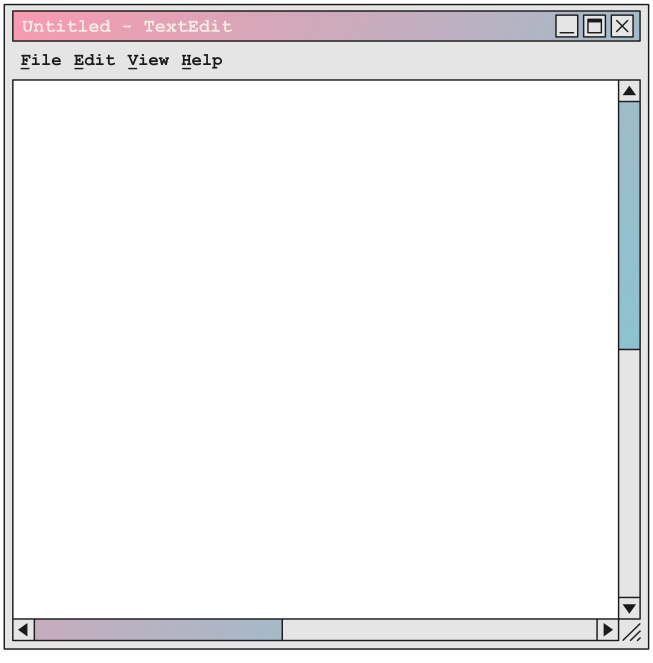
Text Editor Mini project

Report-01

An Overview

Submitted by
MERWIN PINTO
SRN 202100102
ROLL NO 1



Problem Statement :

Design and implement a basic text editor with fundamental features, such as text entry, editing, saving, and opening files. Include support for features like cut, copy, paste, undo, and redo. Develop a simple user interface for your editor and provide documentation on how to use these basic features effectively.

Abstract :

This project centres on the design and implementation of a fundamental text editor, a versatile tool with broad applications in various domains. The primary objective is to create an intuitive and user-friendly interface that empowers users with essential functionalities, including text entry, editing, saving, opening files, and support for actions like cut, copy, paste, undo, and redo.

Key concepts crucial to the project's theoretical foundation encompass Graphical User Interface (GUI), widgets, the text widget as the core text-editing component, the clipboard for data transfer, the mechanism for undo and redo operations, and file handling capabilities for opening and saving text files.

The practical realization involves developing a Python-based text editor using the Tkinter library, a well-established framework for GUI applications. The project's code comprises a class-based structure, employing methods for performing various functions, such as clipboard operations and maintaining a history of text states for enabling undo and redo actions.

In conclusion, this project sets the stage for constructing a functional text editor by elucidating the critical theoretical principles and concepts underlying its design and implementation. It forms the basis for the creation of a user-friendly text editing tool, which, with further enhancements and refinements, can cater to a wide range of user requirements.

Introduction:

Text editors are everyday tools that empower users to create, edit, and organize textual content. They play a vital role in various aspects of computing, including writing, coding, and note-taking. This project focuses on building a straightforward text editor with key features that make text-related tasks more manageable. These features include typing and editing text, working with files, and performing actions like cutting, copying, pasting, undoing, and redoing.

The project's core idea is to create a user-friendly application that simplifies text-related tasks, making it accessible to people in different fields. To achieve this, we need to

understand some essential concepts that underlie text editors. These concepts include designing the visual interface (the part you see on your screen), using interactive elements (like buttons), having a central area for text input and manipulation, enabling easy copying and pasting of text, allowing users to reverse or redo their actions, and letting users work with files conveniently.

To turn these ideas into a working text editor, we'll use a Python library called Tkinter, which is known for creating user interfaces. We'll build our text editor using Python's capabilities and Tkinter's tools.

The heart of this project is a Python class, which is like a blueprint for our text editor. It's organized into different functions, each serving a specific purpose. These functions help us perform actions like copying text and keeping a record of what we've done.

