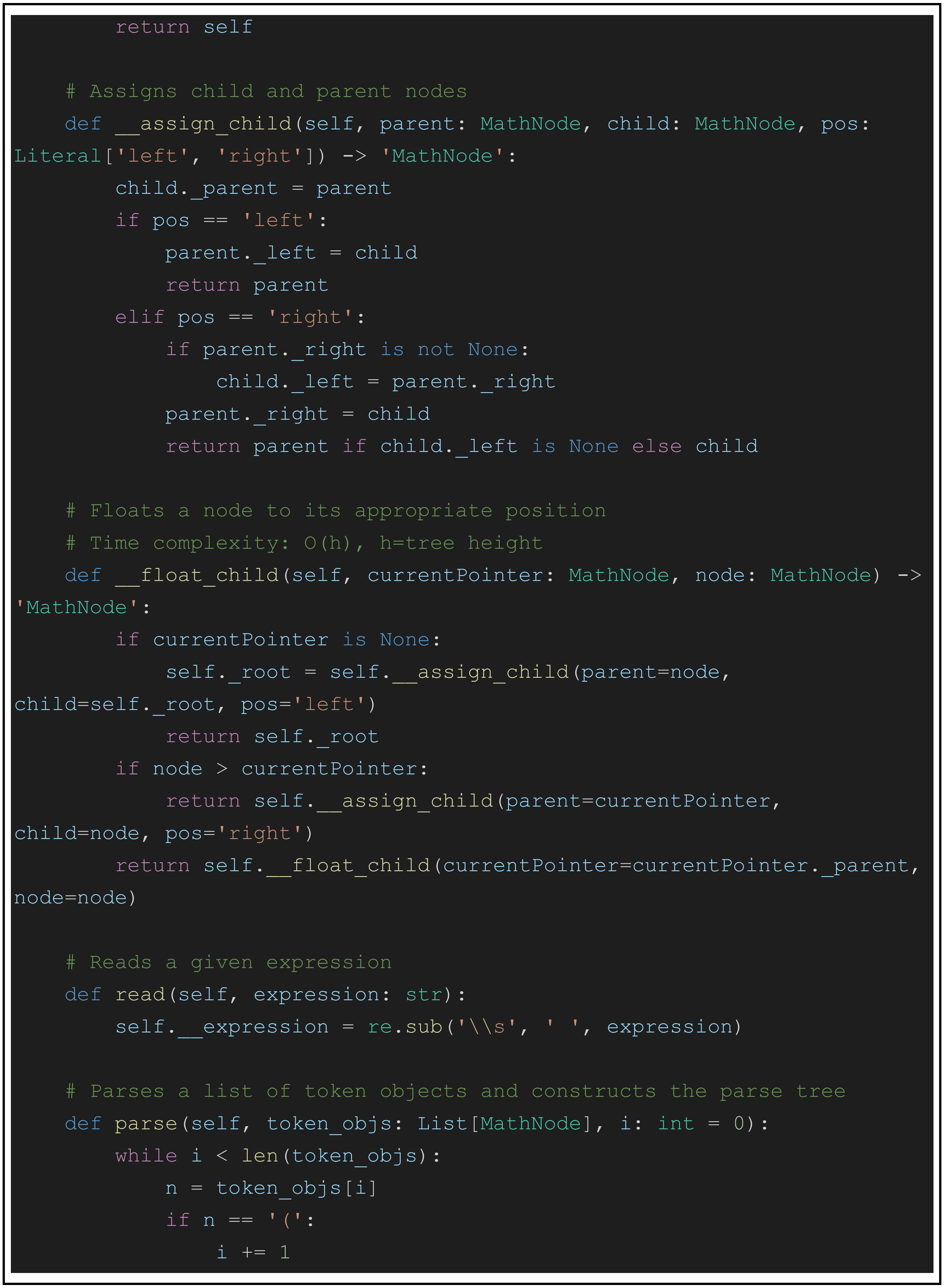
```
'''
Class    : DAAA/FT/2B03
Member 1: Ethan Tan (P2012085)
Member 2: Reshma    (P2011972)

'''
from .math_node import MathNode


class Operator(MathNode):
    def __init__(self, symbol, func, priority) -> None:
        super().__init__(None, symbol, func)
        self.__priority = priority

    # Operators are compared based on their priority
    def __lt__(self, otherNode) -> bool:
        return self.__priority < otherNode.get_priority()

    def __gt__(self, otherNode) -> bool:
        return self.__priority > otherNode.get_priority()

    # Getter for priority
    def get_priority(self) -> int:
        return self.__priority

    # Increase the priority of the root of an expression within
parentheses
    def augment_priority(self):
        self.__priority += 3

    # Returns a copy of self
    def copy(self) -> 'Operator':
        return Operator(symbol=self._symbol, func=self._func,
priority=self.__priority)
```

src/parse_tree.py

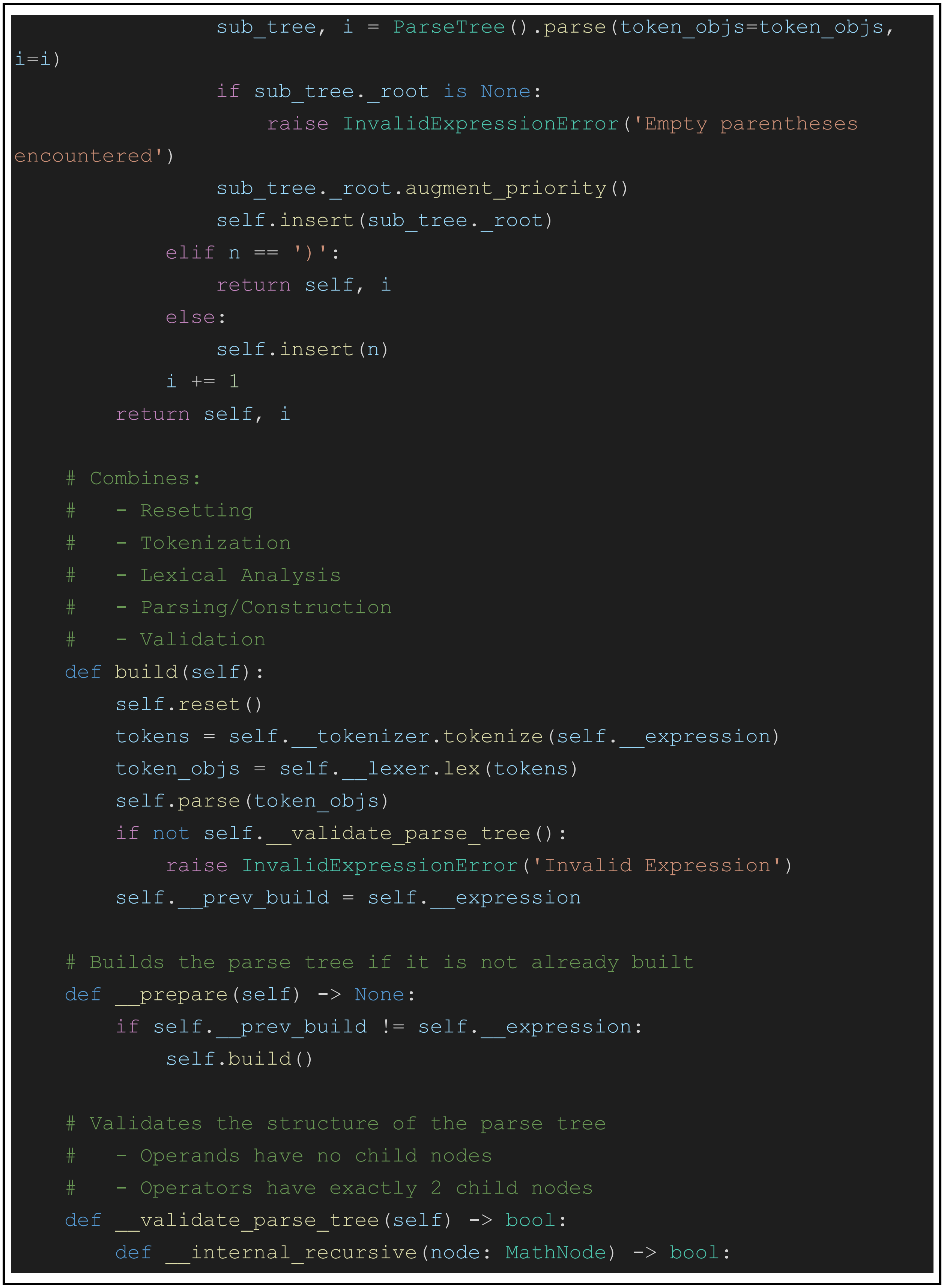
```
# type: ignore
'''
Class   : DAAA/FT/2B03
Member 1: Ethan Tan (P2012085)
Member 2: Reshma    (P2011972)

'''
from typing import List, Literal
import re

from .exceptions import InvalidExpressionError, InvalidOptionError
from .math_node import MathNode
from .operator_ import Operator
from .operand_ import Operand
from .tokenizer import Tokenizer
from .lexer import Lexer
from .tree import Tree
from .expression import Expression
from .utils.mergesort import mergeSort

# For registering custom operators
import math
from math import *


class ParseTree(Tree):
    def __init__(self,
                 depth_symbol: str = '.',
                 mode: int = 1,
                 precision: int = 5) -> None:
        super().__init__(depth_symbol=depth_symbol)
        self.__precision = precision
```



