Text Editor

**Project Report**

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE AWARD OF THE DEGREE OF

**Bachelor Of Technology**

**Computer Science Engineering**

Submitted By

**ASHISH KUMAR PATEL**

(2103490100013)

December 2024



**Dr. APJ Abdul Kalam Technical University, Uttar Pradesh**

Lucknow, INDIA

**ACKNOWLEDGEMENT**

I would like to express my special thanks of gratitude to my teacher DR. VIJAY SINGH who gave me the golden opportunity to do this wonderful project on the topic (**Text Editor**), which also helped me in doing a lot of Research and i came to know about so many new things I am really thankful to them. Secondly i would also like to thank my friends who helped me a lot in finalizing this project within the limited time frame.

**Certificate**

This is to certify that this project report intitled **Text Editor** being submitted to the record Maharana Pratap Engineering College by Ashish Kumar Patel, Roll No.2103490100013 for partial fulfilment for the B.Tech in Computer Science and Engineering is a bonafide record of work carried out by them under the guidance and supervision for the academic year 2024-25.

Abhishek Singh Senger

(Project Incharge)

**CONTENT OF REPORT :**

1. Introduction ………………….
2. Objective ………………….

2.1 MODULE 1 ……………….

2.2 MODULE 2 ……………….

2.3 MODULE 3 ……………….

2.4 MODULE 4 ……………….

2.5 MODULE 5 ……………….

1. S/W, H/W Requirements .…….
2. Flowcharts ……………………
3. Literature Review …………….
4. Technologies Required ……….
5. Feasibility Study …………..….

7.1 Economic Feasibility………….

7.2 Technical Feasibility…………..

7.3 Operational Feasibility………...

1. Output Analysis ……………….
2. Conclusion …………………….
3. Test Analysis ………………….
4. Future Scope ………………….
5. References …………………….
6. APPENDIX

13.1 Code of the solution………...

**INTRODUCTION:**

A text editor is an application that allows a user to create and revise documents in a computer. The project aims to create a simple text editor using the ‘Tkinter’ module in Python. The text editor will have the basic functionalities of a text editor, such as the ability to create, open, save, and edit text files.

The project will start by importing the necessary modules and libraries. Then, the main window will be created. The main window will contain a text area, a scrollbar, and a button. The text area will be where the user can type their text. The scrollbar will allow the user to scroll through the text. The button will save the text to a file.

The save function will be created next. The save function will take the text from the text area and save it to a file. The user will be able to choose the name of the file and the location where the file will be saved.

The main command will be created last. The main command will create the main window and start the text editor.

The project will be tested by creating a new text file, typing some text into the text file, and saving the text file. The project will also be tested by opening an existing text file, editing the text file, and saving the text file.

In Python, a text editor is a software tool that allows developers to create, modify, and save Python code files. Text editors are essential for writing and editing source code, as they provide a user-friendly interface and features tailored to programming tasks. Unlike integrated development environments (IDEs), which offer a more comprehensive set of tools, text editors focus primarily on text manipulation and are often lighter in terms of resource usage.

These text editors provide a flexible environment for Python developers, allowing them to write and manage their code efficiently. The choice of a text editor often depends on personal preferences, the specific needs of the project, and the desired features for coding productivity.

A Python GUI-based text editor is a software application that allows users to create, edit, and manipulate text-based files through a graphical user interface (GUI) built using Python. Unlike command-line text editors, GUI-based text editors provide a visual environment with buttons, menus, and other graphical elements to facilitate user interaction.

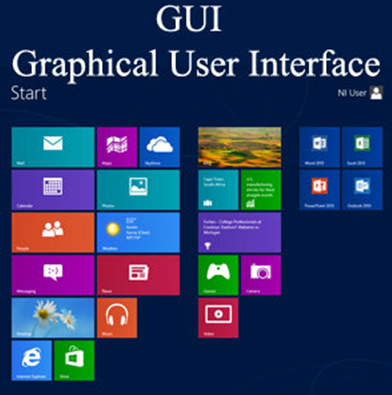
These text editors are often created using GUI frameworks such as Tkinter, PyQt, or Kivy, which enable developers to design interactive and user-friendly interfaces. The choice of GUI framework depends on factors like personal preference, project requirements, and the desired look and feel of the application.

Python GUI-based text editors typically include features such as syntax highlighting, code folding, search and replace functionality, and support for various file formats. Additionally, they may offer customization options, themes, and extensions to enhance the user experience and cater to the specific needs of programmers and writers.

Developers often choose to build custom text editors tailored to their workflow or project requirements, leveraging the flexibility and extensibility provided by Python and its GUI frameworks. Whether used for coding, writing, or general text editing, these applications contribute to a more intuitive and efficient text-editing experience for users within the Python ecosystem.