Key Concepts :

1. **Graphical User Interface (GUI):**
   a. A graphical user interface is the visual component of an application that allows users to interact with the software. It includes windows, buttons, menus, and other graphical elements.
   b. In our text editor project, a GUI is created using the Tkinter library. This GUI provides the user interface for text entry and editing.
2. **Widgets:**
   a. Widgets are GUI components like buttons, text boxes, and labels that enable user interaction. They are the building blocks of a graphical interface.
   b. In the text editor, widgets like buttons are used to provide functionality for actions such as cut, copy, paste, and undo/redo.
3. **Text Widget:**
   a. The text widget is a core component of the text editor, providing an area for users to enter and edit text.
   b. It supports various text-related features, such as text selection, copying, pasting, and undoing and redoing changes.
4. **Clipboard:**
   a. The clipboard is a temporary storage area for data that the user wants to copy from one location and paste into another.
   b. In the text editor project, the clipboard is used to implement copy, cut, and paste functionality. It temporarily holds text that users select and manipulate.
5. **Undo and Redo:**
   a. Undo and redo functionalities allow users to reverse or reapply their previous actions.
   b. In the text editor, maintaining a history of text states is essential. The history keeps track of changes, enabling users to undo and redo text modifications.
6. **File Operations:**
   a. File operations refer to actions that involve opening and saving text files.
   b. In the text editor project, the file dialogue module is used to create file dialogues, allowing users to open existing text files and save their edited content to new or existing files.

Project Essentials :

This section provides an overview of the text editor project, offering insights into the fundamental aspects that have influenced its development. It begins by introducing the choice of the Python programming language and its aptness for the task. Subsequently, it delves into the requirement analysis and design phases, establishing the groundwork for a comprehensive grasp of the project's evolution and objectives.

Language Used :

Introduction to the Language:

In the development of our text editor, we opted for the Python programming language. Python is known for its simplicity, readability, and versatility, making it a fitting choice for our project. It's widely recognized as one of the most user-friendly languages, ideal for both beginners and experienced developers. Its well-designed syntax and extensive standard libraries streamline the development process.

Python is versatile in nature, being equally adept at catering to the needs of novice programmers and seasoned developers. Its uncomplicated syntax and minimalistic, human-readable code structure make it an ideal choice for creating applications where simplicity and clarity are paramount. In the context of a text editor, where users expect an intuitive and straightforward interface, Python's characteristics make it an apt selection.

Why Python?:

The choice of Python can be attributed to several factors:

- **Readability:** Python's clean and uncluttered syntax enhances code readability, which was especially beneficial for this text editor project where clarity and simplicity are crucial.
- **Extensive Libraries:** Python boasts a vast collection of libraries and modules. For our graphical user interface (GUI), we harnessed the power of Tkinter, a standard GUI library that simplifies the creation of user-friendly interfaces.
- **Cross-Platform Compatibility:** Python's cross-platform compatibility ensures that our text editor can be used on a variety of operating systems, from Windows to macOS and Linux.
- **Community and Documentation:** Python's large and active community provides access to abundant resources, tutorials, forums, and documentation, facilitating the development process and problem-solving.

Library (Tkinter):

Tkinter, the library chosen for GUI development, is a fundamental part of Python's standard library. Its intuitive and straightforward nature aligns perfectly with the requirements of our project, making it an excellent choice for creating the graphical interface of our text editor. Additionally, Tkinter's compatibility with various platforms ensures that our text editor is accessible to a broad user base.

Simplicity and Readability:

The simplicity and readability of Python were particularly advantageous during the development of our text editor. The concise code allowed us to focus on the implementation of essential features without getting bogged down by complex syntax or convoluted structures. This was paramount in ensuring that our text editor remains user-friendly and comprehensible.

Execution Plan :

Requirement Analysis :

The requirement analysis phase marks the inception of the project, serving as the critical groundwork for designing and developing the text editor. During this phase, the primary objective was to identify and define the essential functionalities and features that the text editor needed to encompass. Requirement analysis encompassed several key components:

Functional Requirements:

1. **Text Entry and Editing:** The fundamental functionality of a text editor is to enable users to input and edit text. This requirement included the ability to create, modify, and delete text within the editor.
2. **Cut, Copy, and Paste Operations:** Basic text manipulation features such as cutting, copying, and pasting text were identified as crucial. Users needed the capability to select and transfer text within the editor seamlessly.
3. **Undo and Redo Functionality:** The requirement for undo and redo functionality was identified to empower users to reverse or reapply their text modifications. This feature was essential for enhancing the editing experience and correcting inadvertent changes.
4. **File Operations (Open and Save):** Recognizing the importance of file management, the text editor was expected to support file operations. This included the ability to open existing text files for editing and save the editor's content to both new and existing files.