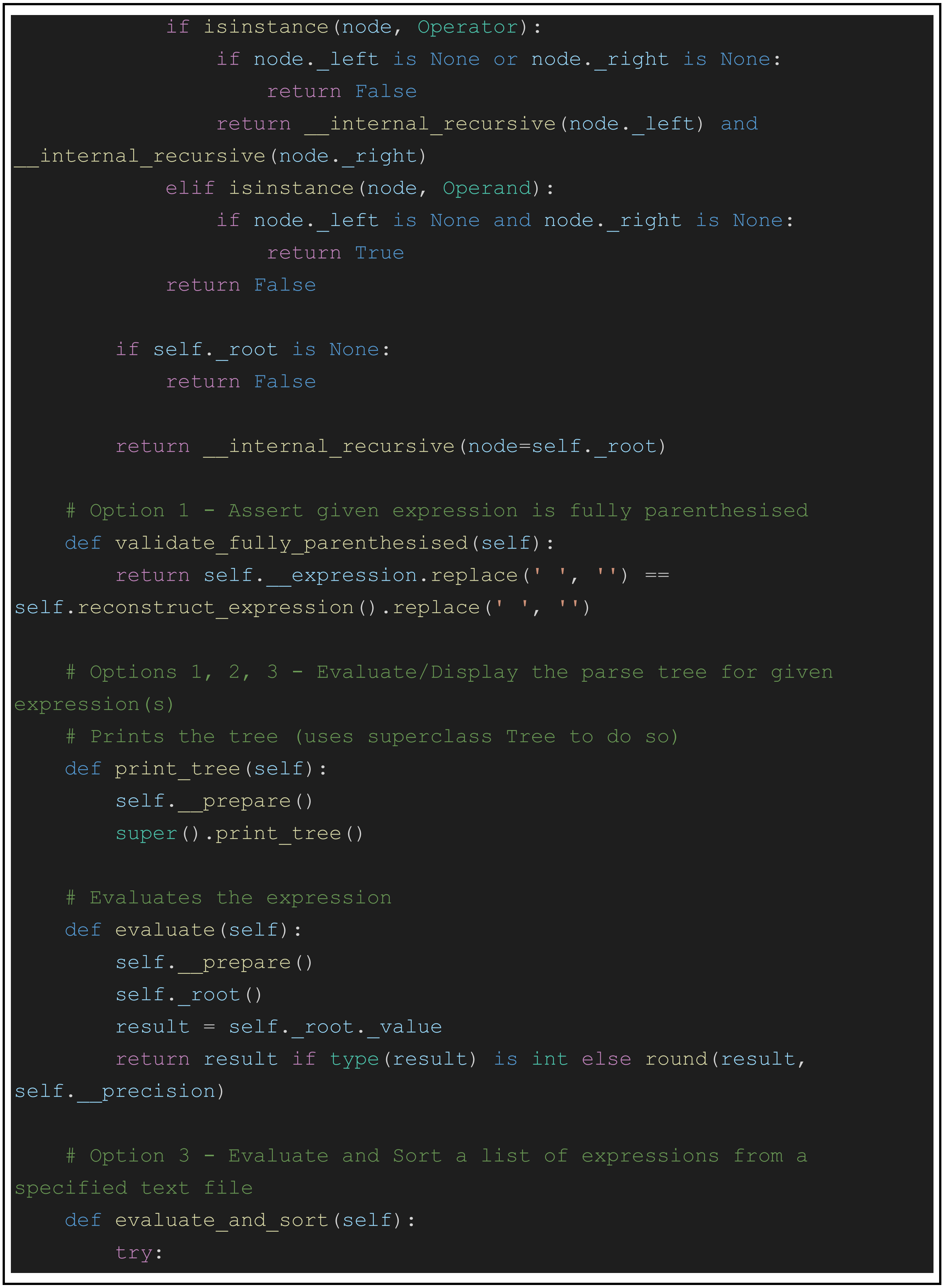
```
        self.__token_lookup = [None, {
            '+': Operator('+', lambda a, b: a + b, priority=1),
            '-': Operator('-', lambda a, b: a - b, priority=1),
            '*': Operator('*', lambda a, b: a * b, priority=2),
            '/': Operator('/', lambda a, b: a / b, priority=2),
            '**': Operator('**', lambda a, b: a ** b, priority=3)
        }, {
            '+': Operator('+', lambda a, b: max(a, b), priority=1),
            '-': Operator('-', lambda a, b: min(a, b), priority=1),
            '*': Operator('*', lambda a, b: round(a * b), priority=2),
            '/': Operator('/', lambda a, b: round(a / b), priority=2),
            '**': Operator('**', lambda a, b: a % b, priority=3)
        }][mode]
        self.__tokenizer = Tokenizer(self.__token_lookup.keys())
        self.__lexer = Lexer(self.__token_lookup)
        self.__expression = ''
        self.__prev_build = ''

    # General parse tree utilities
    # Inserts a new node
    # Time complexity: O(h), h=tree height
    def insert(self, node: MathNode):
        if self._root is None:
            self.__assign_root(node=node)
        elif self.currentPointer._left is None:
            self.currentPointer = self.__assign_child(parent=node, 
child=self.currentPointer, pos='left')
            self._root = self.currentPointer
        elif self.currentPointer._right is None:
            self.currentPointer = 
self.__assign_child(parent=self.currentPointer, child=node, pos='right')
        else:
            self.currentPointer = 
self.__float_child(currentPointer=self.currentPointer, node=node)

    # Assigns a new node to become the root node
    def __assign_root(self, node: MathNode) -> 'ParseTree':
        self._root = node
        self.currentPointer = node
```



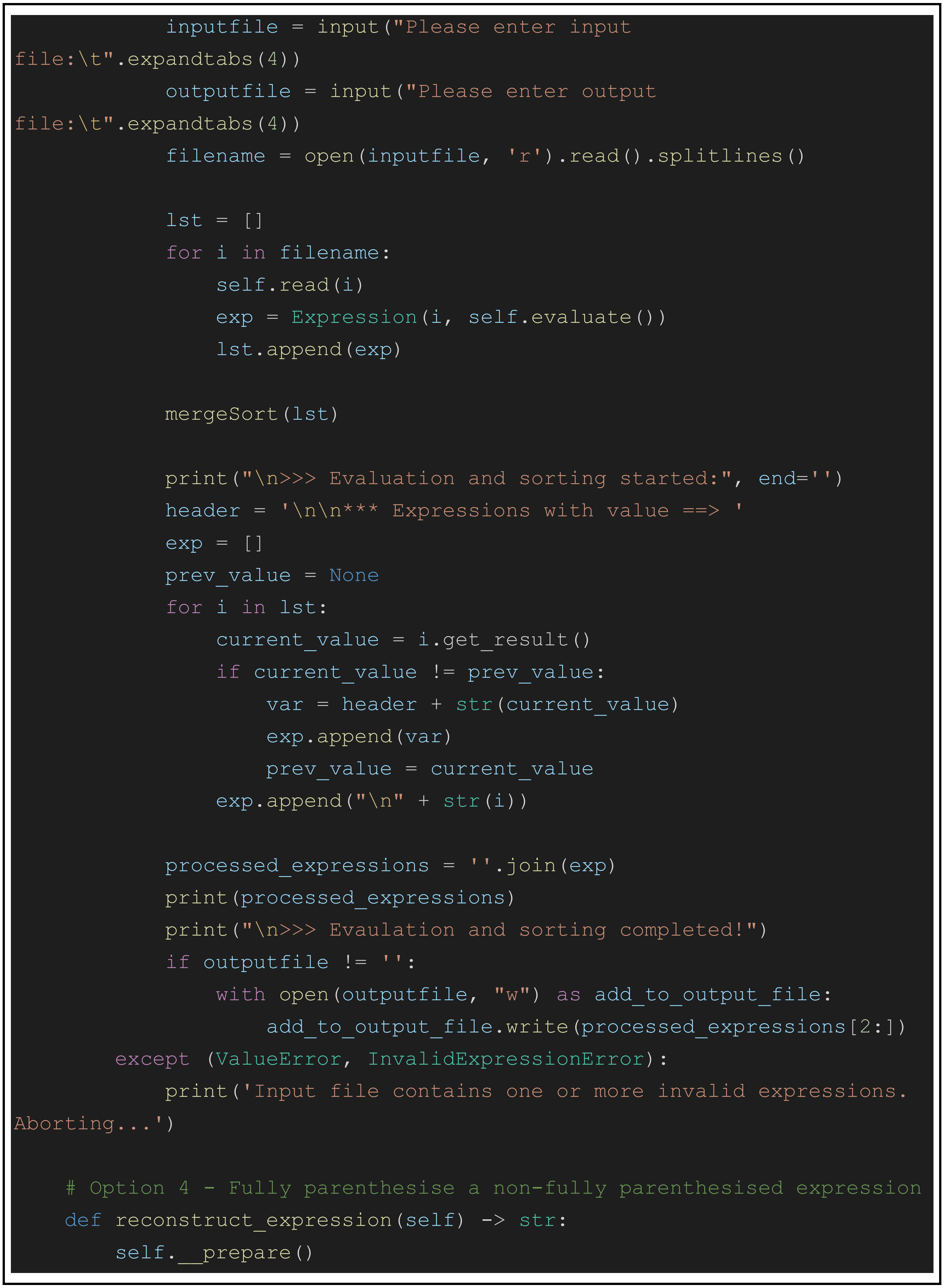
```
        return self

    # Assigns child and parent nodes
    def __assign_child(self, parent: MathNode, child: MathNode, pos:
Literal['left', 'right']) -> 'MathNode':
        child._parent = parent
        if pos == 'left':
            parent._left = child
            return parent
        elif pos == 'right':
            if parent._right is not None:
                child._left = parent._right
            parent._right = child
            return parent if child._left is None else child

    # Floats a node to its appropriate position
    # Time complexity: O(h), h=tree height
    def __float_child(self, currentPointer: MathNode, node: MathNode) ->
'MathNode':
        if currentPointer is None:
            self._root = self.__assign_child(parent=node,
child=self._root, pos='left')
            return self._root
        if node > currentPointer:
            return self.__assign_child(parent=currentPointer,
child=node, pos='right')
        return self.__float_child(currentPointer=currentPointer._parent,
node=node)

    # Reads a given expression
    def read(self, expression: str):
        self.__expression = re.sub('\\s', ' ', expression)

    # Parses a list of token objects and constructs the parse tree
    def parse(self, token_objs: List[MathNode], i: int = 0):
        while i < len(token_objs):
            n = token_objs[i]
            if n == '(':
                i += 1
```