## **GUI: Graphical USer Interface**

GUI stands for Graphical User Interface. It refers to an interface that allows one to interact with electronic devices like computers and tablets through graphic elements. It uses icons, menus and other graphical representations to display information, as opposed to text-based commands. The graphic elements enable users to give commands to the computer and select functions by using mouse or other input devices.



The programs which run under a GUI has a specific set of graphic elements so that after learning a specific interface, a user can use these programs without learning new commands.

Xerox 8010 Information system was the first GUI-centric computer operating model. It was developed at Xerox PARC by Alan Kay, Douglas Engelbart and their associates.

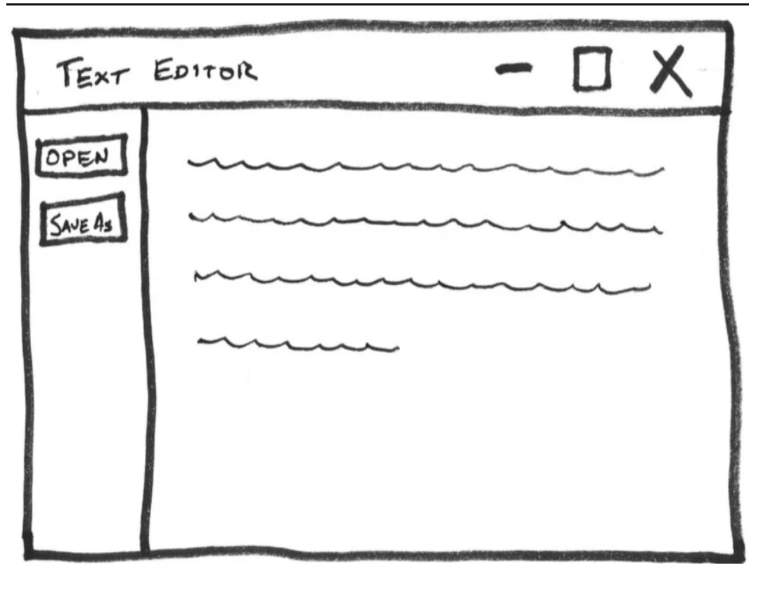
As of 2014, the most popular GUIs are Microsoft Windows and Mac OS X. And, if we talk about mobile devices, the Apple's IOS and Google's Android Interface are the widely used GUIs.

## **Basic Components of a GUI**

* **Pointer:** It is a symbol that appears on the display screen. It can be moved to select commands and objects.
* **Pointing device:** It allows you move the pointer and select objects on the screen, e.g. mouse or trackball.
* **Icons:** It refers to small images on the display screen that represent commands, files, windows etc. Using pointer and pointing device, you can execute these commands.
* **Desktop:** It is the display screen that contains the icons.

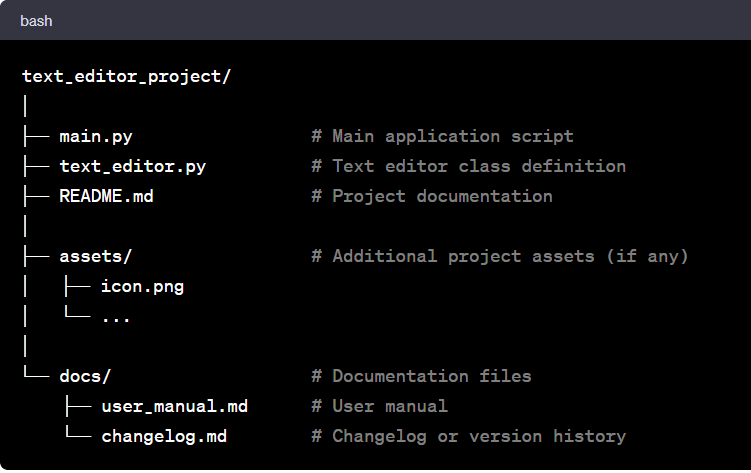
### GUI KEY Benefits

* It allows you to place more information within a program.
* The graphics allow users to use complex programs with greater ease.
* It saves time as you do not need to edit configurations manually.
* You can easily memorize the tasks (point-and-click).
* Helps create user-friendly software with a point-and-click interface.



**STRUCTURE :**

The structure of a Python project can vary, but here's a suggested outline for the text editor project using Tkinter. This structure includes main files, subdirectories, and potential content for each:

****

**Details:**

1. **‘main.py’:**

* The main script to launch the text editor application.
* Initializes the Tkinter application and creates an instance of the TextEditor class.

**2. ‘text\_editor.py’:**

* Contains the definition of the TextEditor class.
* Includes methods for various functionalities like file handling, text editing, and GUI setup.