User Interface Requirements:

1. **Simplicity and Intuitiveness:** The user interface was required to be simple and intuitive, ensuring that users could easily grasp the editor's functionalities without confusion or a steep learning curve.
2. **Clarity in Design:** The user interface design needed to maintain a clean and clear layout, facilitating efficient text manipulation and user interaction.

Design Phase :

The design phase is a critical component of the project, representing the bridge between the initial requirement analysis and the practical implementation of the text editor. During this phase, we translated the identified requirements into a comprehensive and practical plan for the text editor. The design phase encompassed the following key aspects:

Graphical User Interface (GUI) :

1. **Choosing Tkinter:** A crucial decision made during the design phase was the selection of the graphical user interface (GUI) library. Tkinter, a Python library that is a part of the standard library, emerged as the natural choice. Tkinter is known for its simplicity, making it accessible to developers and ensuring that the user interface remains straightforward and easy to use.
2. **GUI Layout and Design :**

The design phase entailed planning the layout and visual design of the graphical interface. The GUI was structured to be clean, uncluttered, and intuitive. It featured a central text editing area, menu buttons for actions like "Cut," "Copy," "Paste," "Undo," "Redo," "Open," and "Save," and a user-friendly toolbar.

Implementation :

The implementation phase transformed the text editor project from a conceptual plan into a fully functional application. It covers the coding of core features, the integration of clipboard operations, the addition of undo and redo actions and support for file operations. Extensive testing and debugging ensured the editor's reliability and efficiency. The inclusion of keyboard shortcuts further enhances the user experience.

During the implementation phase, we translated the project design into a functional text editor. This phase involved a series of significant steps:

1. Core Functionality:

At this stage, the focus was on implementing the core functionalities of the text editor, which encompassed text entry and editing. These features allowed users to input and modify text within the editor's interface.

2. Text Manipulation:

Another essential element was the integration of cut, copy, and paste operations. These actions provided users with the capability to select, duplicate, and move text seamlessly within the editor, significantly enhancing text manipulation capabilities.

3. Undo and Redo Actions:

To enhance the overall user experience, undo and redo functionalities were integrated. These features enabled users to reverse or reapply their text modifications, a valuable asset for correcting errors and managing changes effectively.

4. File Operations:

Support for file operations, including opening and saving files, was also part of the implementation phase. To facilitate file opening, the 'filedialog' module was utilized to provide users with a file selection dialog. For saving files, users could specify the name and location of the file, with the editor's content being written to the chosen file.

5. Clipboard Functionality:

Incorporating clipboard functionality was a crucial aspect of implementation. This functionality allowed users to cut or copy selected text to the clipboard and paste it at the cursor's position within the editor.

6. Testing and Debugging:

The implementation phase was followed by comprehensive testing to ensure that all features operated as intended. Functionality testing assessed the behaviour of each feature, while user testing evaluated the editor's user-friendliness. Issues identified during testing were resolved through rigorous debugging and code optimization, guaranteeing the reliability and efficiency of the text editor.

7. Keyboard Shortcuts:

To further enhance user productivity, the implementation phase included the integration of keyboard shortcuts. For instance, Ctrl+X was linked to the 'cut' function, Ctrl+C to 'copy,' Ctrl+V to 'paste,' and Ctrl+Z to 'undo,' enabling users to perform these actions quickly using keyboard commands.

Conclusion :

To conclude, this report has offered a comprehensive insight into the design and implementation of a fundamental text editor. The primary objective was to create an intuitive and user-friendly interface equipped with essential text editing capabilities, including text entry, editing, saving, and file management, along with features like cut, copy, paste, undo, and redo.

The implementation phase played a pivotal role in translating the design plan into a fully functional text editor. It encompassed the coding of core features, integration of clipboard operations, the addition of undo and redo functionality and support for file operations.

By selecting Python and Tkinter, powerful tools were harnessed to create a user-friendly text editor. Python's simplicity and readability made it an ideal choice for this project, while Tkinter provided a straightforward GUI framework. The design phase meticulously established the layout and visual design of the graphical interface, focusing on clarity and intuitiveness.