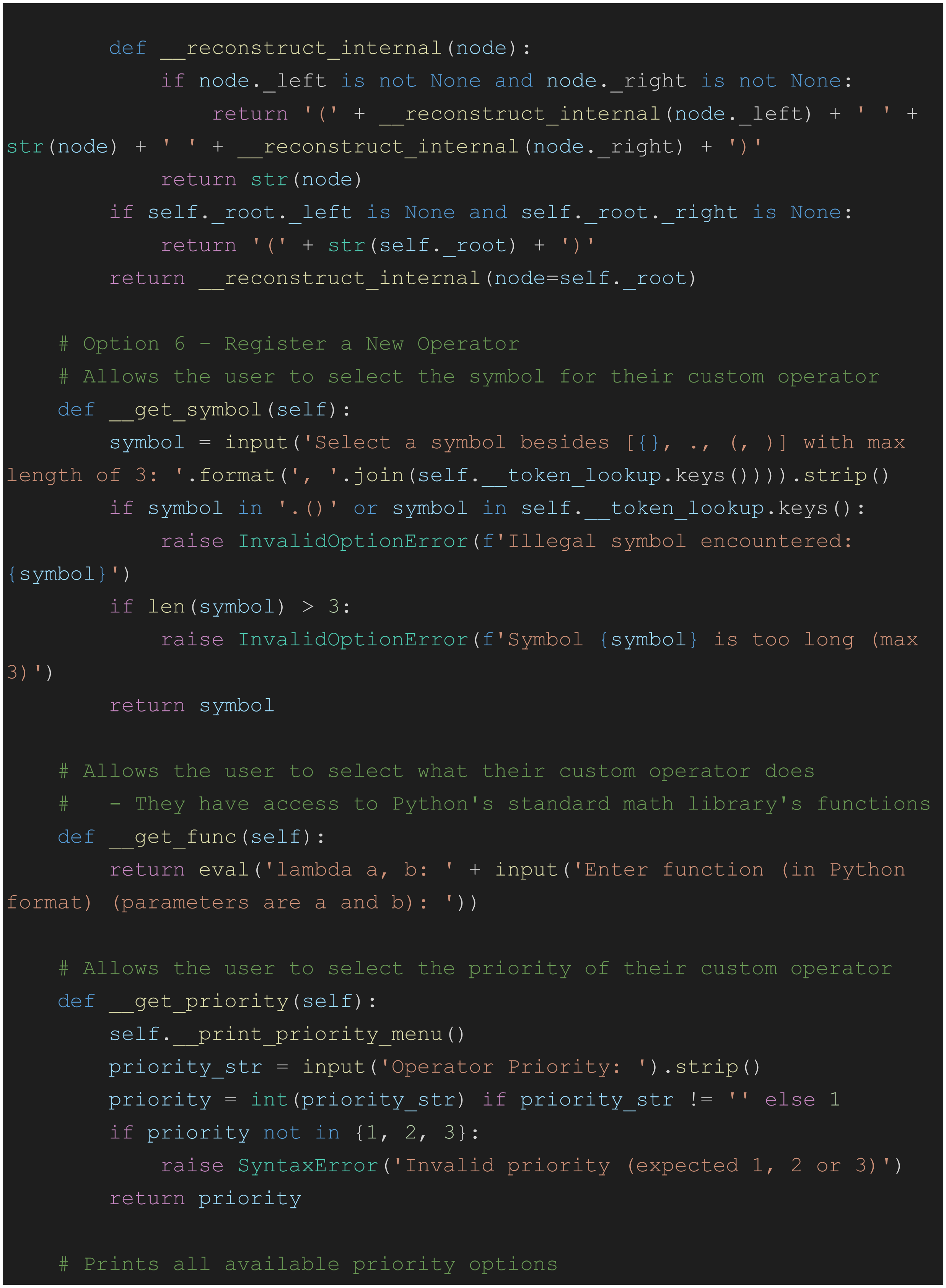
```
                sub_tree, i = ParseTree().parse(token_objs=token_objs,
i=i)
                if sub_tree._root is None:
                    raise InvalidExpressionError('Empty parentheses
encountered')
                sub_tree._root.augment_priority()
                self.insert(sub_tree._root)
            elif n == ')':
                return self, i
            else:
                self.insert(n)
            i += 1
        return self, i

    # Combines:
    #   - Resetting
    #   - Tokenization
    #   - Lexical Analysis
    #   - Parsing/Construction
    #   - Validation
    def build(self):
        self.reset()
        tokens = self.__tokenizer.tokenize(self.__expression)
        token_objs = self.__lexer.lex(tokens)
        self.parse(token_objs)
        if not self.__validate_parse_tree():
            raise InvalidExpressionError('Invalid Expression')
        self.__prev_build = self.__expression

    # Builds the parse tree if it is not already built
    def __prepare(self) -> None:
        if self.__prev_build != self.__expression:
            self.build()

    # Validates the structure of the parse tree
    #   - Operands have no child nodes
    #   - Operators have exactly 2 child nodes
    def __validate_parse_tree(self) -> bool:
        def __internal_recursive(node: MathNode) -> bool:
```