**3. ‘README.md’:**

* Project documentation providing an overview of the project.
* Instructions for setting up and running the application.
* Project description, goals, and potential future enhancements.

**4. ‘assets/’:**

* Directory for additional project assets (e.g., icons, images).
* It may include the application icon, images used in the user interface, etc.

**5. ‘docs/’:**

* Directory for project documentation.
* user\_manual.md: Detailed instructions on how to use the text editor.
* changelog.md: A log of changes made to the project across different versions.

**Notes:**

* **Version Control (Git):**
* Initialize a Git repository for version control.
* Include a .gitignore file to exclude unnecessary files (e.g., \_\_pycache\_\_/, .vscode/, .idea/, \*.pyc, etc.).

**Virtual Environment:**

* Consider using a virtual environment for dependencies.
* Include a requirements.txt file if external libraries are used.

**Testing (Optional):**

* If applicable, create a tests/ directory for unit tests.
* Include a test\_main.py file for testing core functionalities.

**CI/CD (Optional):**

* Set up continuous integration and deployment configuration files (e.g., .travis.yml, .github/workflows/).

**License:**

* Include a LICENSE file specifying the project's license (e.g., MIT License).

This structure is a starting point, and you can adapt it based on your preferences and project requirements. If your project becomes more complex, you might consider organizing the code into multiple modules or packages within subdirectories.

**OBJECTIVE:**

The objective of this text editor project is to create a simple yet functional GUI-based text editor using the Tkinter library in Python. The project aims to provide users with a basic platform for creating, editing, and saving text documents. Below are the detailed objectives of the project:

**User Interface (UI):**

* Create an intuitive and user-friendly interface using Tkinter a standard GUI toolkit in Python.
* Implement a main window that includes a text widget for editing text and a menu bar for accessing various functionalities.

**File Operations:**

* Provide essential file operations such as creating a new file, opening an existing file, saving the current file, and saving the current file with a new name (Save As).
* Implement file dialog boxes to facilitate easy file selection and saving.

**Text Editing Features:**

* Enable basic text editing functionalities like cut, copy, paste, undo, and redo through menu options and keyboard shortcuts.
* Implement the ability to select all text in the editor.

**Search and Replace**:

* Allow users to find occurrences of a specific text within the document.
* Provide a replace functionality, allowing users to replace specific occurrences of a text with another.

**Font and Font Size Options:**

* Include options in the menu to change the font family and font size of the text.
* Utilize the Tkinter Font class to dynamically update the font settings of the text widget.

**Status Bar:**

* Implement a status bar at the bottom of the window to display informative messages to the user.
* Toggle the visibility of the status bar based on user preference.

**Additional Features:**

* Allow users to toggle the visibility of the status bar.
* Use undo and redo functionality for text editing operations.
* Provide feedback to the user about the current state of the application.

**Project Structure:**

* Organize the code into classes and methods to ensure modularity and maintainability.
* Follow best practices for code readability and documentation.

**Testing and Debugging:**

* Test the application thoroughly to ensure that all functionalities work as expected.
* Handle errors gracefully and provide meaningful error messages.

**Educational Purpose:**

* Serve as a learning project for individuals interested in GUI development with Tkinter and text processing in Python.
* Provide a foundation for further expansion and customization based on the user's needs.

By achieving these objectives, the project aims to deliver a basic but functional text editor that users can utilize for simple text editing tasks. The project also serves as a learning opportunity for those interested in GUI development in Python.

**ABSTRACT:**

The goal of this project is to create a simple text editor in Python. The text editor will be able to open and save files, as well as allow users to edit text. The project will be completed using the Tkinter library, which is a GUI library for Python.

The first step in creating the text editor will be to import the necessary libraries. The following libraries will be used:

**tkinter:** This library will be used to create the GUI for the text editor.

**tkinter.filedialog:** This library will be used to allow users to open and save files.

Once the necessary libraries have been imported, the next step will be to create the main window for the text editor. The main window will contain a text area, a scrollbar, and a save button.

The text area will be used to display the text that the user is editing. The scrollbar will be used to allow the user to scroll through the text. The save button will be used to save the text to a file.

The next step will be to create the save function. The save function will take the text from the text area and save it to a file. The user will be able to choose the name of the file and the location where the file will be saved.

The final step will be to create the main command. The main command will create the main window and start the text editor.

The text editor will be a simple, but useful tool for users who need to edit text. The text editor will be able to open and save files, as well as allow users to edit text. The project will be completed using the Tkinter library, which is a GUI library for Python.

**MODULE USED:**

The Python text editor using Tkinter, several modules are used to implement the graphical user interface (GUI) and specific functionalities. Here's a summary of the modules used:

**1. tkinter Module:**

Used for creating the main GUI components, such as windows, text widgets, menus, and labels.