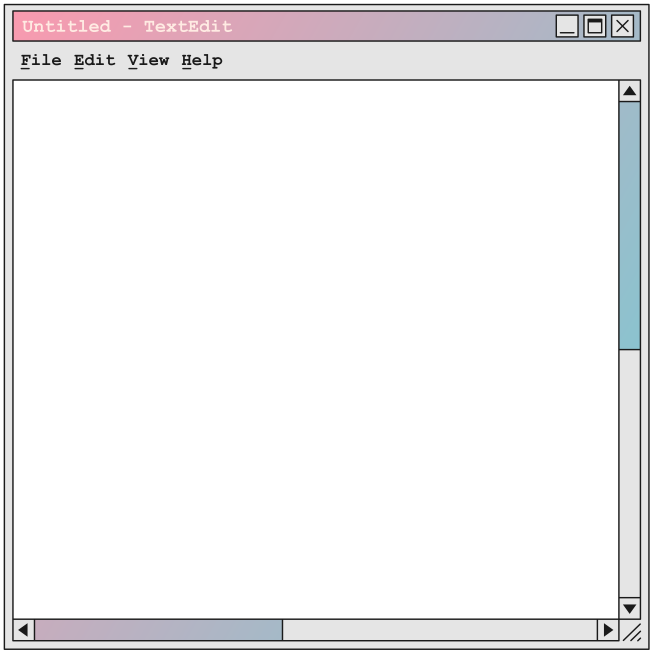
In closing, this report lays the groundwork for building a practical and easy-to-use text editor by explaining the crucial theoretical ideas and concepts behind it.

Text Editor Mini project

Report-02

Implementation and Output

Submitted by
MERWIN PINTO SRN 202100102
ROLL NO 1



Problem Statement:

Design and implement a basic text editor with fundamental features, such as text entry, editing, saving, and opening files. Include support for features like cut, copy, paste, undo, and redo. Develop a simple user interface for your editor and provide documentation on how to use these basic features effectively.

Language used for Implementation:

The language chosen for implementing the text editor is ‘Python’ because of its extensive libraries and user friendliness. Python provided all in one solution for creation of GUI and also implement the functionalities as per the requirements. For implementing the GUI python library ‘Tkinter’ was used.

Python Program:

```
import tkinter as tk

from tkinter import filedialog

class TextEditorApp:

  def __init__(self):

    self.root = tk.Tk()

    self.root.title("Text Editor")


    # Create a toolbar

    toolbar = tk.Frame(self.root)

    toolbar.pack(side="top", fill="x")


    # Create toolbar buttons

    cut_button = tk.Button(toolbar, text="Cut", command=self.cut_text)

    cut_button.pack(side="left")

    copy_button = tk.Button(toolbar, text="Copy", command=self.copy_text)

    copy_button.pack(side="left")
```

```
paste_button = tk.Button(toolbar, text="Paste", command=self.paste_text)
paste_button.pack(side="left")
undo_button = tk.Button(toolbar, text="Undo", command=self.undo_text)
undo_button.pack(side="left")
redo_button = tk.Button(toolbar, text="Redo", command=self.redo_text)
redo_button.pack(side="left")
open_button = tk.Button(toolbar, text="Open", command=self.open_file)
open_button.pack(side="left")
save_button = tk.Button(toolbar, text="Save", command=self.save_file)
save_button.pack(side="left")

# Create a text widget for the main editing area
self.text_widget = tk.Text(self.root)
self.text_widget.pack(expand="yes", fill="both")

# Maintain a history of text changes for undo
self.history = []
self.history_index = -1

# Bind keyboard shortcuts
self.text_widget.bind("<Key>", self.on_text_change)
self.text_widget.bind("<Control-x>", lambda e: self.cut_text())
self.text_widget.bind("<Control-c>", lambda e: self.copy_text())
self.text_widget.bind("<Control-v>", lambda e: self.paste_text())
self.text_widget.bind("<Control-z>", lambda e: self.undo_text())
```

```
def on_text_change(self, event):
    self.record_change()

def cut_text(self):
    selected_text = self.text_widget.get("sel.first", "sel.last")
    self.copy_to_clipboard(selected_text)
    self.text_widget.delete("sel.first", "sel.last")
    self.record_change()

def copy_text(self):
    selected_text = self.text_widget.get("sel.first", "sel.last")
    self.copy_to_clipboard(selected_text)

def paste_text(self):
    clipboard_text = self.get_clipboard_text()
    self.text_widget.insert("insert", clipboard_text)
    self.record_change()

def undo_text(self):
    if self.history_index > 0:
        self.history_index -= 1
        self.text_widget.delete("1.0", "end")
        self.text_widget.insert("1.0", self.history[self.history_index])

def redo_text(self):
    if self.history_index < len(self.history) - 1:
```

```
        self.history_index += 1
        self.text_widget.delete("1.0", "end")
        self.text_widget.insert("1.0", self.history[self.history_index])

    def copy_to_clipboard(self, text):
        self.root.clipboard_clear()
        self.root.clipboard_append(text)

    def get_clipboard_text(self):
        return self.root.clipboard_get()

    def record_change(self):
        text = self.text_widget.get("1.0", "end")
        if self.history_index < len(self.history) - 1:
            self.history = self.history[:self.history_index + 1]
        self.history.append(text)
        self.history_index = len(self.history) - 1

    def open_file(self):
        file_path = filedialog.askopenfilename()
        if file_path:
            with open(file_path, 'r') as file:
                self.text_widget.delete("1.0", "end")
                self.text_widget.insert("1.0", file.read())

    def save_file(self):
```