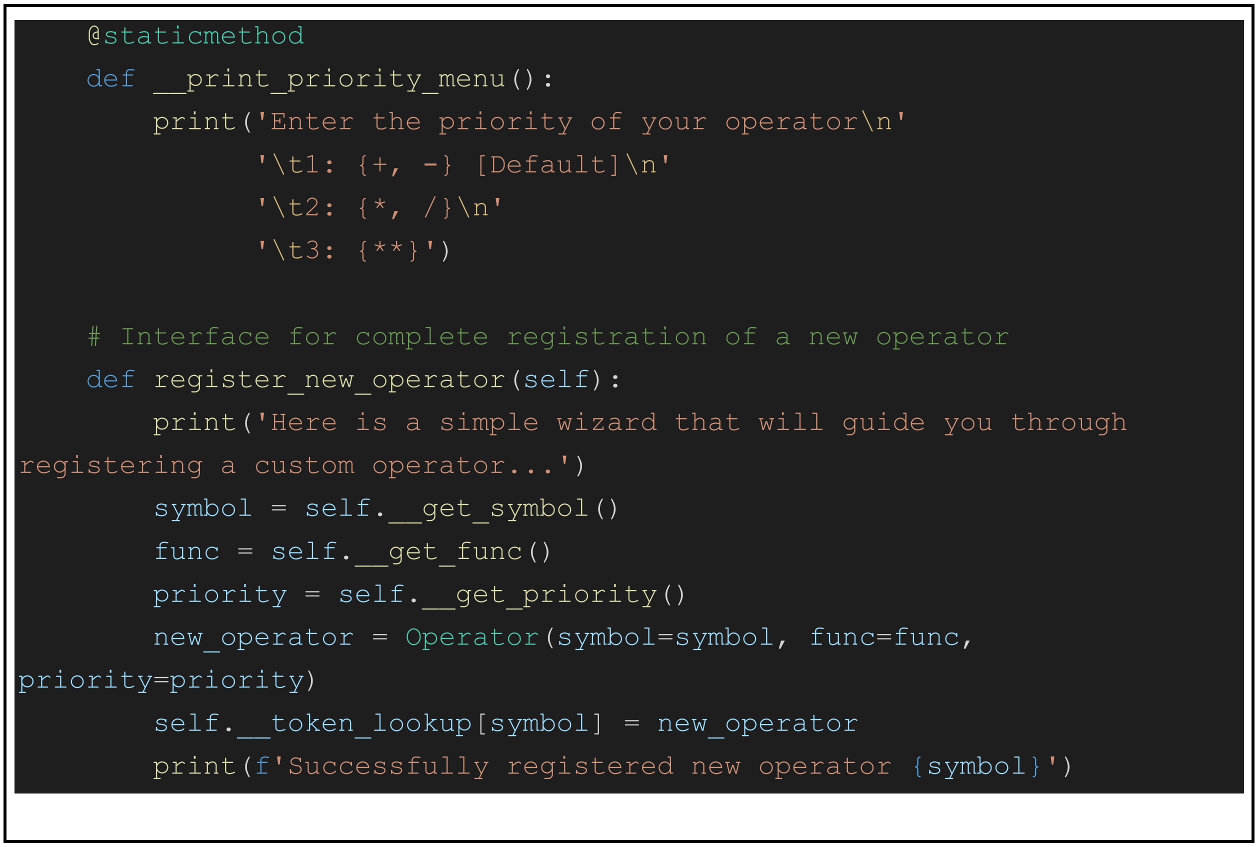
```
            if isinstance(node, Operator):
                if node._left is None or node._right is None:
                    return False
                return __internal_recursive(node._left) and
__internal_recursive(node._right)
            elif isinstance(node, Operand):
                if node._left is None and node._right is None:
                    return True
            return False

        if self._root is None:
            return False

        return __internal_recursive(node=self._root)

    # Option 1 - Assert given expression is fully parenthesised
    def validate_fully_parenthesised(self):
        return self.__expression.replace(' ', '') ==
self.reconstruct_expression().replace(' ', '')

    # Options 1, 2, 3 - Evaluate/Display the parse tree for given
expression(s)
    # Prints the tree (uses superclass Tree to do so)
    def print_tree(self):
        self.__prepare()
        super().print_tree()

    # Evaluates the expression
    def evaluate(self):
        self.__prepare()
        self._root()
        result = self._root._value
        return result if type(result) is int else round(result,
self.__precision)

    # Option 3 - Evaluate and Sort a list of expressions from a
specified text file
    def evaluate_and_sort(self):
        try:
```



```
            inputfile = input("Please enter input
file:\t".expandtabs(4))
            outputfile = input("Please enter output
file:\t".expandtabs(4))
            filename = open(inputfile, 'r').read().splitlines()

            lst = []
            for i in filename:
                self.read(i)
                exp = Expression(i, self.evaluate())
                lst.append(exp)

            mergeSort(lst)

            print("\n>>> Evaluation and sorting started:", end='')
            header = '\n\n*** Expressions with value ==> '
            exp = []
            prev_value = None
            for i in lst:
                current_value = i.get_result()
                if current_value != prev_value:
                    var = header + str(current_value)
                    exp.append(var)
                    prev_value = current_value
                exp.append("\n" + str(i))

            processed_expressions = ''.join(exp)
            print(processed_expressions)
            print("\n>>> Evaulation and sorting completed!")
            if outputfile != '':
                with open(outputfile, "w") as add_to_output_file:
                    add_to_output_file.write(processed_expressions[2:])
        except (ValueError, InvalidExpressionError):
            print('Input file contains one or more invalid expressions.
Aborting...')

    # Option 4 - Fully parenthesise a non-fully parenthesised expression
    def reconstruct_expression(self) -> str:
        self.__prepare()
```

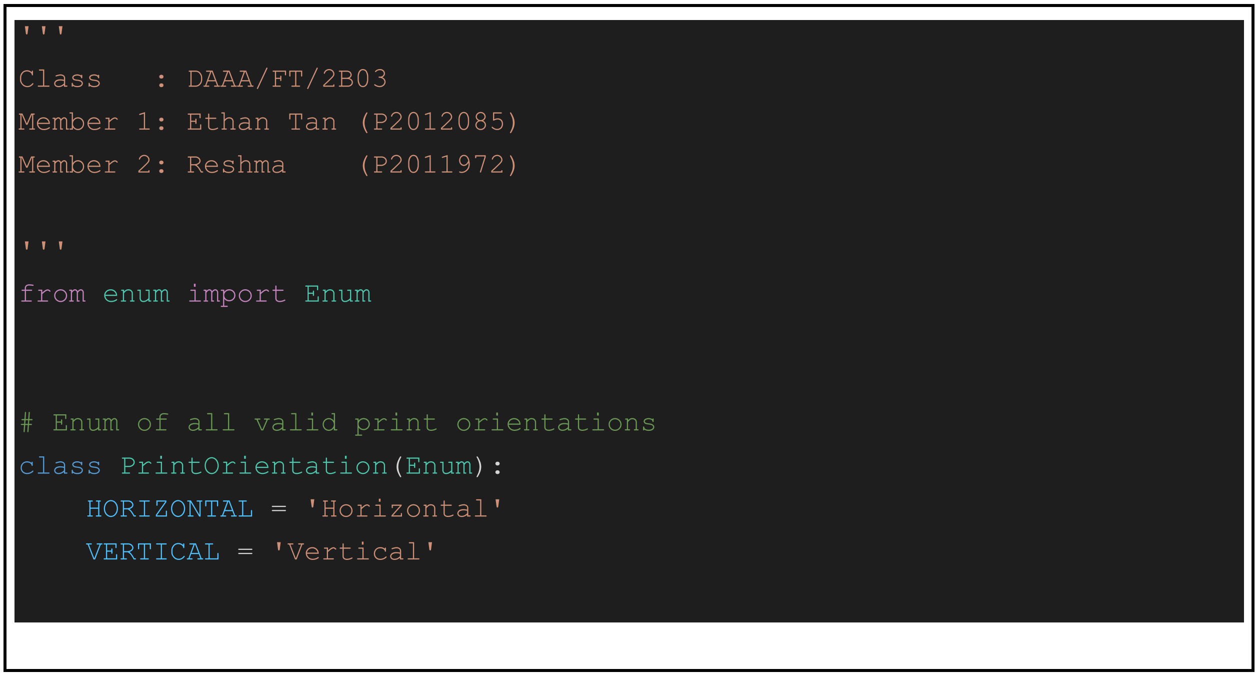


```
        def __reconstruct_internal(node):
            if node._left is not None and node._right is not None:
                return '(' + __reconstruct_internal(node._left) + ' ' +
str(node) + ' ' + __reconstruct_internal(node._right) + ')'
            return str(node)
        if self._root._left is None and self._root._right is None:
            return '(' + str(self._root) + ')'
        return __reconstruct_internal(node=self._root)

    # Option 6 - Register a New Operator
    # Allows the user to select the symbol for their custom operator
    def __get_symbol(self):
        symbol = input('Select a symbol besides [{}, ., (, )] with max
length of 3: '.format(', '.join(self.__token_lookup.keys()))).strip()
        if symbol in '.()' or symbol in self.__token_lookup.keys():
            raise InvalidOptionError(f'Illegal symbol encountered:
{symbol}')
        if len(symbol) > 3:
            raise InvalidOptionError(f'Symbol {symbol} is too long (max
3)')
        return symbol

    # Allows the user to select what their custom operator does
    #   - They have access to Python's standard math library's functions
    def __get_func(self):
        return eval('lambda a, b: ' + input('Enter function (in Python
format) (parameters are a and b): '))

    # Allows the user to select the priority of their custom operator
    def __get_priority(self):
        self.__print_priority_menu()
        priority_str = input('Operator Priority: ').strip()
        priority = int(priority_str) if priority_str != '' else 1
        if priority not in {1, 2, 3}:
            raise SyntaxError('Invalid priority (expected 1, 2 or 3)')
        return priority

    # Prints all available priority options
```



```
    @staticmethod
    def __print_priority_menu():
        print('Enter the priority of your operator\n'
              '\t1: {+, -} [Default]\n'
              '\t2: {*, /}\n'
              '\t3: {**}')

    # Interface for complete registration of a new operator
    def register_new_operator(self):
        print('Here is a simple wizard that will guide you through
registering a custom operator...')
        symbol = self.__get_symbol()
        func = self.__get_func()
        priority = self.__get_priority()
        new_operator = Operator(symbol=symbol, func=func,
priority=priority)
        self.__token_lookup[symbol] = new_operator
        print(f'Successfully registered new operator {symbol}')
```

