SCHOOL OF COMPUTING (SOC)

Diploma in Applied AI and Analytics

ST1507 DATA STRUCTURES AND ALGORITHMS (AI)

2025/26 SEMESTER 1
ASSIGNMENT TWO (CA2)

~ Restoring old newspapers ~
(using prefix tries & predictive text analysis)

Name: Aaron Ng(P2442631) & Stephen Bermudo (P2442657)

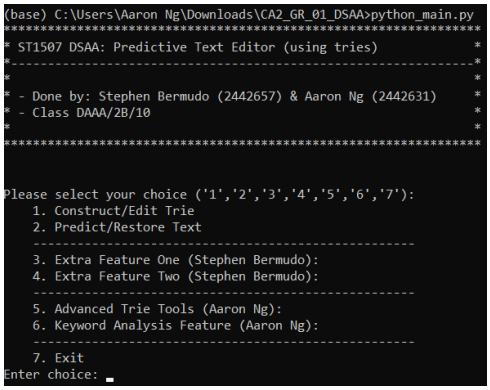
Class: DAAA/FT/2A/03

Group Number: 01

1. User Guide & Application Usage

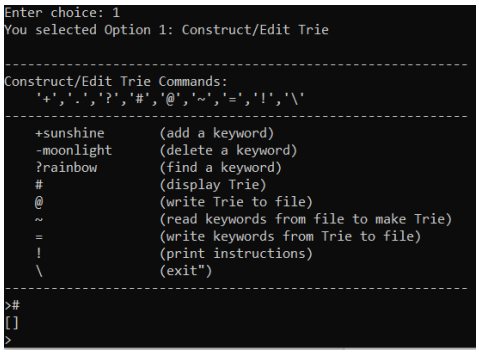
1.1 Launching the Application

Open Anaconda Prompt. Navigate to your project directory. Run the application using 'python_main.py'. It shows the title and the Menu Options:

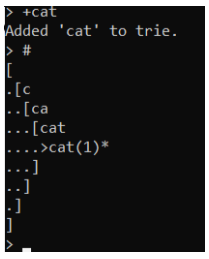


1.2 Feature One: Constructing & Editing Trie

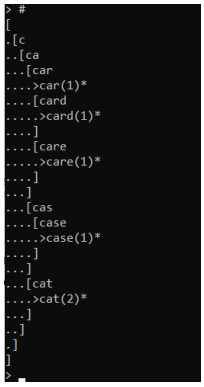
When the user selects Option 1, they are brought to the Construct/Edit Trie program. This feature allows users to build a prefix trie data structure by adding, deleting, searching, displaying, saving, and loading words from files. Upon entering this mode, the application will display the following command menu. Initially, the Trie is empty, as represented by [].



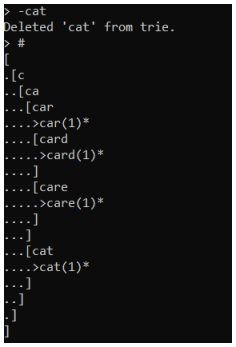
1.2.1 (+) Add A Keyword: User has to use +(keyword) to add a keyword into the trie. Then use # to display the Trie



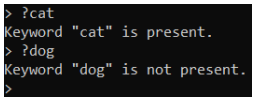
User can add more keywords e.g. card, care, case and the trie shows the same parent of different letters in the trie. If the user add the same keyword twice, it shows (2)*.



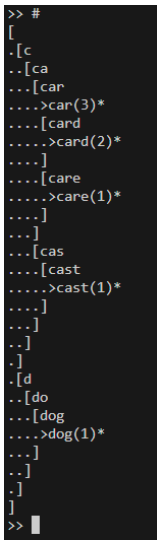
1.2.2 (-) Delete A Keyword: User can enter –(keyword) to remove a keyword by frequency.



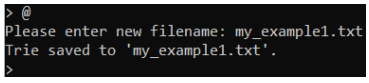
1.2.3 (?) Find A Keyword: User can enter ?(keyword) to see if it is present in the trie.



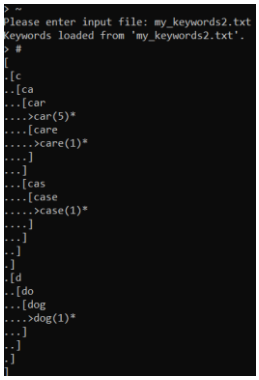
1.2.4 (#) Display Trie: User can enter #(keyword) to see the entire Trie.



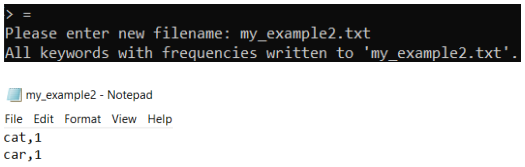
1.2.5 (@) Write Trie To File: User can enter @ and it will ask the user to give the name to the txt file and the current trie will be saved into the file.



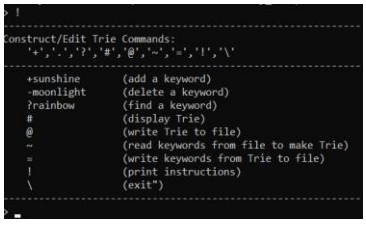
1.2.6 (~) Read keywords From File To Make Trie: User can put ~filename to read keywords to trie.



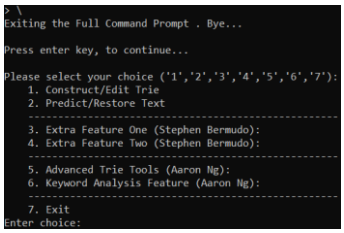
1.2.7 (=) Write Keywords From Trie To File: User can enter = to save the trie to a txt file and name it



1.2.8 (\) Print Instructions: User can enter ! to see the instructions again.

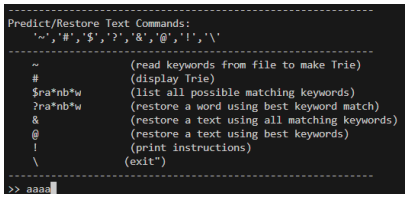


1.2.9 Exit Feature One: User can enter \ to exit and press enter again to see the Menu Options

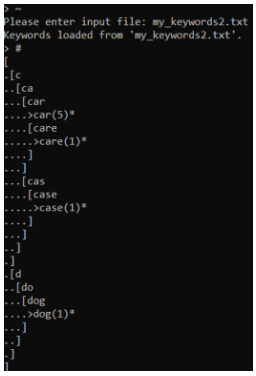


1.3 Feature Two: Predicting & Restoring Text

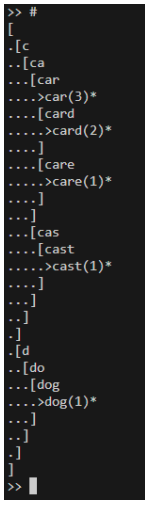
When the user selects Option 2, they are brought to the Predict/Restore Trie program. This feature allows users to restore or predict texts, with wild cards (*)



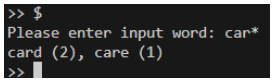
1.3.1 (~) Read keywords From File To Make Trie: User can put ~filename to read keywords to Trie, uses the same code as 1.2.6



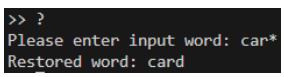
1.3.2 (#) Display Trie: User can enter #(keyword) to see the entire Trie. Functions and has the same code as 1.2.4



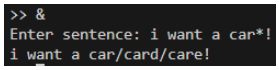
1.3.3 ($) Gets all matching keywords: Lists all possible keywords and their frequencies.



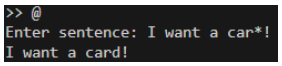
1.3.4 (?) Restore by using best keyword match: Restores a word using the most frequent word.



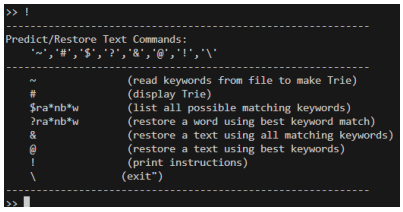
1.3.5 (&) Restores a text by using all matching keywords



1.3.6 (@) Restores a text by using Most frequent keyword.



1.3.7 (!) Prints Instructions: User can enter ! to see the instructions again. Uses the same code as 1.2.8



1.3.8 Exit Feature Two: User can enter \ to exit and press enter again to see the Menu Options. Uses the same code as 1.2.9.