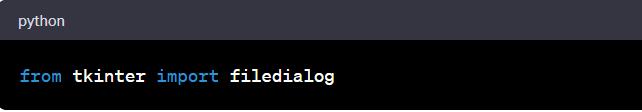
**2. tkinter.ttk Module:**

Used for accessing themed Tkinter widgets, which includes additional styling options.



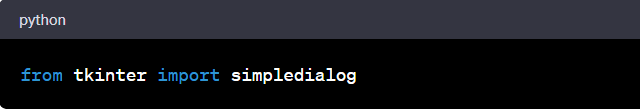
**3. tkinter.filedialog Module:**

Used for creating file dialogs to handle opening and saving files.



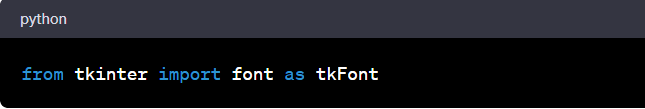
**4. tkinter.simpledialog Module:**

Used for creating simple dialogs to get user input, particularly for search and replace queries.



**5. tkinter.font Module:**

Used for handling fonts and font sizes in the text widget.



Here, five modules from the standard Python library are used in this project to create the GUI and implement various functionalities. These modules provide the necessary tools and components for building a simple text editor with basic editing features.

**HARDWARE AND SOFTWARE:**

The text editor project implemented in Python using Tkinter primarily relies on software components, and the hardware requirements are standard for running Python applications. Here's an overview of the hardware and software aspects:

**Software Components:**

**Python:**

The project is developed using the Python programming language. Python provides a straightforward and expressive syntax, making it suitable for GUI applications.

**Tkinter:**

Tkinter is the standard GUI toolkit that comes with Python. It is used for creating the graphical user interface of the text editor.

**Operating System:**

The text editor can run on various operating systems, including Windows, macOS, and Linux, as Tkinter is platform-independent.

**Hardware Components:**

**Computer:**

Any standard computer capable of running Python applications can be used. This includes desktops, laptops, and servers.

**Processor (CPU):**

The project does not have specific requirements for the CPU. A modern processor is sufficient for running a Python-based text editor.

**Memory (RAM):**

Adequate RAM is necessary for smooth execution. However, as a text editor is not resource-intensive, a typical amount of RAM found in most computers is sufficient.

**Storage:**

The project does not have large storage requirements. The storage capacity needed is minimal, as the focus is on text file manipulation.

**Development Environment:**

**Code Editor or IDE:**

A code editor or integrated development environment (IDE) is used for writing and editing the Python code. Examples include Visual Studio Code, PyCharm, or IDLE (Python's built-in IDE).

**Python Interpreter:**

The Python interpreter is required to execute the Python code. The project is developed using Python 3.x.

**Version Control (Optional):**

Version control systems like Git can be used for tracking changes to the codebase, collaborating with others, and managing the project's history.

**Additional Notes:**

The Tkinter library is part of the Python standard library, so there is no need for additional installations for the GUI components.

The project is designed to be lightweight, and the hardware and software requirements are minimal compared to more complex applications.

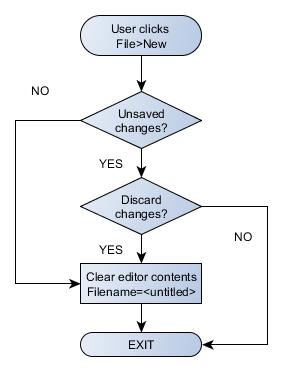
The text editor's portability across operating systems is one of the advantages of using Tkinter for GUI development in Python.

Overall, the hardware and software requirements for this text editor project are common and well within the capabilities of standard computing systems.

**FLOWCHART:**

**Creating a new file**

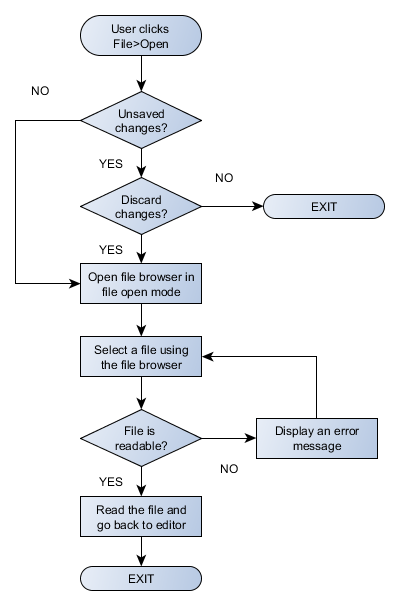
The Text Editor user can create a new file by opening the file menu and selecting “New”. If there are unsaved changes we need to ask whether we can discard the changes. These steps are shown in the flowchart below.