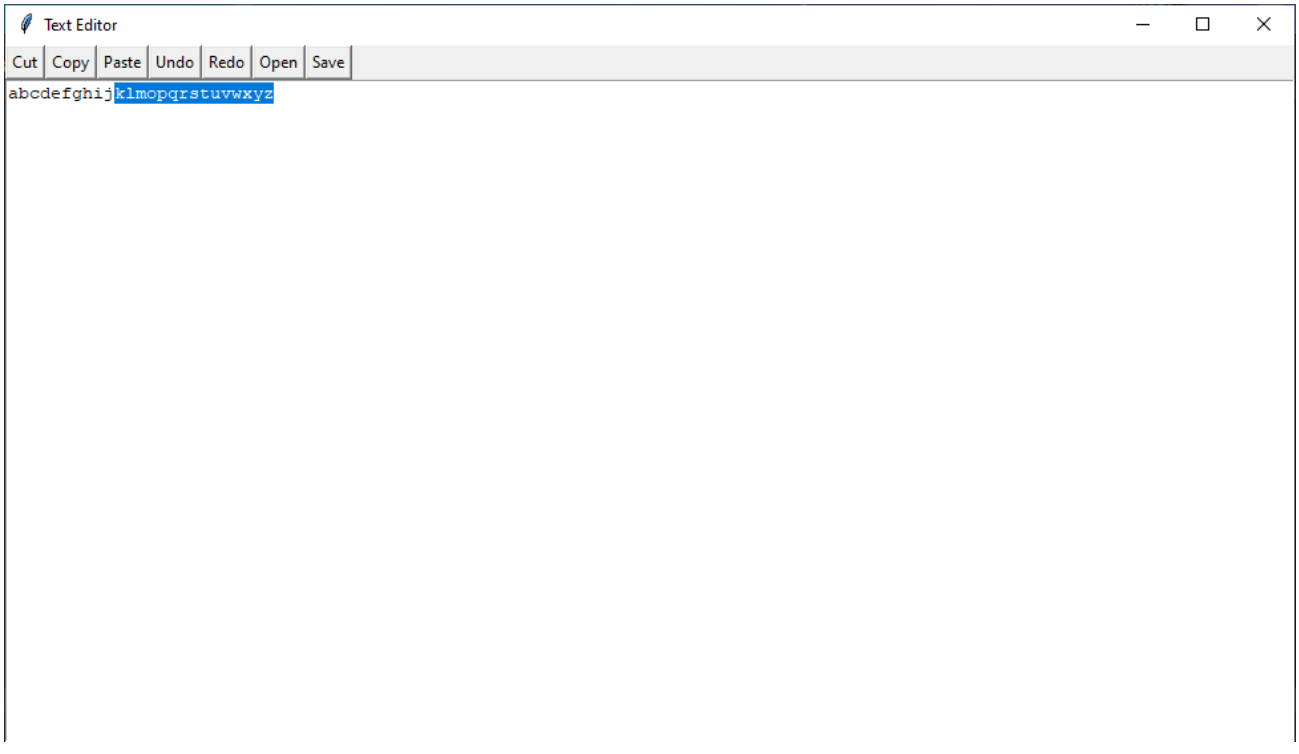
```
        file_path = filedialog.asksaveasfilename(defaultextension='.txt')
        if file_path:
            with open(file_path, 'w') as file:
                file.write(self.text_widget.get("1.0", "end"))

if __name__ == "__main__":
    app = TextEditorApp()
    app.root.mainloop()
```

The above code creates a GUI for a text editor. The text editor has Cut, Copy, Paste, Undo, Redo, Open and Save options. Also the inherent keyboard shortcuts like ‘Ctrl+Z’ for Cut option are also functional. When the user chooses the open and save option the directory window is opened allowing the user to select where to store and under what name.