2 Object-Oriented Programming (OOP) Implementation

The Predictive Text Editor application is developed using an Object-Oriented Programming (OOP) approach. Principles such as encapsulation, inheritance, and polymorphism form the foundation of its design, contributing to a modular, scalable, and maintainable codebase. This section discusses how each of these principles has been applied and elaborates on the key classes and their responsibilities within the system.

2.1 Encapsulation

Encapsulation is achieved by grouping related data and operations within dedicated classes. In our implementation, the Trie and TrieNode classes encapsulate all logic related to Trie operations, including insertion, deletion, searching, displaying, and file input/output. The TrieEditor class is responsible for handling user commands and acts as the controller that bridges user input with the underlying Trie data structure. The UserInterface class is solely focused on presenting output and displaying user instructions. By clearly separating these responsibilities, the application maintains high cohesion within each class and low coupling between them, which enhances modularity and simplifies maintenance.

2.2 Inheritance

Inheritance is incorporated into the design to facilitate code reuse and support future feature expansion. Although the current system does not employ deep inheritance hierarchies due to the relatively simple scope of the application, the class architecture has been structured to allow straightforward extensions. For example, the TrieEditor class could be extended into specialized subclasses such as PredictiveEditor or AnalysisEditor to add advanced editing capabilities. Similarly, the UserInterface class could be subclassed into a GUIUserInterface to enable graphical user interactions without modifying existing functionality. While inheritance is not heavily utilised in the present version, the system is designed with scalability in mind, enabling more extensive use of this principle in future developments.

2.3 Polymorphism

Polymorphism is demonstrated through the flexible command parsing system implemented within the TrieCommandHandler class. The command_prompt() method dynamically responds to different command types (such as +, -, ?, @, ~, and others) based on user input. While Python does not support traditional function overloading, polymorphic behaviour is achieved through conditional dispatching, where each command triggers a specific operation. For instance, when the + command is entered with a valid alphabetical argument, the application inserts the word into the trie and confirms the addition. If an invalid argument is supplied, the system provides an error message without interrupting execution. Similarly, the - command removes a specified word if it exists in the trie and notifies the user of the outcome. For example:

```
if cmd == '+':
    if arg.isalpha():
        self.trie.insert(arg)
        print(f"Added '{arg}' to trie.")
    elif arg:
        print("Invalid input! Only letters allowed.")
    else:
        print("Please provide a word to add.")

elif cmd == '-':
    if arg.isalpha():
        if self.trie.search(arg):
            self.trie.delete(arg)
            print(f"Deleted '{arg}' from trie.")
        else:
            print("Is not a keyword in trie.")
    elif arg:
        print("Invalid input! Only letters allowed.")
    else:
        print("Please provide a word to delete.")
```

This command-handling structure supports easy extension: new commands can be incorporated without altering the core loop logic. By allowing multiple command types to be processed through a single interface, the application demonstrates polymorphic design principles in a way that ensures adaptability and future scalability.

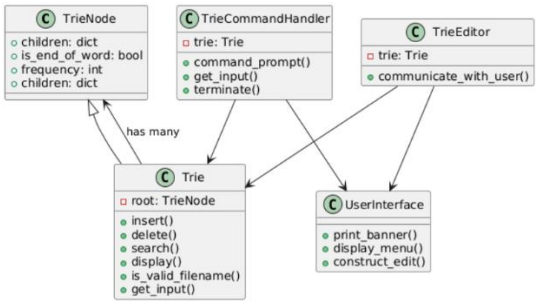
2.4 Class Responsibilities and Interactions

The main classes and their responsibilities are as follows:

Class Name	Responsibility
TrieNode	Represents a node in the Trie, storing child nodes, word-end status, and frequency.
Trie	Implements all Trie-related operations: insert, delete, search, display, file I/O, and prefix matching.
TrieEditor	Handles user commands to modify and interact with the Trie. Interfaces between user and data structure.
UserInterface	Displays banners, menus, and usage instructions. Handles text UI output.
main.py	Main entry point that manages the application loop and dispatches control to various features.
TrieCommamdHandler	Gets the input of the user, and initiates the commands based on UserInputs and passing functions.

2.5 Class Diagram

This is the Class Diagram that shows relationships under Predictive Text Editor:



2.6 Examples of OOP Principles in Code

Below are examples demonstrating encapsulation and method delegation in the Trie class:

Example 1: Encapsulation in the insert() method

```
class Trie:
    def insert(self, word):
        current = self.root
        for ch in word:
            if ch not in current.children:
                current.children[ch] = TrieNode()
            current = current.children[ch]
        if current.is_end_of_word:
            current.frequency += 1
        else:
            current.is_end_of_word = True
            current.frequency = 1
```

The insert() function encapsulates how the Trie grows, shielding other classes from knowing the internal node logic.

Example 2: Method Delegation in TrieCommandHandler.command_prompt()

```
elif cmd == '+':
    if arg.isalpha():
        self.trie.insert(arg)
        print(f"Added '{arg}' to trie.")
    elif arg:
        print("Invalid input! Only letters allowed.")
    else:
        print("Please provide a word to add.")
```

Here, the TrieCommandHandler class delegates the actual insertion operation to the Trie class, demonstrating how higher-level controllers can rely on encapsulated lower-level methods without handling internal details.

3 Data Structures and Algorithms

3.1 Overview of Core Data Structures

The primary data structure used in the Predictive Text Editor is the **Trie** (prefix tree), which is particularly suited for autocomplete and predictive text functionalities. The Trie is implemented from scratch in Python, with each node represented by the TrieNode class. A TrieNode stores its child nodes in a Python dictionary, a Boolean flag is_end_of_word to indicate whether the node corresponds to the end of a valid word, and an integer frequency counter to record how many times a word has been inserted. The Trie class encapsulates all core operations, including insertion, deletion, searching, and displaying words, as well as retrieving keywords based on a given prefix. It also supports wildcard prefix matching and file input/output for saving and loading keyword data. This structure enables efficient storage and retrieval of words, making it ideal for the application's predictive capabilities.

3.2 Design Justification and Suitability

A Trie was selected over alternative data structures such as hash tables or balanced search trees due to its superior efficiency in handling prefix-based operations. While hash tables are efficient for exact lookups, they are less suitable for prefix queries, requiring additional logic to filter matching entries. The Trie allows both insertion and prefix lookups in **O(m)** time, where *m* is the length of the word or prefix, making it highly efficient for real-time text prediction tasks. By using Python dictionaries for the children attribute of each TrieNode, the implementation benefits from constant-time access to child nodes. Furthermore, Python's built-in data types such as dict, list, and str simplify the design, resulting in an implementation that is both performant and easy to maintain.

3.3 Algorithm Performance and Complexity

Below is a summary of the time complexities of the key algorithms implemented within the Trie:

Operation	**Time Complexity**	**Explanation**
Insert a word	O(m)	Traverse each character of the word once; m is the word length.
Search for a word	O(m)	Follows the same logic as insert; checks each character step by step.
Delete a word	O(m)	Recursive depth-first deletion through the Trie based on characters.
Display all words	O(n)	Full traversal of the Trie, where n is total characters across all words.
Get all words with prefix	O(p + k)	p is the prefix length, k is the number of matching words.
Load keywords from file	O(n * m)	Loads and inserts each word from file into the Trie individually.
Save Trie to file (visual)	O(n)	Performs depth-first traversal to write structured Trie to file.
Get words with prefix (wildcard * support)	$O(b^a)$ worst-case	Performs DFS through all branching possibilities for each *. a is the number of wildcards, b is average branching factor.
Find best match (wildcard search)	$O(b^a)$ worst-case	Traverses all possible paths for wildcards, tracking the highest-frequency match found.
Separate words in a sentence	O(w)	Splits a string into w words using whitespace and punctuation separation.
Loop sentence with best matches	$O(w \times b^a)$ worst-case	For each word, attempts wildcard resolution using find_best_match; returns restored sentence.
Loop sentence with all matches	$O(w \times b^a)$ worst-case	For each word containing wildcards, retrieves all possible matches, joining them in the output.

These algorithms were carefully implemented to avoid redundant work and ensure responsiveness even when handling large dictionaries. For example, the delete method decrements the frequency of a word and only deletes nodes when necessary, preventing excessive modification to the structure.