[](https://n9.dy.fi/wp-content/uploads/2016/01/TextEditor-UC-01.jpg)

The figure below illustrates the signalling flow for the File>New action. The user has the QML edit Page open and then selects New from the the File menu. This emits the signal file New Requested which has been connected to the slot with the same name in the C++ side. We will pass the edit Page. content property (= the current content as a string) to the file New Requested slot and compare that to the current content. If they match we will call new Confirmed slot which will clear the current content and emit the signal editor Cleared. This will invoke the QML event handler on Editor Cleared that will clear the editor buffer and initialize the file name and path (and return control to the user). If there were unsaved changes we will emit new To Be Confirmed signal that will be catched by the on New To Be Confirmed event handler on the QML side. We will open a dialog (new Confirm Dialog) to ask whether we can discard the changes. If the user answers NO we will return control to the user, otherwise we will emit the signal new Confirmed that is connected to the slot new Confirmed on the C++ side. From there we will follow the same steps as described earlier.

**Opening an existing file**

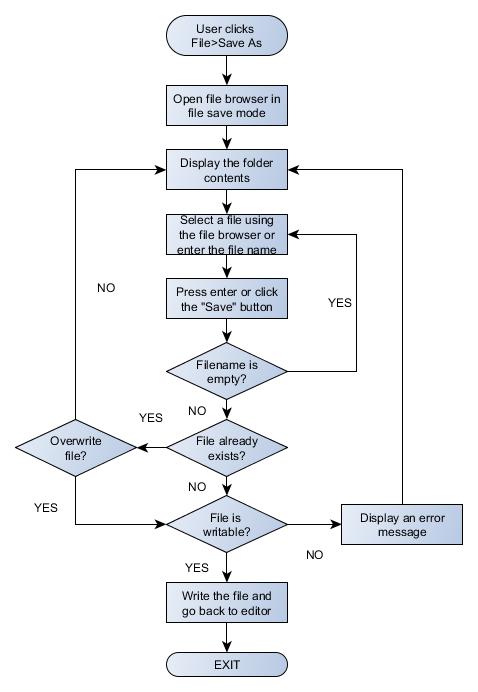
The Text Editor user can open an existing file by opening the file menu and selecting “Open”. If there are unsaved changes we need to ask whether we can discard the changes. The file browser can then be used to select a file. These steps are shown in the flowchart below.

[](https://n9.dy.fi/wp-content/uploads/2016/01/TextEditor-UC-09.png)

The signalling flow is shown below. As before we will check if there are unsaved changes. If there are no unsaved changes or if the user decides to discard the changes we will emit the signal browse Requested which will trigger the on Browse Requested event handler on the QML side. This will open the browse Page (will be pushed into the page stack). As input the browse Page needs to know the path to browse and whether we are browsing to open or save a file.

### Saving to a new file

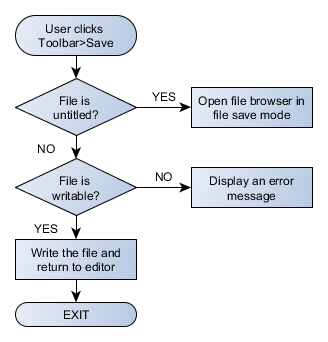
The user can save the editor contents to a new file by selecting “Save As” from the file menu. The file browser will be opened to select the destination folder and file. The user can also enter the file name directly using normal text entry method with the on-screen keyboard. If the file already exists we will ask whether we can overwrite the file. If the file is write-protected we will give an appropriate error message. These steps are shown in the flow chart below.

[](https://n9.dy.fi/wp-content/uploads/2016/01/TextEditor-UC-12B.jpg)

The signalling flow for the first steps is shown below. When File >Save As is clicked the signal menu Save As Clicked is sent to the python slot. The slot method sends the current folder information back to the QML side. The browse Page element is pushed into the page stack to make it visible. The save Requested parameter is used to select whether we want to open or save a file.

### Saving the file

When the user clicks the “Quick Save” button in the editor toolbar the editor content will be saved to the current file. If there is no file name (file name is “Untitled”) the file browser will be opened to let the user select the file name (here the steps are the same as described above). If the file name is already defined we will try to save the file. If save failed we will show an appropriate error message. If save succeeded we will just return to the editor.

[](https://n9.dy.fi/wp-content/uploads/2016/01/TextEditor-UC-17.png)

The signalling flow is shown below. Clicking the “Quick Save” button will send the signal menu Save Clicked to the python Text Editor object. If the editor is still untitled we will open the file browser by calling the menu SaveAs Clicked method. Otherwise we will call the save Current Content method to save the editor content and go back to the edit Page.