Output:

- Initial GUI of Text Editor:

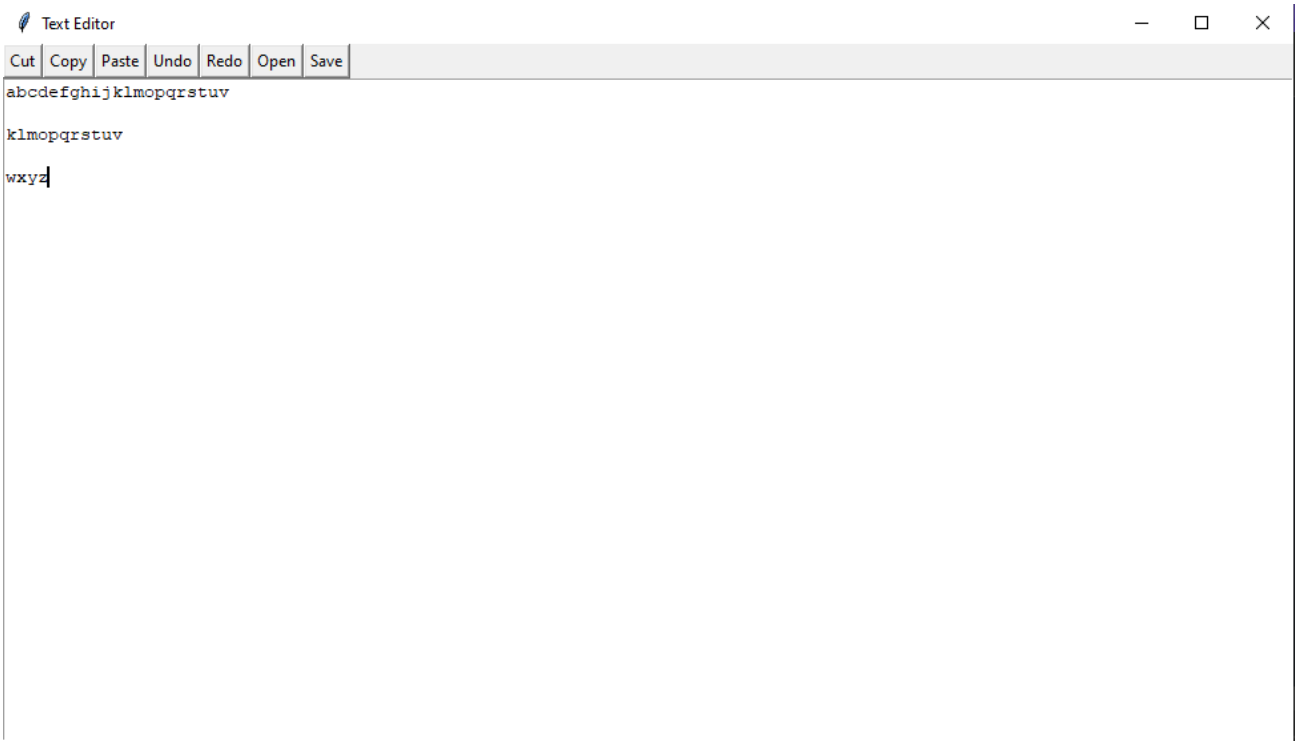
- Text Editor While Copying:



- Text Editor after Paste:

- Text Editor When Cut is used:

(From above state of text editor we will now cut 'wxyz’ from the line number 2 and paste it on new line)

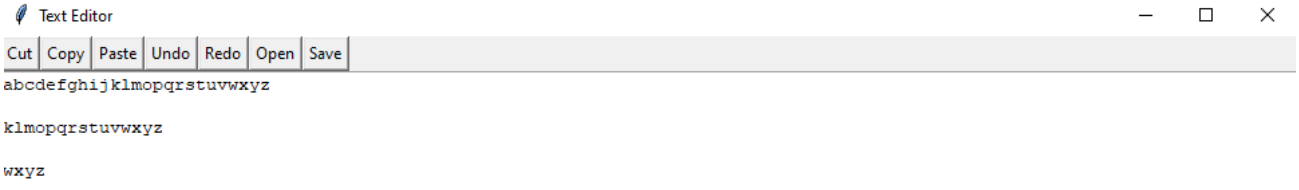


- Undo Option:

From above example we will undo pasting of ‘zyx’ from line number 3.