3.4 Summary Table of Data Structures Used

Data Structure	**Type**	**Usage Purpose**	**Justification for Use**
TrieNode	Custom Class	Represents individual nodes in the Trie	Allows easy management of children, word-end status, and frequency tracking.

Trie	Custom Class	Implements the main Trie functionality	Efficient for insert/search/predict operations; scalable for large vocabularies.
dict	Built-in (Python)	Maps characters to child TrieNode objects	Average O(1) access time for key lookups, improving Trie performance.
list	Built-in (Python)	Collects and stores words during traversal and predictions	Flexible size, good for temporary storage of results.
str	Built-in (Python)	Handles user input, prefixes, keywords, and file paths	Lightweight, immutable, and optimized for text operations.
input() and print()	Built-in (Python)	User input and interaction via the command prompt	Facilitates a simple and readable terminal interface for user interaction.
UserInterface	Custom Class	Manages terminal-based user interaction	Encapsulates input/output logic, improving code modularity and maintainability.
TrieCommandHandler	Custom Class	Interprets and executes user commands (e.g., insert, search, predict)	Separates application logic from data structure implementation, improving code clarity.
Tuple	Built-in (Python)	Used to return multiple values from functions (e.g., word-frequency pairs)	Immutable, lightweight way to group related values.
Int	Built-in (Python)	Tracks word frequencies and counts	Minimal memory overhead for numerical data.

The application's design is grounded on efficient, well-suited data structures with predictable algorithmic performance. The custom Trie and its accompanying logic provide fast and scalable solutions for text prediction and manipulation tasks, aligning well with the requirements of a predictive text editor.

4 Challenges Faced, Key Takeaways and Learning Achievements

4.1 Challenges

During development, we encountered several technical and collaborative challenges. On the technical side, managing edge cases in Trie deletion proved tricky, particularly when ensuring that no unintended data loss occurred. Additionally, debugging file-loading errors was time-consuming, as it involved tracing through multiple dependencies. From a teamwork perspective, aligning different coding styles and merging code without conflicts was challenging, especially when working remotely under tight deadlines. Trying to find a way to add the wildcard search with a less than exponential rate of Big O cost.

4.2 Key Takeaways

This project reinforced the importance of clean, modular code and consistent coding practices. Version control through GitHub proved invaluable, not just for storing code, but for effective collaboration. Regular commits, detailed commit messages, and clear pull request reviews allowed us to integrate work smoothly and avoid major conflicts. We also learned the value of continuous testing throughout development, catching bugs early before they grew into more

complex problems. Effective communication, supported by comments in code and updates in the repository, ensured that everyone stayed informed and on track.

4.3 Learning Achievements

Through this project, we gained practical experience in implementing and optimizing data structures such as Tries. We strengthened our problem-solving skills by tackling edge cases and performance issues and improved our debugging techniques. Equally important, we developed teamwork and collaboration skills that are essential in real-world software development. By working together on a shared codebase, we learned how to balance individual contributions with group objectives, adapt to each other's work styles, and deliver a functional, well-structured Python application on schedule.

5 Roles and Contributions of Each Member in the Team

5.1 Aaron Ng

Aaron Ng was primarily responsible for implementing the main program loop that integrates all features of the application, allowing users to navigate between different modes through the main menu. He developed Option 1, “Construct/Edit Trie”, along with its dedicated command-line interface, which enables users to add, delete, search, display, load, and save keywords. Aaron designed and implemented the core Trie data structure in trie.py, including both the TrieNode and Trie classes. These classes handle essential operations such as inserting, deleting, and searching for words; displaying the Trie with indentation and frequency counts; retrieving all stored words with their frequencies; and loading keywords from files. In TrieCommandHandler.py, Aaron implemented the command processing logic for the “Construct/Edit Trie” mode, mapping user commands (+, -, ?, #, @, ~, =, !, and) to their corresponding Trie operations. His implementation also included input parsing, file name validation, and program termination handling.

5.2 Stephen Bermudo

Stephen Bermudo was responsible for designing and implementing the “Predict/Restore Text” feature of the application, which is accessed via Option 2 in the main menu. He built the command-line interface for this feature within TrieCommandHandler.py, enabling users to perform operations such as loading keywords from a file, displaying the Trie, listing possible matches for partially entered words, and restoring individual words or entire sentences with missing characters. His implementation supported user commands (~, #, $, ?, &, @, !, and), ensuring each was linked to the correct underlying Trie operation. Stephen also created the predict_restore method in user_interface.py, which presents a clear and user-friendly list of commands with descriptions for the “Predict/Restore Text” mode. This improved usability by allowing users to quickly understand how to operate the feature.

6 Summary

This project successfully implemented a command-line-based predictive text editor using the Trie data structure. Through collaborative effort, the team designed and developed features that allow users to construct, edit, and analyze keywords efficiently. Despite encountering technical and coordination challenges, the group adapted and improved through teamwork and problem-solving. The final product reflects a strong understanding of data structures, file handling, and user interaction in Python, and demonstrates each member's contributions to building a functional and modular application.

7 Appendix

7.1 References

GeeksforGeeks. (n.d.). Trie (2025) | (Insert and Search). [online] Available at: https://www.geeksforgeeks.org/trie-insert-and-search/[Accessed 9 Jul. 2025].

Miller, B.N. and Ranum, D.L. (2011). *Problem Solving with Algorithms and Data Structures Using Python*. 2nd ed. [online] Runestone Interactive. Available at: https://runestone.academy/ns/books/published/pythonds/index.html[Accessed 10 Jul. 2025].

Bari, A. (2016). *Trie Data Structure*. [video] YouTube. Available at: https://www.youtube.com/watch?v=zIjfhVPRZCg [Accessed 8 Jul. 2025].

7.2 Source Code

python_main.py:

```
# Group Members Info
group_members = [
"Stephen Bermudo (2442657)",
"Aaron Ng (2442631)"
]

# Import required classes from other modules
from trie import Trie
from trie_editor import TrieEditor
from TrieCommandHandler import TrieCommandHandler
from feature_advanced_editor import AdvancedTrieFeature
from user_interface import UserInterface
from keyword_analysis_feature import KeywordAnalysisFeature

# --------------------------
# Main program loop (Done by Aaron)
# --------------------------
def main():
trie_instance = Trie()
TCH = TrieCommandHandler()
UI = UserInterface()

# Display welcome banner once at the start
UI.print_banner()

# Main menu loop
while True:
UI.display_menu()
choice = input("Enter choice: ").strip()

# --------------------------
# Option 1: Construct/Edit Trie (Done by Aaron)
# --------------------------
if choice == '1':
print("You selected Option 1: Construct/Edit Trie\n")
TCH.command_prompt("construct_edit")
```

```
# -------------------------
# Option 2: Predict/Restore Text (Done by Stephen)
# -------------------------
elif choice == '2':
print("You selected Option 2: Predict/Restore Text\n")
TCH.command_prompt("predict_restore")

# -------------------------
# Extra Feature One (To be implemented by Stephen)
# -------------------------
elif choice == '3':
print("You selected Trie Charter (Stephen Bermudo)\n")
TCH.command_prompt("trieChart")

# -------------------------
# Extra Feature Two (To be implemented by Stephen)
# -------------------------
elif choice == '4':
print("You selected Inbuilt Analytics (Stephen Bermudo)\n")
TCH.command_prompt("autoComplete")

# -------------------------
# Option 5: Advanced Trie Tools (Done by Aaron)
# -------------------------
elif choice == '5':
print("You selected Option 5: Advanced Trie Tools (Aaron Ng)\n")
feature5 = AdvancedTrieFeature(trie_instance)
feature5.run()

# -------------------------
# Option 6: Keyword Analysis Feature (Done by Aaron)
# -------------------------
elif choice == '6':
print("You selected Option 6: Keyword Analysis Feature (Aaron Ng)\n")
feature6 = KeywordAnalysisFeature(trie_instance)
feature6.run()

# -------------------------
# Option 7: Exit the program
# -------------------------
elif choice == '7':
print("Exiting the program. Goodbye!\n")
break

# -------------------------
# Handle invalid menu input
# -------------------------
else:
print("Invalid choice! Please enter a number from 1 to 7.\n")
```

```
# Start the main program
if __name__ == "__main__":
main()
```