src/print_orientation.py



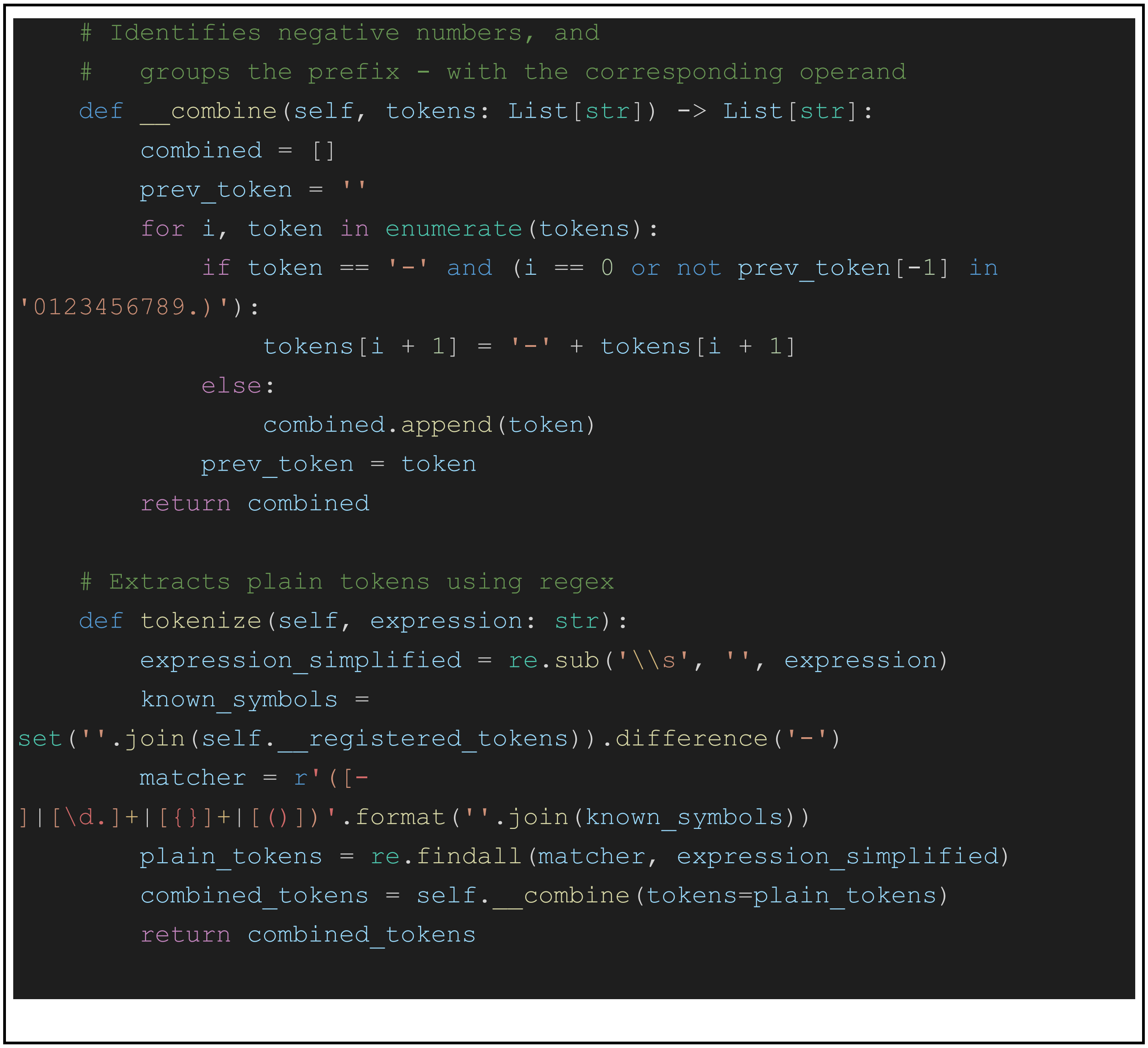
```
'''
Class   : DAAA/FT/2B03
Member 1: Ethan Tan (P2012085)
Member 2: Reshma    (P2011972)

'''
from enum import Enum


# Enum of all valid print orientations
class PrintOrientation(Enum):
    HORIZONTAL = 'Horizontal'
    VERTICAL = 'Vertical'
```

src/temp_node.py

```
'''
Class   : DAAA/FT/2B03
Member 1: Ethan Tan (P2012085)
Member 2: Reshma    (P2011972)

'''
from .node import Node


# Class for temporary nodes for vertical printing
class TempNode(Node):
    def __init__(self, width) -> None:
        super().__init__(value=' ' * width, width=width)

    # Pad with spaces
    def display(self):
        return ' ' * self._width
```

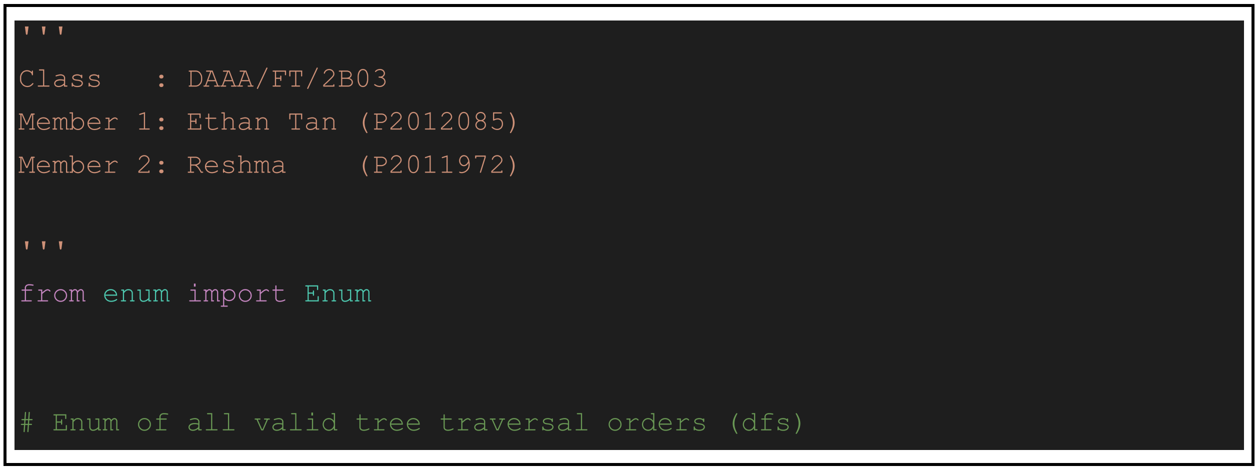
src/tokenizer.py

```
'''
Class   : DAAA/FT/2B03
Member 1: Ethan Tan (P2012085)
Member 2: Reshma    (P2011972)

'''
from typing import List, Set
import re


# Extracts tokens from a string expression
class Tokenizer:
    def __init__(self,
                 registered_tokens: Set[str]) -> None:
        self.__registered_tokens = registered_tokens
```



```
    # Identifies negative numbers, and
    #   groups the prefix - with the corresponding operand
    def __combine(self, tokens: List[str]) -> List[str]:
        combined = []
        prev_token = ''
        for i, token in enumerate(tokens):
            if token == '-' and (i == 0 or not prev_token[-1] in
'0123456789.)'):
                tokens[i + 1] = '-' + tokens[i + 1]
            else:
                combined.append(token)
            prev_token = token
        return combined

    # Extracts plain tokens using regex
    def tokenize(self, expression: str):
        expression_simplified = re.sub('\\s', '', expression)
        known_symbols =
set(''.join(self.__registered_tokens)).difference('-')
        matcher = r'([-
]|[\d.]+|[{}]+|[()])'.format(''.join(known_symbols))
        plain_tokens = re.findall(matcher, expression_simplified)
        combined_tokens = self.__combine(tokens=plain_tokens)
        return combined_tokens
```