**ALGORITHM:**

The provided code for the Python text editor using Tkinter doesn't explicitly present a detailed algorithm, as it's more of an implementation of a graphical user interface (GUI) with specific functionalities. However, I can outline the general flow and algorithms used for key functionalities in the code:

**File Handling Functions (e.g., open\_file, save\_file):**

1. **Open File Algorithm:**

* Use filedialog.askopenfilename to get the path of the selected file.
* Open the file using open in read mode.
* Read the file content and insert it into the text widget.

1. **Save File Algorithm:**

* Use filedialog.asksaveasfilename to get the path where the user wants to save the file.
* Open the file using open in write mode.
* Write the content of the text widget to the file.

**Text Editing Functions (e.g., cut, copy, paste):**

1. **Cut Algorithm:**

* Use text\_widget.event\_generate("<<Cut>>") to perform the cut operation.

1. **Copy Algorithm:**

* Use text\_widget.event\_generate("<<Copy>>") to perform the copy operation.

1. **Paste Algorithm:**

* Use text\_widget.event\_generate("<<Paste>>") to perform the paste operation.

**Font Handling Functions (e.g., change\_font):**

1. **Change Font Algorithm:**

* Use filedialog.askstring to get the font name from the user.
* Use tkFont.nametofont to get the details of the current font.
* Configure the text widget's font using ‘text\_widget.configure’.

**Search and Replace Functions (e.g., find\_text, replace\_text):**

1. **Find Text Algorithm:**

* Use simpledialog.askstring to get the search query from the user.
* Use text\_widget.search to find occurrences of the query in the text.
* Highlight or select the found text.

1. **Replace Text Algorithm:**

* Use simpledialog.askstring to get the search and replace queries from the user.
* Use text\_widget.get to retrieve the content of the text widget.
* Replace occurrences of the search query with the replace query.
* Update the text widget with the modified content.

**Status Bar Handling (e.g., update\_status):**

1. **Update Status Algorithm:**

* Use status\_var.set to update the content of the status\_var variable.
* Reflect the current status in the status bar.

These algorithms represent the high-level steps involved in the core functionalities of the text editor. The actual implementation details can be found in the corresponding functions within the provided code.

**PSEUDO CODE:**

The pseudocode representation of the key functionalities in the provided Python text editor project using Tkinter :

**class Text Editor:**

**method \_\_init\_\_(self, root):**

* **Initialize the text editor**
  + Create root window
  + Set up GUI elements (text widget, menu bar, status bar)
  + Set default status

**method new\_file(self):**

* **Clear the text widget and update status**
  + Delete content from text widget
  + Set window title to default
  + Update status: "New file created"

**method open\_file(self):**

* **Open a file and display its content**
  + Get file path using filedialog.askopenfilename
  + Read file content
  + Display content in the text widget
  + Update window title and status

**method save\_file(self):**

* **Save the current content to a file**
* Get file path using filedialog.asksaveasfilename
* Write text widget content to the file
* Update window title and status

**method cut(self):**

* **Cut selected text in the text widget**
  + Trigger "<<Cut>>" event
  + Update status: "Cut"

**method copy(self):**

* **Copy selected text in the text widget**
  + Trigger "<<Copy>>" event
  + Update status: "Copy"

**method paste(self):**

* **Paste copied/cut text into the text widget**
  + Trigger "<<Paste>>" event
  + Update status: "Paste"

**method change\_font(self):**

* **Change the font of the text widget**
  + Get font name from user using filedialog.askstring
  + Get current font details using tkFont.nametofont
  + Configure text widget font
  + Update status: "Font changed"

**method find\_text(self):**

* **Find and highlight occurrences of a text in the text widget**
  + Get search query from user using simpledialog.askstring
  + Search for the query in the text widget
  + Highlight or select occurrences
  + Update status: "Found occurrences of: {query}"

**method replace\_text(self):**

* **Replace occurrences of a text in the text widget**
  + Get find and replace queries from user using simpledialog.askstring
  + Get current content of the text widget
  + Replace occurrences in the content
  + Update text widget with modified content
  + Update status: "Replaced occurrences of: {find\_query}"

**method toggle\_status\_bar(self):**

* **Toggle the visibility of the status bar**
  + If status bar is visible, hide it; otherwise, show it
  + Update status: "Status Bar hidden" or "Status Bar shown"

**method update\_status(self, message):**

* **Update the status bar with a given message**
* Set the message to the status\_var

**LITERATURE REVIEW:**

A **literature review** is an overview of the previously published works on a topic. The term can refer to a full [scholarly paper](https://en.wikipedia.org/wiki/Scholarly_paper) or a section of a scholarly work such as a book, or an article. Either way, a literature review is supposed to provide the [researcher](https://en.wikipedia.org/wiki/Research)/author and the audiences with a general image of the existing knowledge on the topic under question. A good literature review can ensure that a proper research question has been asked and a proper theoretical framework and/or research [methodology](https://en.wikipedia.org/wiki/Methodology) have been chosen. To be precise, a literature review serves to situate the current study within the body of the relevant literature and to provide context for the reader. In such case, the review usually precedes the methodology and results sections of the work.