user_interface.py:

```
import os

#Class Created By Stephen
class UserInterface():

# Function to display the banner (Aaron)
def print_banner(self):
print("**********************************************************************")
print("* ST1507 DSAA: Predictive Text Editor (using tries) *")
print("*----------------------------------------------------------------*")
print("* *")
print("* - Done by: Stephen Bermudo (2442657) & Aaron Ng (2442631) *")
print("* - Class DAAA/2B/10 *")
print("* *")
print("**********************************************************************\n\n")

def quit_text(self):
Print("Exiting the Full Command Promot . Bye...")

# Function to display the menu (Aaron)
def display_menu(self):
print("Please select your choice ('1','2','3','4','5','6','7'):")
print(" 1. Construct/Edit Trie")
print(" 2. Predict/Restore Text")
print(" ------------------------------------------------")
print(" 3. Visualise Trie Charts (Stephen Bermudo):")
print(" 4. AutoComplete (Stephen Bermudo):")
print(" ------------------------------------------------")
print(" 5. Advanced Trie Tools (Aaron Ng):")
print(" 6. Keyword Analysis Feature (Aaron Ng):")
print(" ------------------------------------------------")
print(" 7. Exit")

# Function to Display Feature 1 (Aaron)
def construct_edit(self, show_empty_trie=True):
print("-----------------------------------------------------------")
print("Construct/Edit Trie Commands:")
print(" '+','.','?','#','@','~','=','!','\\'")
print("-----------------------------------------------------------")
print(" +sunshine (add a keyword)")
print(" -moonlight (delete a keyword)")
print(" ?rainbow (find a keyword)")
print(" # (display Trie)")
print(" @ (write Trie to file)")
```

```
print(" ~ (read keywords from file to make Trie)")
print(" = (write keywords from Trie to file)")
print(" ! (print instructions)")
print(" \\ (exit\")")
print("-------------------------------------------------------")
# If Trie is empty
if show_empty_trie:
print(">#")
print("[]")

# Function to Display Feature 2 (Stephen)
def predict_restore(self):
print("-------------------------------------------------------")
print("Predict/Restore Text Commands:")
print(" '~','#','$','?','&','@','!','\\'")
print("-------------------------------------------------------")
print(" ~ (read keywords from file to make Trie)")
print(" # (display Trie)")
print(" $ra*nb*w (list all possible matching keywords)")
print(" ?ra*nb*w (restore a word using best keyword match)")
print(" & (restore a text using all matching keywords)")
print(" @ (restore a text using best keywords)")
print(" ! (print instructions)")
print(" \\ (exit\")")
print("-------------------------------------------------------")

def getTrieChart(self):
print("-------------------------------------------------------")
print("Trie Chart Drawing Menu:")
print(" '~','^','!','%','\\'")
print("-------------------------------------------------------")
print(" ~ (read keywords from file to make Trie)")
print(" * (Visualize Trie Structure with NetworkX)")
print(" ^ (Visualize Longest Trie Structure with NetworkX)")
print(" ! (Visualize Specifc Word/Prefix Path with NetworkX)")
print(" % (Generate Chart by frequency)")
print(" \\ (exit\")")
print("-------------------------------------------------------")

def autoCompleteGame(self):
print("-------------------------------------------------------")
print("Auto Complete Menu:")
print(" '~','#','1','2','\\'")
print("-------------------------------------------------------")
print(" ~ (read keywords from file to make Trie)")
print(" # (display Trie)")
print(" 1 (Start Guess)")
print(" 2 (Review Recent Guesses)")
print(" \\ (exit\")")
print("-------------------------------------------------------")
```

```
def Game_UI(self, guesses):
print("-----------------------------------------------------------")
print("My guesses: ")
print("-----------------------------------------------------------")
for i, guess in enumerate(guesses, 1):
if isinstance(guess, tuple):
word, freq = guess
print(f"{i} : {word} (freq: {freq})")
else:
print(f"{i} : {guess}")
print("-----------------------------------------------------------")

# Function to display Feature 5: Advanced Trie Tools (Aaron)
def display_advanced_trie_tools(self):
print("-----------------------------------------------------------------------")
print("Advanced Trie Tools - Feature 5 (Aaron Ng)")
print("-----------------------------------------------------------------------")
print(" ~file1,file2 (load and merge two Trie keyword files into one)")
print(" >file.txt (show top keywords by frequency from a file)")
print(" + (add a keyword to a TXT file and update Trie)")
print(" - (remove a keyword from a TXT file and update Trie)")
print(" ^ (replace 'old' keyword with 'new' in current Trie)")
print(" ! (print instructions again)")
print(" \\ (exit)")
print("-----------------------------------------------------------------------")

# Function to display Feature 6: Keyword Analysis Feature (Aaron)
def display_keyword_analysis_feature(self):
print("-----------------------------------------------------------------------")
print("Extra Feature Two - Keyword Tools (Aaron Ng)")
print("-----------------------------------------------------------------------")
print(" =file1,file2 (Compare common keywords in both TXT files)")
print(" >from,to (Transfer a keyword from one TXT file to another)")
print(" #file.txt (Show keywords from longest to shortest)")
print(" *file.txt (Group keywords alphabetically by first letter)")
print(" %file.txt (Show most frequent starting letters)")
print(" $file.txt (List palindromic keywords)")
print(" ! (Print instructions again)")
print(" \\ (Exit this feature)")
print("-----------------------------------------------------------------------")
```

trie.py:

```
import string

# ---------------------- Trie Node Class ----------------------
# Class created by Aaron to represent each node in the trie
class TrieNode:
def __init__(self):
self.children = {}
self.is_end_of_word = False
```

```
self.frequency = 0 # Stores frequency of word appearance

# ---------------------- Trie Class ----------------------
# Class created by Aaron. Main trie implementation to support insert, delete, search, etc.
class Trie:
def __init__(self):
self.root = TrieNode()

# Insert a word and update frequency
def insert(self, word):
current = self.root
for ch in word:
if ch not in current.children:
current.children[ch] = TrieNode()
current = current.children[ch]
if current.is_end_of_word:
current.frequency += 1 # increment if already exists
else:
current.is_end_of_word = True
current.frequency = 1 # new word

# Delete one occurrence of a word
def delete(self, word):
def _delete(node, word, depth):
if depth == len(word):
if not node.is_end_of_word:
return False # Word doesn't exist
node.frequency -= 1
if node.frequency <= 0:
node.is_end_of_word = False
node.frequency = 0
return len(node.children) == 0
return False
ch = word[depth]
if ch not in node.children:
return False
should_delete_child = _delete(node.children[ch], word, depth + 1)
if should_delete_child:
del node.children[ch]
return not node.is_end_of_word and len(node.children) == 0
return False
_delete(self.root, word, 0)

# Search for a word in the trie
def search(self, word):
current = self.root
for ch in word:
if ch not in current.children:
return False
```

```
current = current.children[ch]
return current.is_end_of_word

# Display the trie visually with indentations
def display(self, node=None):
if node is None:
node = self.root
def _display(current, prefix, depth):
lines = []
indent = '.' * depth

# Print the prefix at the current level
lines.append(indent + '[' + prefix)
# If it's a word, print it with frequency
if current.is_end_of_word:
lines.append('.' * (depth + 1) + f">" + prefix + f"({current.frequency})*")
# Recurse into children
for ch, next_node in sorted(current.children.items()):
lines.extend(_display(next_node, prefix + ch, depth + 1))

# Closing bracket
lines.append(indent + ']')
return lines
print("[")
for ch, next_node in sorted(node.children.items()):
result = _display(next_node, ch, 1)
for line in result:
print(line)
print("]")

# Return all words and their frequencies
def get_all_words_with_freq(self, prefix='', frequency=True):
# Helper: find the node of the prefix first
def find_prefix_node(node, prefix):
current = node
for ch in prefix:
if ch not in current.children:
return None
current = current.children[ch]
return current
start_node = find_prefix_node(self.root, prefix)
if not start_node:
return []
words = []
def _dfs(current, current_prefix):
if current.is_end_of_word:
if frequency:
words.append((current_prefix, current.frequency))
else:
words.append(current_prefix)
```

```
for ch, node in current.children.items():
_dfs(node, current_prefix + ch)
_dfs(start_node, prefix)
return words

# Load keywords from file and populate the trie
def load_keywords_from_file(self, filename):
try:
with open(filename, 'r') as f:
for line in f:
line = line.strip()
if line:
if ',' in line:
word, freq_str = line.rsplit(',', 1)
try:
freq = int(freq_str)
except ValueError:
freq = 1
for _ in range(freq):
self.insert(word)
print(f"Keywords loaded from '{filename}'.")
except FileNotFoundError:
print(f"File '{filename}' not found.")

# Get predictions with support for wildcards '*'
def get_words_with_prefix(self, prefix):
results = []
def _dfs(node, current_prefix, index):
if index == len(prefix):
if node.is_end_of_word:
results.append((current_prefix, node.frequency))
for ch, next_node in node.children.items():
_dfs(next_node, current_prefix + ch, index)
return
ch = prefix[index]
if ch == '*':
for next_ch, next_node in node.children.items():
_dfs(next_node, current_prefix + next_ch, index + 1)
elif ch in node.children:
_dfs(node.children[ch], current_prefix + ch, index + 1)
_dfs(self.root, '', 0)
return results

def find_best_match(self, pattern):
best_match = ("", -1) # (word, frequency)
def _dfs(node, index, path):
nonlocal best_match
if index == len(pattern):
if node.is_end_of_word:
if node.frequency > best_match[1]:
```

```
best_match = (path, node.frequency)
return
ch = pattern[index]
if ch == '*':
for next_ch, child in node.children.items():
_dfs(child, index + 1, path + next_ch)
elif ch in node.children:
_dfs(node.children[ch], index + 1, path + ch)
_dfs(self.root, 0, "")
return best_match[0] if best_match[1] > 0 else None
def separate_words(self, text):
return text.strip().split()

def loop_Sentence(self, array):
result = []
for word in array:
# Separate trailing punctuation except '*'
stripped_word = word.rstrip(string.punctuation.replace('*', ''))
trailing_punct = word[len(stripped_word):]
if '*' in word:
restored_word = self.find_best_match(stripped_word)
if restored_word is not None:
result.append(restored_word + trailing_punct)
else:
result.append(word)
else:
result.append(word)
return ' '.join(result)

def loop_Sentence_AllMatches(self, array):
result = []
for word in array:
stripped_word = word.rstrip(string.punctuation.replace('*', ''))
trailing_punct = word[len(stripped_word):]
stripped_word_clean = stripped_word.rstrip('*')
if '*' in word:
matches = self.get_all_words_with_freq(stripped_word_clean, False)
if matches:
result.append('/'.join(matches) + trailing_punct)
else:
result.append(word)
else:
result.append(word)
return ' '.join(result)
```