src/tree_traversal_order.py

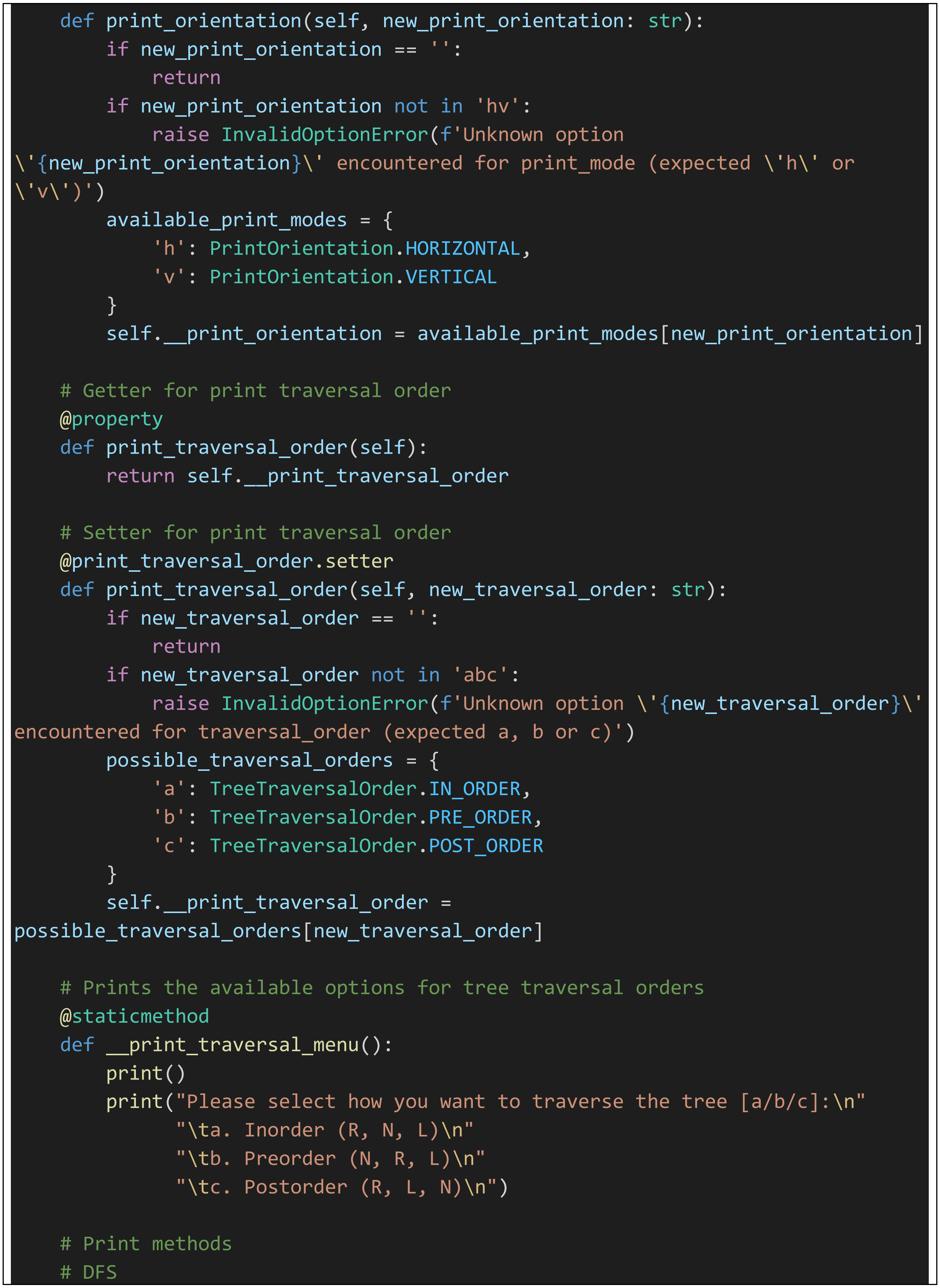


```
'''
Class    : DAAA/FT/2B03
Member 1: Ethan Tan (P2012085)
Member 2: Reshma    (P2011972)

'''
from enum import Enum



# Enum of all valid tree traversal orders (dfs)
```

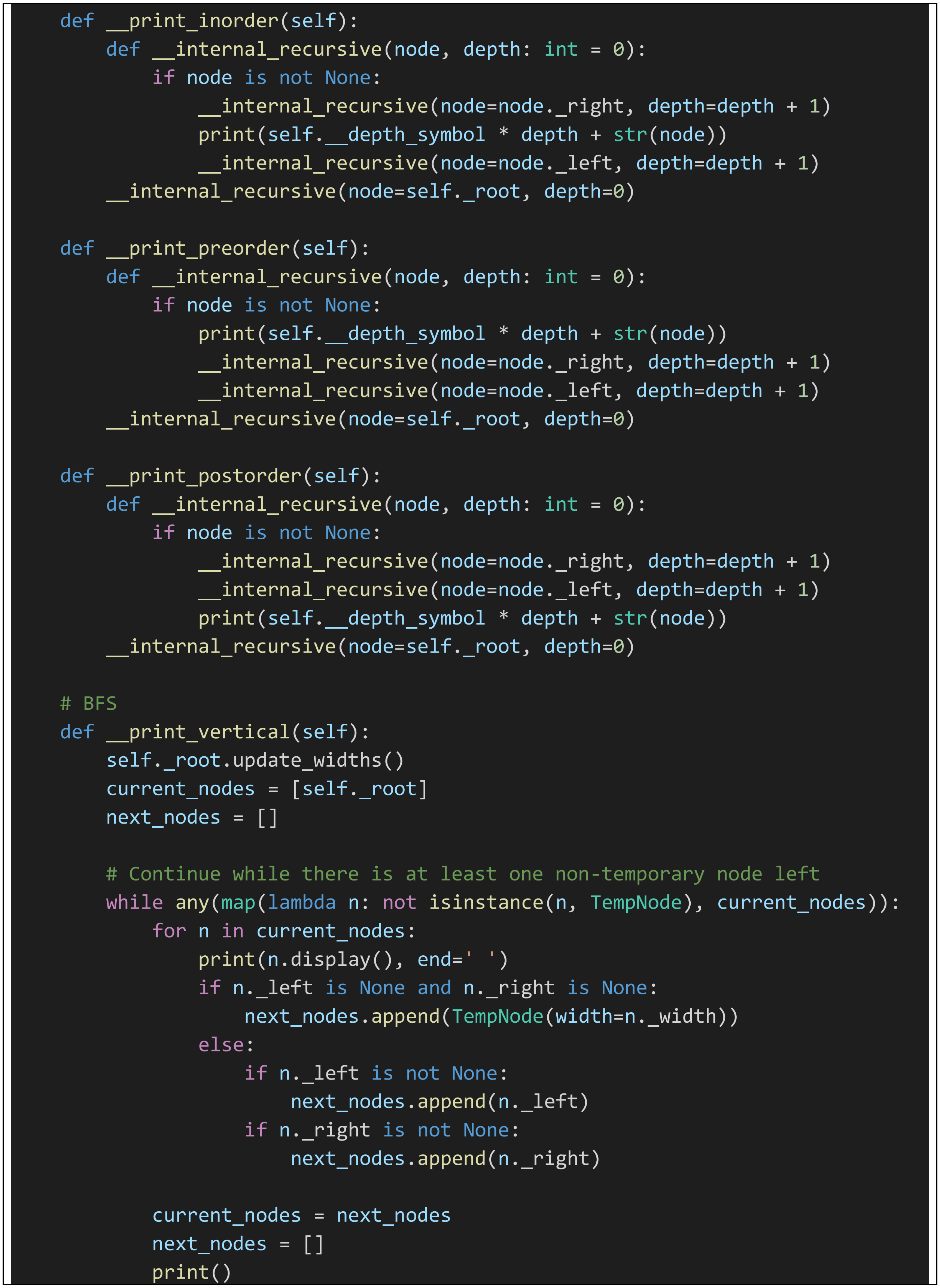
```
class TreeTraversalOrder(Enum):
    IN_ORDER = 'In-Order'
    PRE_ORDER = 'Pre-Order'
    POST_ORDER = 'Post-Order'
```