Producing a literature review is often a part of graduate and post-graduate student work, including in the preparation of a [thesis](https://en.wikipedia.org/wiki/Thesis), [dissertation](https://en.wikipedia.org/wiki/Dissertation), or a journal article. Literature reviews are also common in a [research proposal](https://en.wikipedia.org/wiki/Research_proposal) or prospectus (the document that is approved before a student formally begins a dissertation or thesis).[[1]](https://en.wikipedia.org/wiki/Literature_review#cite_note-1)

A literature review can be a type of [review article](https://en.wikipedia.org/wiki/Review_article). In this sense, a literature review is a [scholarly paper](https://en.wikipedia.org/wiki/Scholarly_paper) that presents the current knowledge including substantive findings as well as theoretical and methodological contributions to a particular topic. Literature reviews are [secondary sources](https://en.wikipedia.org/wiki/Secondary_sources) and do not report new or original experimental work. Most often associated with academic-oriented literature, such reviews are found in [academic journals](https://en.wikipedia.org/wiki/Academic_journals) and are not to be confused with [book reviews](https://en.wikipedia.org/wiki/Book_reviews), which may also appear in the same publication. Literature reviews are a basis for research in nearly every academic field.

**Evaluation of Text Editors**

This paper presents a methodology for evaluating computer text editors from the viewpoint of their users from novices learning the editor to dedicated experts who have mastered the editor. The dimensions which this methodology addresses are:

* Time to perform edit tasks by experts.
* Errors made by experts.
* Learning of basic edit tasks by novices.
* Functionality over all possible edit tasks.

The methodology is objective and thorough, yet easy to use. The criterion of objectivity implies that the evaluation scheme not be biased in favor of any particular editor's conceptual model its way of representing text and operations on the text. In addition, data is gathered by observing people who are equally familiar with each system. Thoroughness implies that several different aspects of editor usage be considered. Ease-of-use means that the methodology is usable by editor designers, managers of word processing centers, or other non-psychologists who need this kind of information, but have limited time and equipment resources. In this paper, we explain the methodology first, then give some interesting empirical results from applying it to several editors.

**THE METHODOLOGY**

The methodology is based on a taxonomy of 212 editing tasks which could be performed by a text editor. These tasks are specified in terms of their effect on a document, independent of any specific editor's conceptual model. The tasks cover:

* modifying the content of the document,
* altering the appearance of paragraphs and characters and the page layout,
* creating and modifying special kinds of text (such as tables),
* specifying locations and text in the document in various ways,
* programming automatic repetition of edits,
* displaying the document in various ways,
* printing, filing, and other miscellaneous tasks.

The functionality dimension of an editor is measured with respect to this taxonomy. However, comparisons between editors on the performance dimensions (time, errors, and learning) must be done on tasks which all editors can do. For this purpose, a set of 32 core tasks was identified. The core tasks were chosen to be those tasks that most editors perform and that are most common in everyday work. Most of the core tasks are generated by crossing a set of basic text editing operations with a set of basic text entities. Thus, a core task consists of one of the operations (insert, delete, replace, move, copy, transpose, split, merge) applied to one of (or a string of) the text entities (character, word, number, sentence, paragraph, line, section). The core tasks also include locating a place in the online document which corresponds to a place in a hardcopy document (using the editor's simplest addressing mechanism), locating a siring of text according to its contents, displaying a continuous segment of the document, saving and retrieving copies of the document, printing, and creating a new document.

**Time:** The speed at which normal text modification can be done is measured by observing expert users as they perform a set of benchmark tasks from the core tasks. There are 50 editing tasks in the benchmark, embedded in four documents: a short inter-office memo, two two-page reports, and one chapter from a philosophy book. The locations and complexities of the benchmark tasks are randomly distributed. The distribution emphasizes small tasks because those are most common in normal work and tasks involving boundary conditions in order to identify special cases, such as insertion at the beginning of a paragraph, which editors may treat awkwardly

**Errors:** The error-proneness of the editor is measured by recording the amount of time the expert users spend making and correcting errors on the benchmark tasks. Only those errors which take more than a few seconds to correct are noted (which is the best that can be done with a stopwatch). Thus, the time taken by simple typographical errors is not counted. Actually, this does not hurt the error time estimate too much, since the total amount of time in these Kinds of small eiiors is relatively small. In addition to liming errors made and corrected while the user is working on the benchmarks, the evaluator also notes the tasks incorrectly performed; at the end of the experiment the user is asked to go back and complete those tasks correctly. The time to redo these tasks is added to the error time. Thus, the error score consists of all this error time as a percentage of the error-free time. The overall error score is the average for the four expert users.