trie_editor.py:

```
from trie import Trie
from user_interface import UserInterface
UI = UserInterface()
```

```
# ---------------------- Trie Editor Class ----------------------
# This class handles command-line interactions for the user. Class Created By Aaron
class TrieEditor:
#Done By Aaron
def __init__(self):
self.trie = Trie()

# Validate file name Done By Aaron
def is_valid_filename(self, filename):
invalid_chars = set('<>:"/\\|?*')
return filename and not any(char in invalid_chars for char in filename)
def load_trie_from_folder(self):
path = UI.get_trie_folder_and_file()
if path:
self.trie = Trie() # Reset current trie
self.trie.load_keywords_from_file(path)
print("Trie loaded from selected file.")
else:
print("No file selected.")

# Handle command input parsing Done By Aaron
def get_input(self):
print("> ", end='') # Prompt with "> "
user_input = input().strip()
if not user_input:
return '', '' # return empty cmd and arg
command_parts = user_input.split(maxsplit=1)
cmd = command_parts[0][0] if command_parts[0] else ''
arg = command_parts[0][1:] if len(command_parts[0]) > 1 else ''
if len(command_parts) > 1:
arg += ' ' + command_parts[1]
return cmd, arg

# Graceful exit Done By Aaron
def terminate(self):
print("Exiting the Full Command Prompt . Bye...\n")
print("Press enter key, to continue...")
input()
```

TrieCommandHandler.py

```
from trie import Trie
from user_interface import UserInterface
from TrieVisualiser import TrieVisualizer

UI = UserInterface()

class TrieCommandHandler:
def __init__(self, trie=None):
self.trie = trie if trie else Trie()
self.recent_rounds = []
```

```
self.visualizer = TrieVisualizer(self.trie)

def get_input(self):
try:
raw = input(">> ").strip()
return raw[0], raw[1:].strip() if len(raw) > 1 else ''
except EOFError:
return '\\', ''
except Exception:
return '', ''

def is_valid_filename(self, filename):
import re
return not re.search(r'[\\/:*?"<>|]', filename)

def terminate(self):
print("Exiting program.")
exit()

# Autocomplete game
def _autoComplete_recursive(self, prefix, guesses):
suggestions = self.trie.get_words_with_prefix(prefix)
if not suggestions:
print("No more suggestions. Ending round.")
return guesses
current_guesses = suggestions[:3]
UI.Game_UI(current_guesses)
user_input = input("Is the word one of these? Enter number (1-3), or 'n' for
none: ").strip().lower()
if user_input in ['1', '2', '3']:
chosen_index = int(user_input) - 1
if chosen_index < len(current_guesses):
chosen_word = current_guesses[chosen_index]
print(f"Great! The word is '{chosen_word}'.")
guesses.append(chosen_word)
return guesses
else:
print(f"Invalid input: only {len(current_guesses)} suggestion(s) shown.")
return self._autoComplete_recursive(prefix, guesses)
elif user_input == 'n':
new_prefix = input("Enter more letters to refine your guess (or just press Enter
to stop): ").strip().lower()
if not new_prefix:
print("Stopping round.")
return guesses
return self._autoComplete_recursive(prefix + new_prefix, guesses)
else:
print("Invalid input, try again.")
return self._autoComplete_recursive(prefix, guesses)
def _start_autoComplete_round(self):
```

```
query = input("Enter initial prefix to start autocomplete: ").strip()
if not query:
print("Empty input. Try again.")
return
guesses = self._autoComplete_recursive(query, [])
if guesses:
print("Round completed! Your guesses were:", guesses)
else:
print("No guesses were made this round.")
self.recent_rounds.append((query, guesses))
def _review_recent_rounds(self):
if not self.recent_rounds:
print("No recent rounds to review.")
return
print("Recent Rounds:")
for i, (query, guesses) in enumerate(self.recent_rounds[-5:], 1):
print(f"Round {i}: Query = '{query}'")
for rank, guess in enumerate(guesses, 1):
print(f" {rank}: {guess}")
print("-" * 50)

# Main controller
def command_prompt(self, function, repeat=False):
if function == "construct_edit":
if not repeat:
self.trie = Trie()
UI.construct_edit(show_empty_trie=True)
else:
UI.construct_edit(show_empty_trie=False)
while True:
cmd, arg = self.get_input()
if not cmd:
continue
if cmd == '+':
if arg.isalpha():
self.trie.insert(arg)
print(f"Added '{arg}' to trie.")
elif arg:
print("Invalid input! Only letters allowed.")
else:
print("Please provide a word to add.")
elif cmd == '-':
if arg.isalpha():
if self.trie.search(arg):
self.trie.delete(arg)
print(f"Deleted '{arg}' from trie.")
else:
print("Is not a keyword in trie.")
elif arg:
print("Invalid input! Only letters allowed.")
```

```
else:
print("Please provide a word to delete.")
elif cmd == '?':
if arg.isalpha():
found = self.trie.search(arg)
print(f'Keyword "{arg}" is {"present" if found else "not present"}.')
elif arg:
print("Invalid input! Only letters allowed.")
else:
print("Please provide a word to search.")
elif cmd == '@':
filename = input("Please enter new filename: ").strip()
if filename:
self.trie.save_trie_visual(filename)
print(f"Trie saved to '{filename}'.")
else:
print("No filename entered.")
elif cmd == '~':
filename = input("Please enter input file: ").strip()
if filename:
self.trie = Trie()
self.trie.load_keywords_from_file(filename)
else:
print("No filename entered.")
elif cmd == '=':
if arg and self.is_valid_filename(arg):
try:
self.trie.save_keywords_to_file(arg)
print(f"All keywords written to '{arg}'.")
except OSError:
print(f"Error: Cannot write to '{arg}'.")
else:
filename = input("Please enter new filename: ").strip()
if filename and self.is_valid_filename(filename):
try:
self.trie.save_keywords_to_file(filename)
print(f"All keywords written to '{filename}'.")
except OSError:
print(f"Error: Cannot write to '{filename}'.")
else:
print("Invalid filename.")
elif cmd == '#':
self.trie.display()
elif cmd == '!':
UI.construct_edit(show_empty_trie=False)
continue
elif cmd == '\\':
UI.quit_text()
break
else:
```

```
print("Invalid command! Please try again.")

elif function == "predict_restore":
UI.predict_restore()
while True:
cmd, arg = self.get_input()
if not cmd:
continue
if cmd == '~':
filename = input("Please enter input file: ").strip()
if filename:
self.trie.load_keywords_from_file(filename)
else:
print("No filename entered.")
elif cmd == '#':
self.trie.display()
elif cmd == '$':
arg = input("Please enter input word: ").strip()
print(', '.join(f"{word} ({freq})" for word, freq in
self.trie.get_words_with_prefix(arg)))
elif cmd == '?':
arg = input("Please enter input word: ").strip()
if arg:
result = self.trie.find_best_match(arg)
print(f'Restored word: {result}' if result else "No matching word found.")
else:
print("Please provide a pattern to match.")
elif cmd == '&':
arg = input("Enter sentence: ").strip()
word_array = self.trie.separate_words(arg)
print(self.trie.loop_Sentence_AllMatches(word_array))
elif cmd == '@':
arg = input("Enter sentence: ").strip()
word_array = self.trie.separate_words(arg)
print(self.trie.loop_Sentence(word_array))
elif cmd == '!':
self.command_prompt("predict_restore")
return
elif cmd == '\\':
UI.quit_text()
break
else:
print("Invalid command! Please try again.")

elif function == "trieChart":
UI.getTrieChart()
while True:
cmd = input("Chart Command: ").strip()
if cmd == '~':
filename = input("Please enter input file: ").strip()
```

```
if filename:
self.trie.load_keywords_from_file(filename)
else:
print("No filename entered.")
elif cmd == '^':
longest_word = self.visualizer.get_longest_path()
print(f"Longest path (word): {longest_word}")
self.visualizer.visualize_path(longest_word)
elif cmd == '!':
word = input("Enter word/prefix: ").strip()
if word:
self.visualizer.visualize_subtree_from_prefix(word)
else:
print("No word entered.")
elif cmd == '*':
self.visualizer.visualize_structure()
elif cmd == '\\':
UI.quit_text()
break
else:
print("Invalid command.")

elif function == "autoComplete":
UI.autoCompleteGame()
while True:
cmd = input("Game Command: ").strip()
if cmd == '~':
filename = input("Please enter input file: ").strip()
if filename:
self.trie.load_keywords_from_file(filename)
else:
print("No filename entered.")
elif cmd == '#':
self.trie.display()
elif cmd == '1':
self._start_autoComplete_round()
elif cmd == '2':
self._review_recent_rounds()
elif cmd == '\\':
UI.quit_text()
break
else:
print("Invalid Command.")
```