src/tree.py

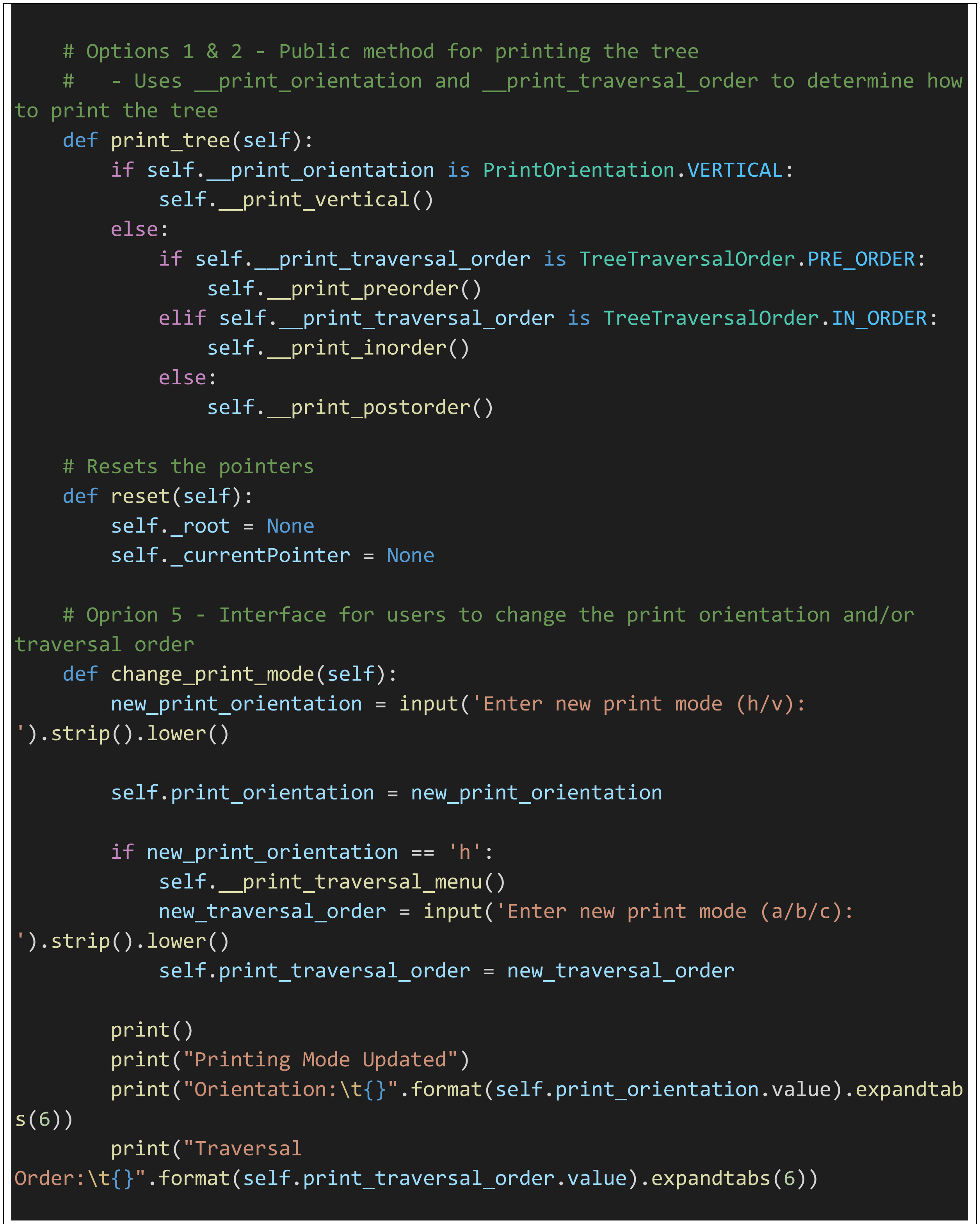
```
# type: ignore
'''
Class    : DAAA/FT/2B03
Member 1: Ethan Tan (P2012085)
Member 2: Reshma    (P2011972)

'''
from .temp_node import TempNode
from .exceptions import InvalidOptionError
from .print_orientation import PrintOrientation
from .tree_traversal_order import TreeTraversalOrder


# Class for generic trees
class Tree:
    def __init__(self,
                 depth_symbol: str = '.') -> None:
        # Protected properties
        self._root = None
        self._currentPointer = None

        # Private properties
        self.__depth_symbol = depth_symbol
        self.__print_orientation: PrintOrientation =
PrintOrientation.HORIZONTAL
        self.__print_traversal_order: TreeTraversalOrder =
TreeTraversalOrder.IN_ORDER

    # Getter for print orientation
    @property
    def print_orientation(self):
        return self.__print_orientation

    # Setter for print orientation
    @print_orientation.setter
```



```
    def print_orientation(self, new_print_orientation: str):
        if new_print_orientation == '':
            return
        if new_print_orientation not in 'hv':
            raise InvalidOptionError(f'Unknown option
\'{new_print_orientation}\' encountered for print_mode (expected \'h\' or
\'v\')')
        available_print_modes = {
            'h': PrintOrientation.HORIZONTAL,
            'v': PrintOrientation.VERTICAL
        }
        self.__print_orientation = available_print_modes[new_print_orientation]

    # Getter for print traversal order
    @property
    def print_traversal_order(self):
        return self.__print_traversal_order

    # Setter for print traversal order
    @print_traversal_order.setter
    def print_traversal_order(self, new_traversal_order: str):
        if new_traversal_order == '':
            return
        if new_traversal_order not in 'abc':
            raise InvalidOptionError(f'Unknown option \'{new_traversal_order}\'
encountered for traversal_order (expected a, b or c)')
        possible_traversal_orders = {
            'a': TreeTraversalOrder.IN_ORDER,
            'b': TreeTraversalOrder.PRE_ORDER,
            'c': TreeTraversalOrder.POST_ORDER
        }
        self.__print_traversal_order =
possible_traversal_orders[new_traversal_order]

    # Prints the available options for tree traversal orders
    @staticmethod
    def __print_traversal_menu():
        print()
        print("Please select how you want to traverse the tree [a/b/c]:\n"
              "\ta. Inorder (R, N, L)\n"
              "\tb. Preorder (N, R, L)\n"
              "\tc. Postorder (R, L, N)\n")

    # Print methods
    # DFS
```

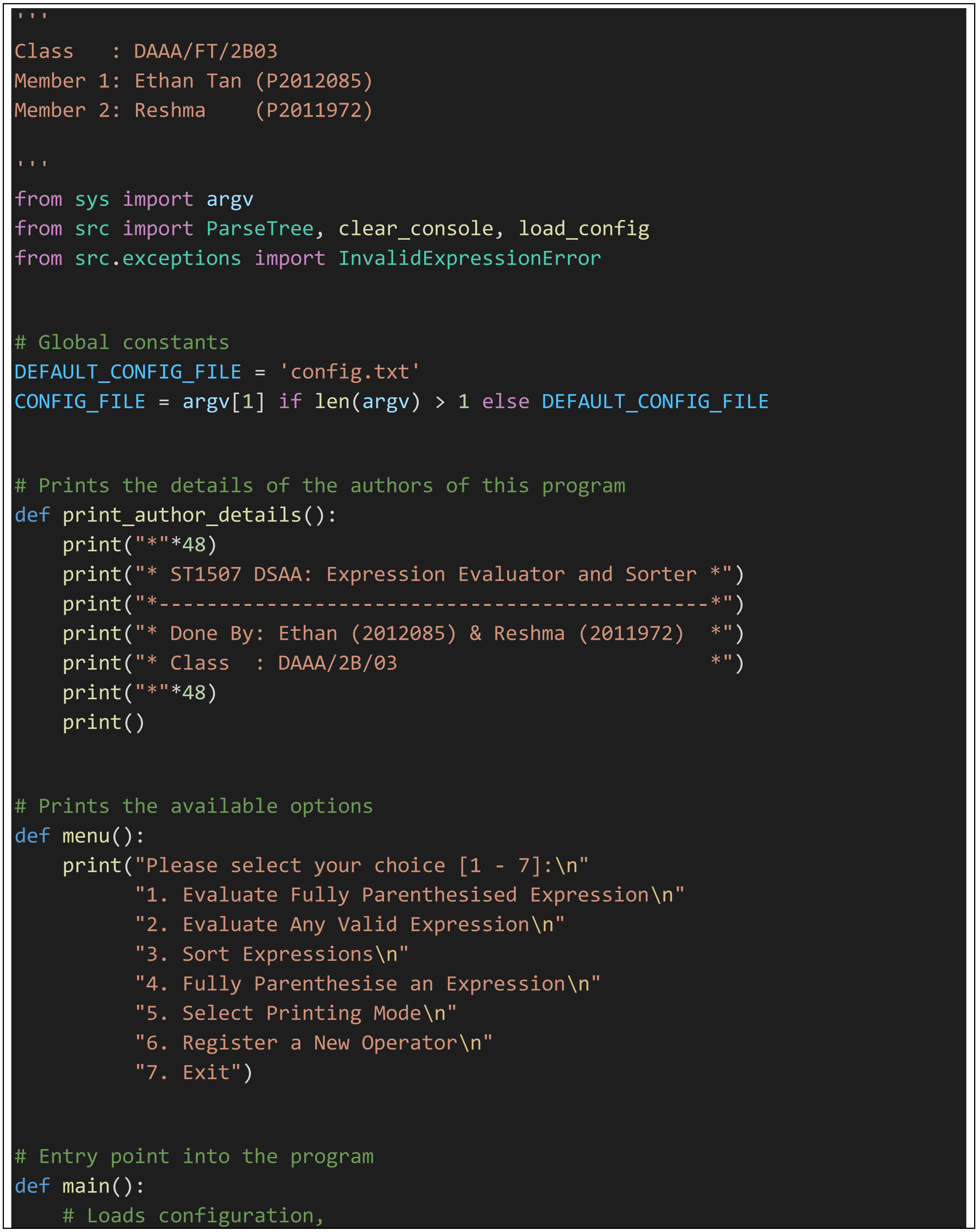


```python
    def __print_inorder(self):
        def __internal_recursive(node, depth: int = 0):
            if node is not None:
                __internal_recursive(node=node._right, depth=depth + 1)
                print(self.__depth_symbol * depth + str(node))
                __internal_recursive(node=node._left, depth=depth + 1)
        __internal_recursive(node=self._root, depth=0)

    def __print_preorder(self):
        def __internal_recursive(node, depth: int = 0):
            if node is not None:
                print(self.__depth_symbol * depth + str(node))
                __internal_recursive(node=node._right, depth=depth + 1)
                __internal_recursive(node=node._left, depth=depth + 1)
        __internal_recursive(node=self._root, depth=0)

    def __print_postorder(self):
        def __internal_recursive(node, depth: int = 0):
            if node is not None:
                __internal_recursive(node=node._right, depth=depth + 1)
                __internal_recursive(node=node._left, depth=depth + 1)
                print(self.__depth_symbol * depth + str(node))
        __internal_recursive(node=self._root, depth=0)

    # BFS
    def __print_vertical(self):
        self._root.update_widths()
        current_nodes = [self._root]
        next_nodes = []

        # Continue while there is at least one non-temporary node left
        while any(map(lambda n: not isinstance(n, TempNode), current_nodes)):
            for n in current_nodes:
                print(n.display(), end=' ')
                if n._left is None and n._right is None:
                    next_nodes.append(TempNode(width=n._width))
                else:
                    if n._left is not None:
                        next_nodes.append(n._left)
                    if n._right is not None:
                        next_nodes.append(n._right)

            current_nodes = next_nodes
            next_nodes = []
            print()
```