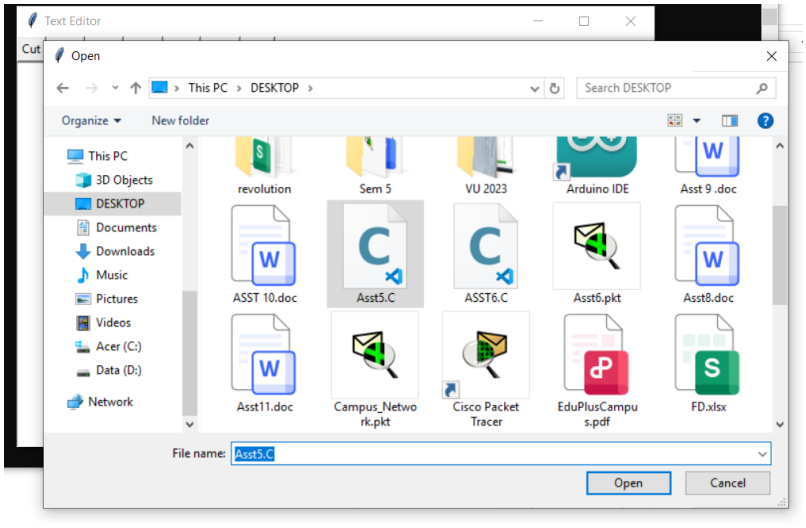
- Redo Option:

We will redo the previous undo option:



- Open and Save option:

Both these options when pressed open the windows directory to choose the file and location where to save.



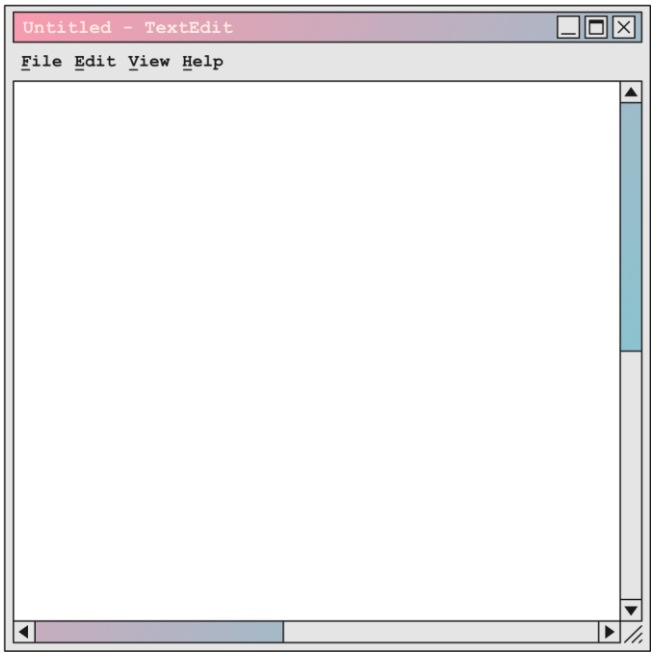
Conclusion:

In the above report we saw how the problem statement was implemented. ‘Python’ programming language was used for implementation because of it’s extensive libraries and user friendliness. The code implements GUI for text editor and created functional Cut, Copy, Paste, Undo, Redo, Open and Save options.

Report-03

Algorithm and Implementation

Submitted by
MERWIN PINTO
SRN 202100102
ROLL NO 1



Problem Statement:

Design and implement a basic text editor with fundamental features, such as text entry, editing, saving, and opening files. Include support for features like cut, copy, paste, undo, and redo. Develop a simple user interface for your editor and provide documentation on how to use these basic features effectively.

Algorithm and Data Strcuture:

Algorithm:

If we break down the code:

1.Initialization and GUI Setup:
The code starts by initializing a Tkinter application (self.root) to create the main window for the text editor.
The window's title is set to "Text Editor."