TrieIO.py

```
class TrieIO:
def __init__(self, trie):
self.trie = trie

# Save all keywords with frequencies to a file
```

```
def save_keywords_to_file(self, filename):
words_with_freq = self.trie.get_all_words_with_freq()
with open(filename, 'w') as f:
for word, freq in words_with_freq:
f.write(f'{word},{freq}\n')

# Load keywords from file and populate the trie
def load_keywords_from_file(self, filename):
try:
with open(filename, 'r') as f:
for line in f:
line = line.strip()
if line:
if ',' in line:
word, freq_str = line.rsplit(',', 1)
try:
freq = int(freq_str)
except ValueError:
freq = 1
for _ in range(freq):
self.trie.insert(word)
print(f"Keywords loaded from '{filename}'.")
except FileNotFoundError:
print(f"File '{filename}' not found.")

# Save visual representation of the trie to a file (as indented ASCII-like
structure)
def save_trie_visual(self, filename):
def _display_node(node, prefix='', depth=0):
lines = []
indent = '.' * depth
# Opening bracket with prefix
lines.append(f"{indent}[{prefix}")
# If it's a word, print it with frequency
if node.is_end_of_word:
lines.append(f"{indent}{'.' * 1}>{prefix}({node.frequency})*")
# Recurse into children
for ch, child in sorted(node.children.items()):
lines.extend(_display_node(child, prefix + ch, depth + 1))
# Closing bracket
lines.append(f"{indent}]")
return lines
lines = _display_node(self.trie.root)
with open(filename, 'w') as f:
for line in lines:
f.write(line + '\n')
```

TrieVisualiser.py

```
import matplotlib.pyplot as plt
import networkx as nx
```

```
import scipy
import matplotlib.patches as mpatches

class TrieVisualizer:
def __init__(self, trie):
self.trie = trie

def visualize_structure(self):
print("Generating trie structure visualization (top-down with word buildup)...")
G = nx.DiGraph()
node_id = 0
node_map = {}
positions = {}
node_colors = []
def dfs(node, path, depth, x_offset):
nonlocal node_id
current_id = node_id
# Node label is the progressive word
label = path if path else "ROOT"
G.add_node(current_id, label=label)
node_map[id(node)] = current_id
# Assign position (top-down)
positions[current_id] = (x_offset[0], -depth)
# Color: green if complete word, else blue
if node.is_end_of_word:
node_colors.append('lightgreen')
else:
node_colors.append('lightblue')
node_id += 1
for ch, child in sorted(node.children.items()):
x_offset[0] += 1
child_id = dfs(child, path + ch, depth + 1, x_offset)
G.add_edge(current_id, child_id)
return current_id
dfs(self.trie.root, "", 0, [0])
labels = nx.get_node_attributes(G, 'label')
plt.figure(figsize=(14, 8))
nx.draw(G, pos=positions, with_labels=True, labels=labels,
node_color=node_colors, node_size=1200, font_size=10, arrows=True)
plt.legend(handles=[
mpatches.Patch(color='lightblue', label='Prefix'),
mpatches.Patch(color='lightgreen', label='Complete Word')
])
plt.title("Trie Structure (Progressive Word Formation)")
plt.axis('off')
plt.tight_layout()
plt.show()

def visualize_path(self, word):
print(f"Visualizing path for '{word}'...")
```

```
G = nx.DiGraph()
node = self.trie.root
current = ""
positions = {}
node_colors = []
G.add_node(current)
positions[current] = (0, 0)
node_colors.append('lightblue') # root
for i, char in enumerate(word):
next_label = current + char
G.add_node(next_label)
G.add_edge(current, next_label)
positions[next_label] = (i + 1, -i - 1)
if char in node.children:
node = node.children[char]
# If this node ends a word, color it green
if node.is_end_of_word and i == len(word) - 1:
node_colors.append('lightgreen')
else:
node_colors.append('lightblue')
else:
print(f"'{word}' not found in trie.")
return
current = next_label
plt.figure(figsize=(8, 5))
nx.draw(G, pos=positions, with_labels=True, node_color=node_colors,
node_size=1000, font_size=10, arrows=True)
plt.title(f"Path for '{word}'")
plt.axis('off')
plt.tight_layout()
plt.show()

def get_longest_path(self):
def dfs(node, path):
nonlocal longest_path
if node.is_end_of_word and len(path) > len(longest_path):
longest_path = path[:]
for char, child in node.children.items():
path.append(char)
dfs(child, path)
path.pop()
longest_path = []
dfs(self.trie.root, [])
return ''.join(longest_path)
```