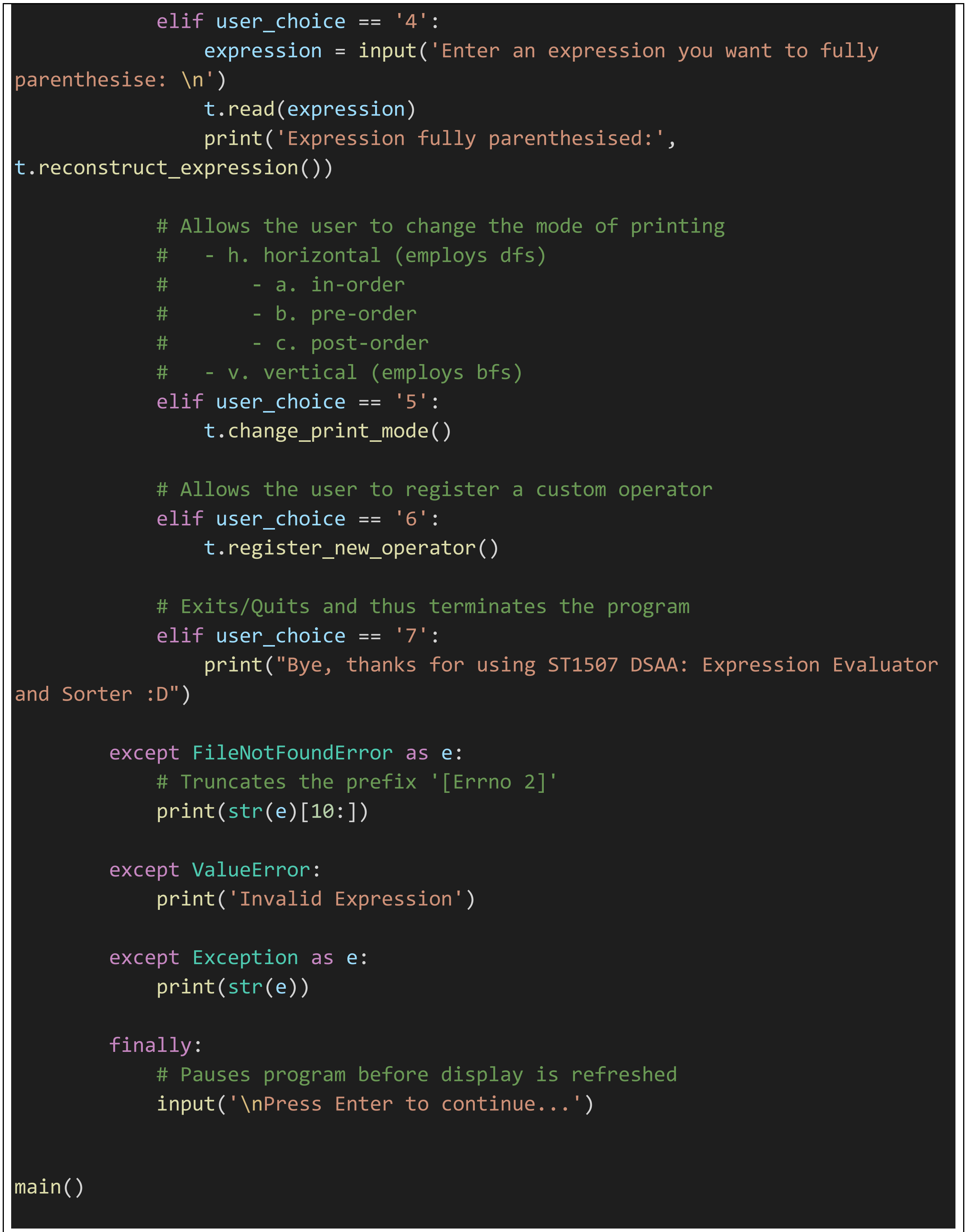


```
    # Options 1 & 2 - Public method for printing the tree
    #   - Uses __print_orientation and __print_traversal_order to determine how
to print the tree
    def print_tree(self):
        if self.__print_orientation is PrintOrientation.VERTICAL:
            self.__print_vertical()
        else:
            if self.__print_traversal_order is TreeTraversalOrder.PRE_ORDER:
                self.__print_preorder()
            elif self.__print_traversal_order is TreeTraversalOrder.IN_ORDER:
                self.__print_inorder()
            else:
                self.__print_postorder()

    # Resets the pointers
    def reset(self):
        self._root = None
        self._currentPointer = None

    # Oprion 5 - Interface for users to change the print orientation and/or
traversal order
    def change_print_mode(self):
        new_print_orientation = input('Enter new print mode (h/v):
').strip().lower()

        self.print_orientation = new_print_orientation

        if new_print_orientation == 'h':
            self.__print_traversal_menu()
            new_traversal_order = input('Enter new print mode (a/b/c):
').strip().lower()
            self.print_traversal_order = new_traversal_order

        print()
        print("Printing Mode Updated")
        print("Orientation:\t{}".format(self.print_orientation.value).expandtab
s(6))
        print("Traversal
Order:\t{}".format(self.print_traversal_order.value).expandtabs(6))
```

main.py



```python
'''
Class    : DAAA/FT/2B03
Member 1: Ethan Tan (P2012085)
Member 2: Reshma    (P2011972)

'''
from sys import argv
from src import ParseTree, clear_console, load_config
from src.exceptions import InvalidExpressionError


# Global constants
DEFAULT_CONFIG_FILE = 'config.txt'
CONFIG_FILE = argv[1] if len(argv) > 1 else DEFAULT_CONFIG_FILE


# Prints the details of the authors of this program
def print_author_details():
    print("*"*48)
    print("* ST1507 DSAA: Expression Evaluator and Sorter *")
    print("*----------------------------------------------*")
    print("* Done By: Ethan (2012085) & Reshma (2011972)  *")
    print("* Class  : DAAA/2B/03                          *")
    print("*"*48)
    print()


# Prints the available options
def menu():
    print("Please select your choice [1 - 7]:\n"
          "1. Evaluate Fully Parenthesised Expression\n"
          "2. Evaluate Any Valid Expression\n"
          "3. Sort Expressions\n"
          "4. Fully Parenthesise an Expression\n"
          "5. Select Printing Mode\n"
          "6. Register a New Operator\n"
          "7. Exit")


# Entry point into the program
def main():
    # Loads configuration,
    #   otherwise falls back to default options
    try:
        depth_symbol, operator_mode = load_config(config_file=CONFIG_FILE, default_config=['.', 1])
        t = ParseTree(depth_symbol=depth_symbol[0], mode=int(operator_mode))
    except:
        t = ParseTree()

    # Infinite loop,
    #   until user exits the program
    user_choice = ''
    while user_choice != '7':
        try:
            # Refresh the display
            clear_console()
            print_author_details()
            menu()

            # Allow user to select option
            user_choice = input("Enter choice: ").strip()

            # Evaluate an expression (1 - fully parenthesised only / 2 - any valid expression)
            #   - Parse tree will be displayed
            #   - Followed by the result of the evaluation
            if user_choice in {'1', '2'}:
                expression = input('Please enter the expression you want to evaluate:\n')
                t.read(expression)
                t.build()
                if user_choice == '1' and not t.validate_fully_parenthesised():
                    raise InvalidExpressionError(f'Expression is not fully parenthesised. Did you mean \'{t.reconstruct_expression()}\'?')
                print('\nParse Tree')
                print('----------')
                t.print_tree()
                print(f'\nExpression evaluates to:')
                print(t.evaluate())

            # Evaluate and sort expressions from a specified text file
            #   - The user can choose to save the results
            elif user_choice == '3':
                t.evaluate_and_sort()

            # Fully parenthesise a non-fully parenthesised valid expression
```



```
            elif user_choice == '4':
                expression = input('Enter an expression you want to fully
parenthesise: \n')
                t.read(expression)
                print('Expression fully parenthesised:',
t.reconstruct_expression())

            # Allows the user to change the mode of printing
            #   - h. horizontal (employs dfs)
            #       - a. in-order
            #       - b. pre-order
            #       - c. post-order
            #   - v. vertical (employs bfs)
            elif user_choice == '5':
                t.change_print_mode()

            # Allows the user to register a custom operator
            elif user_choice == '6':
                t.register_new_operator()

            # Exits/Quits and thus terminates the program
            elif user_choice == '7':
                print("Bye, thanks for using ST1507 DSAA: Expression Evaluator
and Sorter :D")

        except FileNotFoundError as e:
            # Truncates the prefix '[Errno 2]'
            print(str(e)[10:])

        except ValueError:
            print('Invalid Expression')

        except Exception as e:
            print(str(e))

        finally:
            # Pauses program before display is refreshed
            input('\nPress Enter to continue...')


main()
```

References

- Python animated_parse_tree package (https://github.com/ethanolx/Animated-Parse-Tree-py)