2.Toolbar Creation:
The program creates a toolbar at the top of the window (toolbar), which will hold various buttons for performing text editing operations.

3. Toolbar Buttons:
The following buttons are created on the toolbar:
"Cut": Calls the cut_text function.
"Copy": Calls the copy_text function.
"Paste": Calls the paste_text function.
"Undo": Calls the undo_text function.
"Redo": Calls the redo_text function.
"Open": Calls the open_file function.
"Save": Calls the save_file function.
These buttons are placed on the toolbar and serve as user interface elements for interacting with the text editor.

4. Text Widget Creation:
The program creates a Text widget (self.text_widget) in the main area of the window. This text widget is where the user can view and edit text.

5. History for Undo/Redo:
The code maintains a history of text changes to enable undo and redo functionality. The history is stored in a list (self.history), and the self.history_index keeps track of the current position in the history list.

6. Keyboard Shortcuts:
The code binds keyboard shortcuts to specific functions:
<Key>: Calls the on_text_change function whenever a key is pressed, recording text changes.
<Control-x>: Cuts the selected text and records the change.
<Control-c>: Copies the selected text.
<Control-v>: Pastes text from the clipboard and records the change.
<Control-z>: Initiates an undo operation.

7. Text Editing Functions:
The program defines various text editing functions that are associated with the toolbar buttons and keyboard shortcuts:
cut_text: Cuts the selected text and records the change in the history.
copy_text: Copies the selected text to the clipboard.
paste_text: Pastes text from the clipboard and records the change in the history.
undo_text: Reverts to a previous state in the history (undo).
redo_text: Moves forward in the history to redo actions.

8. Clipboard Operations:
Two functions are provided to manage the clipboard:
copy_to_clipboard: Clears the clipboard and appends the provided text.
get_clipboard_text: Retrieves text from the clipboard.

9. Recording Changes:
The record_change function captures the current text content and adds it to the history list.
It ensures that the history remains consistent with the user's actions.

10. Open and Save File:
The program allows the user to open and save text files:
open_file: Opens a file dialog to load the content of a selected file into the text editor.

save_file: Opens a file dialog for saving the current text to a file with a ".txt" extension.

11. Application Entry Point:
The program checks whether it's running as the main program (not imported as a module).
If it's the main program, it creates an instance of the TextEditorApp class and starts the main event loop using app.root.mainloop(). This is essential for the GUI to function.
In summary, this algorithm outlines the functionality and flow of a basic text editor with GUI features. The code's implementation utilizes the Tkinter library to create a graphical user interface for text editing and includes features like undo/redo, clipboard operations, and file open/save functionality.

Data Structure:

In this code, several data structures are used to implement the functionality of a text editor. Here are the main data structures used along with explanations:

1.Class:
TextEditorApp: This class serves as the main data structure that encapsulates the entire text editor application. It contains attributes and methods for managing the GUI, text editing, clipboard, and history.

2.Attributes within the TextEditorApp class:
self.root: The main Tkinter window, which acts as the container for the entire application.
self.text_widget: A Text widget within the window where the user can edit and view text.
self.history: A list that stores snapshots of the text content after each change, enabling undo and redo operations.
self.history_index: An integer representing the current position in the history list.

3.Local Variables within Methods:
Local variables are used within various methods, such as selected_text, clipboard_text, and file_path, to hold temporary data needed for specific operations.

4.Data Structures for GUI Elements:

tk.Frame and tk.Button: These Tkinter data structures are used for creating the toolbar and toolbar buttons. The toolbar holds buttons for various text editing operations.
tk.Text: The main text editing area is represented by a Text widget, which allows multi-line text input and display.

5.List:
self.history is a list used to store previous versions of the text content. It keeps a history of text changes to enable undo and redo functionality.
Dictionary (Not explicitly used in the code):

6.Dictionaries are not used in this code, but they are a common data structure for various applications. In a text editor, you might use a dictionary to store key-value pairs for settings or configurations.

7.String:
Strings are used extensively in the code to store text content. They represent the text displayed in the text widget, the content in the clipboard, and the file paths.

8.Integer:
The self.history_index variable is an integer used to keep track of the current position in the history list.
These are the primary data structures used in the code to create the text editor application, manage text content, and support various user interface elements and operations.

Conclusion:

In summary, the code is a basic text editor with a graphical interface. It mainly uses strings, lists, and integers to manage text content, history, and system programming is not a primary focus.