AutoCompleteGame.py

```
class AutoCompleteGame:
def __init__(self, trie, ui_module):
self.trie = trie
self.UI = ui_module # expects UI.Game_UI() method
```

```
self.recent_rounds = []

def _autoComplete_recursive(self, prefix, guesses):
suggestions = self.trie.get_words_with_prefix(prefix)
if not suggestions:
print("No more suggestions. Ending round.")
return guesses
# Take up to 3 suggestions to show
current_guesses = suggestions[:3]
self.UI.Game_UI(current_guesses)
# Ask user if any guess is correct
user_input = input("Is the word one of these? Enter number (1-3), or 'n' for
none: ").strip().lower()
if user_input in ['1', '2', '3']:
chosen_index = int(user_input) - 1
if chosen_index < len(current_guesses):
chosen_word = current_guesses[chosen_index]
print(f"Great! The word is '{chosen_word}'.")
guesses.append(chosen_word)
return guesses
else:
print(f"Invalid input: only {len(current_guesses)} suggestion(s) shown.")
return self._autoComplete_recursive(prefix, guesses)
elif user_input == 'n':
# User says none matched, ask for next prefix to narrow down
new_prefix = input("Enter more letters to refine your guess (or just press Enter
to stop): ").strip().lower()
if not new_prefix:
print("Stopping round.")
return guesses
return self._autoComplete_recursive(prefix + new_prefix, guesses)
else:
print("Invalid input, try again.")
return self._autoComplete_recursive(prefix, guesses)

def _start_autoComplete_round(self):
query = input("Enter initial prefix to start autocomplete: ").strip()
if not query:
print("Empty input. Try again.")
return
guesses = self._autoComplete_recursive(query, [])
if guesses:
print("Round completed! Your guesses were:", guesses)
else:
print("No guesses were made this round.")
self.recent_rounds.append((query, guesses))

def _review_recent_rounds(self):
if not self.recent_rounds:
print("No recent rounds to review.")
```

```
return
print("Recent Rounds:")
for i, (query, guesses) in enumerate(self.recent_rounds[-5:], 1):
print(f"Round {i}: Query = '{query}'")
for rank, guess in enumerate(guesses, 1):
print(f" {rank}: {guess}")
print("-" * 50)
def run(self):
print("== Autocomplete Game ==")
while True:
print("\nMenu:")
print("1. Start new round")
print("2. Review recent rounds")
print("3. Exit")
choice = input("Choose an option: ").strip()
if choice == '1':
self._start_autoComplete_round()
elif choice == '2':
self._review_recent_rounds()
elif choice == '3':
print("Goodbye!")
break
else:
print("Invalid option.")
```

feature_base.py:

```
# ----------------------------------------
# feature_base.py
# Base class for all extra features
# Done By Aaron
# ----------------------------------------
class FeatureBase:
def __init__(self, trie):
# Store a reference to the Trie instance
self.trie = trie
def run(self):
# Subclasses must override this method
raise NotImplementedError("Subclasses must implement the 'run' method.")
```

feature_advanced_editor.py:

```
# ----------------------------------------
# feature_advanced_editor.py
# Advanced Trie Tools Implementation
# Done By Aaron
# ----------------------------------------

from feature_base import FeatureBase
from user_interface import UserInterface
import os
```

```
class AdvancedTrieFeature(FeatureBase):
def __init__(self, trie_instance):
super().__init__(trie_instance)
self.trie = trie_instance
self.trie_class = trie_instance.__class__

def command_prompt(self):
# Display feature banner and command instructions

ui = UserInterface()
ui.display_advanced_trie_tools() # Reuse centralized instructions

# Main loop to handle user input commands
while True:
print("[Feature 5] > ", end='')
user_input = input().strip()
if not user_input:
continue
command = user_input[0]
args = user_input[1:].strip()
if command == '~':
self.load_and_merge_files(args)
elif command == '>':
self.display_top(args)
elif command == '+':
self.increment_keyword()
elif command == '-':
self.decrement_keyword()
elif command == '^':
self.replace_word(args)
elif command == '!':
self.command_prompt()
return
elif command == '\\':
print("Exiting Feature 5: Advanced Trie Tools. Returning to main menu...")
break
else:
print("Invalid command. Use ! to see instructions.")

def run(self):
# Run this feature through the command prompt
self.command_prompt()

def load_and_merge_files(self, arg):
# Parse filenames and check format
if ',' not in arg:
print("Invalid format. Use: ~file1.txt,file2.txt")
return
file1, file2 = map(str.strip, arg.split(',', 1))
# Check file existence
```

```
if not os.path.exists(file1) or not os.path.exists(file2):
print(f"One or both files '{file1}', '{file2}' do not exist.")
return
print(f"Merging tries from '{file1}' and '{file2}'...")
# Reset and load both Tries
self.trie = self.trie_class() # Reset current trie
self.trie.load_keywords_from_file(file1)
self.trie.load_keywords_from_file(file2)
print("Merge complete. Displaying merged trie:")
self.trie.display()
# Ask to save merged Trie
print("\nDo you want to save the merged Trie to a new TXT file? (yes/no): ",
end='')
save = input().strip().lower()
if save == 'yes':
print("Enter filename to save the merged Trie (e.g., merged.txt): ", end='')
save_filename = input().strip()
if save_filename == '':
print("Invalid filename. Merged Trie not saved.")
return
self.trie.save_keywords_to_file(save_filename)
print(f"Merged Trie has been saved to '{save_filename}'.")
else:
print("Merged Trie was not saved.")

def display_top(self, filename):
# Load file and build a temporary Trie
if not os.path.exists(filename):
print(f"File '{filename}' not found.")
return
temp_trie = self.trie_class()
temp_trie.load_keywords_from_file(filename)
all_words = temp_trie.get_all_words_with_freq()
if not all_words:
print(f"No keywords found in '{filename}'.")
return
# Ask user for number of top keywords
while True:
print("Enter how many top keywords to display (e.g., 5): ", end='')
try:
top_n = int(input().strip())
if top_n <= 0:
print("Please enter a positive integer.")
continue
break
except ValueError:
print("Invalid number. Please enter a valid integer.")
# Sort by frequency descending
top_n_keywords = sorted(all_words, key=lambda x: x[1], reverse=True)[:top_n]
print(f"\nTop {top_n} keywords by frequency:")
```

```
for word, freq in top_n_keywords:
print(f" - {word}: {freq}")

def increment_keyword(self):
# Add new keyword(s) to the file and Trie
print("Enter filename to load the Trie from: ", end='')
filename = input().strip()
if not filename or not os.path.exists(filename):
print(f"File '{filename}' not found.")
return
self.trie = self.trie_class()
self.trie.load_keywords_from_file(filename)
while True:
print("Enter +keyword to add (e.g., +cat): ", end='')
user_input = input().strip()
if not user_input.startswith('+') or len(user_input) <= 1:
print("Invalid format. Use: +keyword")
continue
keyword = user_input[1:].strip()
if not keyword:
print("Keyword cannot be empty.")
continue
self.trie.insert(keyword)
print(f"'{keyword}' has been added to the Trie (frequency increased by 1).")
print("\nUpdated Trie structure:")
self.trie.display()
print("\nDo you want to add another keyword? (yes/no): ", end='')
again = input().strip().lower()
if again != 'yes':
break
# Ask to save
print("\nDo you want to update and save the TXT file? (yes/no): ", end='')
save_choice = input().strip().lower()
if save_choice == 'yes':
self.trie.save_keywords_to_file(filename)
print(f"Trie has been saved to '{filename}'.")
else:
print("Changes were not saved.")

def decrement_keyword(self):
# Remove keyword(s) from the file and Trie
print("Enter filename to load the Trie from: ", end='')
filename = input().strip()
if not filename or not os.path.exists(filename):
print(f"File '{filename}' not found.")
return
self.trie = self.trie_class()
self.trie.load_keywords_from_file(filename)
while True:
print("Enter -keyword to subtract (e.g., -cat): ", end='')
```

```
user_input = input().strip()
if not user_input.startswith('-') or len(user_input) <= 1:
print("Invalid format. Use: -keyword")
continue
keyword = user_input[1:].strip()
if not keyword:
print("Keyword cannot be empty.")
continue
all_words = dict(self.trie.get_all_words_with_freq())
if keyword not in all_words:
print(f"'{keyword}' not found in the Trie.")
continue
self.trie.delete(keyword)
print(f"One occurrence of '{keyword}' has been removed from the Trie.")
print("\nUpdated Trie structure:")
self.trie.display()
print("\nDo you want to subtract another keyword? (yes/no): ", end='')
again = input().strip().lower()
if again != 'yes':
break
# Ask to save
print("\nDo you want to update and save the TXT file? (yes/no): ", end='')
save_choice = input().strip().lower()
if save_choice == 'yes':
self.trie.save_keywords_to_file(filename)
print(f"Trie has been saved to '{filename}'.")
else:
print("Changes were not saved.")

def replace_word(self, args):
# Replace old keyword with new one in the file
print("Enter filename to load the Trie from: ", end='')
filename = input().strip()
if not filename or not os.path.exists(filename):
print(f"File '{filename}' not found.")
return
self.trie = self.trie_class()
self.trie.load_keywords_from_file(filename)
while True:
# Step 1: Show current words
print("\nCurrent keywords with frequencies:")
all_words = self.trie.get_all_words_with_freq()
for word, freq in sorted(all_words):
print(f"{word},{freq}")
print("\nEnter ^old,new to replace a word: ", end='')
replace_input = input().strip()
if not replace_input.startswith('^') or ',' not in replace_input:
print("Invalid format. Use: ^oldword,newword")
return
old, new = map(str.strip, replace_input[1:].split(',', 1))
```

```
# Step 2: Find and replace word in the list
new_word_list = []
replaced = False
freq_to_add = 0
for word, freq in all_words:
if word == old:
freq_to_add = freq
replaced = True
else:
new_word_list.append((word, freq))
if not replaced:
print(f"'{old}' not found in current Trie.")
continue
new_word_list.append((new, freq_to_add))
# Step 3: Rebuild Trie
self.trie = self.trie_class()
for word, freq in new_word_list:
for _ in range(freq):
self.trie.insert(word)
print(f"\n'{old}' has been replaced with '{new}' (with frequency {freq_to_add}).")
# Step 4: Show updated Trie
print("\nUpdated Trie structure:")
self.trie.display()
# Step 5: Ask if want to replace more
print("\nDo you want to replace another keyword? (yes/no): ", end='')
again = input().strip().lower()
if again != 'yes':
break
# Step 6: Ask to save
print("\nDo you want to update and save the TXT file? (yes/no): ", end='')
save_choice = input().strip().lower()
if save_choice == 'yes':
self.trie.save_keywords_to_file(filename)
print(f"Trie has been saved to '{filename}'.")
else:
print("Changes were not saved.")
```

keyword_analysis_feature.py:

```
# ----------------------------------------
# keyword_analysis_feature.py
# Keyword Analysis Feature Implementation
# Done By Aaron
# ----------------------------------------

from feature_base import FeatureBase
from user_interface import UserInterface
import os

class KeywordAnalysisFeature(FeatureBase):
# Initialize with two Trie instances for comparison or transfer
```

```
def __init__(self, trie_instance):
self.trie1 = trie_instance
self.trie2 = trie_instance.__class__() # Or create another one as needed
self.trie_class = trie_instance.__class__

# Entry point for the feature
def run(self):
self.command_prompt()

# Print available commands for the feature
def print_instructions(self):
ui = UserInterface()
ui.display_keyword_analysis_feature() # Reuse centralized instructions

# Main loop that takes user input and triggers appropriate methods
def command_prompt(self):
self.print_instructions()
while True:
print("[Feature 6] > ", end='')
user_input = input().strip()
if not user_input:
continue
command = user_input[0]
args = user_input[1:].strip()
if command == '=':
self.compare_keywords(args)
elif command == '>':
self.transfer_keyword(args)
elif command == '#':
self.sort_keywords_by_length(args)
elif command == '*':
self.group_by_alphabet(args)
elif command == '%':
self.top_starting_letters(args)
elif command == '$':
self.find_palindromes(args)
elif command == '!':
self.print_instructions()
elif command == '\\':
print("Exiting Extra Feature Two: Keyword Tools.")
break
else:
print("Invalid command. Use ! to see instructions.")
# Compare common keywords between two files

def compare_keywords(self, arg):
if ',' not in arg:
print("Invalid format. Use: =file1.txt,file2.txt")
return
file1, file2 = map(str.strip, arg.split(',', 1))
```

```
if not os.path.exists(file1) or not os.path.exists(file2):
print(f"One or both files '{file1}', '{file2}' do not exist.")
return
# Load both tries
trie1 = self.trie_class()
trie1.load_keywords_from_file(file1)
words1 = set(w for w, _ in trie1.get_all_words_with_freq())
trie2 = self.trie_class()
trie2.load_keywords_from_file(file2)
words2 = set(w for w, _ in trie2.get_all_words_with_freq())
# Find common keywords
common = words1.intersection(words2)
if not common:
print("No common keywords found.")
else:
print("Common keywords:")
for word in sorted(common):
print(f" - {word}")
# Transfer a keyword from one file to another

def transfer_keyword(self, arg):
if ',' not in arg:
print("Invalid format. Use: >from.txt,to.txt")
return
file1, file2 = map(str.strip, arg.split(',', 1))
if not os.path.exists(file1) or not os.path.exists(file2):
print(f"One or both files '{file1}', '{file2}' do not exist.")
return
self.trie1 = self.trie_class()
self.trie2 = self.trie_class()
self.trie1.load_keywords_from_file(file1)
self.trie2.load_keywords_from_file(file2)
while True:
print("Enter keyword to transfer from first file to second: ", end='')
word = input().strip()
all_words = dict(self.trie1.get_all_words_with_freq())
if word not in all_words:
print(f"'{word}' not found in '{file1}'.")
continue
# Transfer one instance of the keyword
self.trie1.delete(word)
self.trie2.insert(word)
print(f"'{word}' transferred from '{file1}' to '{file2}'.")
print("\nUpdated Trie for first file:")
self.trie1.display()
print("\nUpdated Trie for second file:")
self.trie2.display()
print("\nDo you want to transfer another keyword? (yes/no): ", end='')
again = input().strip().lower()
if again != 'yes':
```

```
break
print("\nDo you want to save updates to both TXT files? (yes/no): ", end='')
save = input().strip().lower()
if save == 'yes':
self.trie1.save_keywords_to_file(file1)
self.trie2.save_keywords_to_file(file2)
print("Changes saved.")
else:
print("Changes were not saved.")
# Sort and display keywords from longest to shortest

def sort_keywords_by_length(self, filename):
if not os.path.exists(filename):
print(f"File '{filename}' not found.")
return
temp_trie = self.trie_class()
temp_trie.load_keywords_from_file(filename)
all_words = temp_trie.get_all_words_with_freq()
sorted_words = sorted(all_words, key=lambda x: (-len(x[0]), x[0]))
print("\nKeywords from longest to shortest:")
for word, freq in sorted_words:
print(f" - {word} ({freq})")
# Group and display keywords alphabetically

def group_by_alphabet(self, filename):
if not os.path.exists(filename):
print(f"File '{filename}' not found.")
return
temp_trie = self.trie_class()
temp_trie.load_keywords_from_file(filename)
all_words = temp_trie.get_all_words_with_freq()
grouped = {}
for word, freq in all_words:
first_letter = word[0].upper()
if not first_letter.isalpha():
first_letter = '#' # for non-alphabet characters
grouped.setdefault(first_letter, []).append((word, freq))
print(f"\nGrouped keywords in '{filename}':")
for letter in sorted(grouped.keys()):
print(f"\n{letter}:")
for word, freq in sorted(grouped[letter]):
print(f" - {word} ({freq})")
# Show the top starting letters by frequency

def top_starting_letters(self, filename):
if not os.path.exists(filename):
print(f"File '{filename}' not found.")
return
temp_trie = self.trie_class()
temp_trie.load_keywords_from_file(filename)
```

```
all_words = temp_trie.get_all_words_with_freq()
letter_counts = {}
for word, _ in all_words:
first_letter = word[0].upper()
if not first_letter.isalpha():
first_letter = '#'
letter_counts[first_letter] = letter_counts.get(first_letter, 0) + 1
sorted_letters = sorted(letter_counts.items(), key=lambda x: x[1],
reverse=True)[:5]
print(f"\nTop starting letters in '{filename}':")
for letter, count in sorted_letters:
print(f" - {letter}: {count} words")
# Display palindromic keywords from file

def find_palindromes(self, filename):
if not os.path.exists(filename):
print(f"File '{filename}' not found.")
return
temp_trie = self.trie_class()
temp_trie.load_keywords_from_file(filename)
all_words = temp_trie.get_all_words_with_freq()
palindromes = [(word, freq) for word, freq in all_words if word == word[::-1] and
len(word) > 1]
if not palindromes:
print(f"No palindromic keywords found in '{filename}'.")
else:
print(f"\nPalindromic keywords in '{filename}':")
for word, freq in sorted(palindromes):
print(f" - {word} ({freq})")
```