**Learning:** The ease of learning to perform basic text modifications on the editor is tested by teaching four novices (with no previous computer or word processing experience) to perform the core tasks. The learning tests are performed in a one-on-one situation, i.e., by individually teaching each novice the editor. The evaluator orally teaches the novice how to do the core tasks in the editor, and the subject practices the tasks on the system. The methodology specifies the order in which to teach the tasks, but it is up to the evaluator to determine which specific editor commands to teach. Although all the teaching is oral, the evaluator supplies the novice with a one-page summary sheet listing all commands, so that the training is not hung up because of simple memory difficulties. After a set of tasks is taught, the novice is given a quiz, consisting of a document marked with changes to be made. Only a sample of possible tasks appears on each quiz, and not all tasks on the quiz have necessarily been taught up to that point This allows for the novice to figure out, if possible, how to do tasks which haven't explicitly been taught. Referring to the summary sheet is permitted, but discouraged. The novice performs all of the tasks that he or she knows how to do, after which s/he is invited to take a short break if s/he wants it Then another teaching period begins. In all, there are five training-plus-quiz cycles to teach all of the core tasks. Learning is evaluated by scoring the number of different tasks the subject has shown the ability to perform on the quizzes. The learning i»core is the total number of different tasks learned divided by the amount of time taken for the experiment, that is, the average time it takes to learn a task. The overall learning score is the average learning time for the four novices.

**Functionality:** The range of functionality available in the editor is tested by a checklist of tasks covering the full task taxonomy. Determining whether a task can be done or not with a given system isn't as trivial as it seems at first glance. Almost any task can be performed on almost any system, given enough effort. Consequently, the editor gets fiill credit for a task only if the task can be done efficiently with the system. It gets half credit if the task can be done clumsily (where clumsiness has several aspects: repetitiousness, requiring excessive text typing, limitations in the values of parameters to the task, interference with other functions, or a requirement of substantial planning by the user). The editor gets no credit for a task if either it can't be done at all (like use of italic typefaces on a system made for a line printer) or if doing the task requires as much work as retyping all affected text (such as manually inserting a heading on every page). The functionality checklist is filled out by a very experienced user of the editor, who may refer to a reference manual to ensure accuracy. The overall functionality score is the percentage of the total number of tasks that the editor can do. This percentage may be broken down by subclasses of tasks to show the strengths and weakness of the editor.

**TECHNOLOGY USED:**

The technology stack used in the Python text editor project, implemented with Tkinter, consists of several components:

**1.Programming Language:**

•Python: The project is primarily developed using the Python programming language. Python provides simplicity and ease of use, making it a popular choice for GUI application development.

**2.Graphical User Interface (GUI) Library**:

•Tkinter: Tkinter is the standard GUI toolkit that comes with Python. It provides a set of tools for creating graphical interfaces and is widely used for developing desktop applications.

**3.Text Editing and Styling:**

•Text Widget: The project utilizes the Tkinter Text widget for text editing. This widget supports various text-related functionalities, including styling and formatting.

**4.Font Handling:**

•Tkinter Font Module: The ‘tkinter.font’ module is used for handling fonts and font sizes. It allows dynamic changes to the font settings of the Text widget.

**5.File Dialogs:**

•tkinter.filedialog: The tkinter.filedialog module is used for implementing file dialogs, such as opening and saving files. It provides a platform-independent way to interact with the file system.

**6.Dialogs for User Input:**

•tkinter.simpledialog: The tkinter. simpledialog module is used for creating simple dialogs for user input. It is employed, for example, in the "Find" and "Replace" functionalities.

**7.Styling and Theming:**

•ttk.Style: The ttk.Style class from the tkinter.ttk module is used for styling the application. It provides a way to set the overall appearance of the GUI components.

**8.Font Handling (Alternative):**

•tkinter.font: The tkinter.font module is used for handling font-related operations. It is employed, for instance, when changing the font family and font size.

**9.Version Control (Optional):**

•Git: While not explicitly mentioned in the code, version control systems like Git can be used for tracking changes to the codebase, collaborating with others, and managing the project's history.

**10.Integrated Development Environment (IDE):**

•Code Editor or IDE: The project can be developed using various code editors or integrated development environments, such as Visual Studio Code, PyCharm, or IDLE.

These technologies work together to create a simple and functional text editor with a graphical user interface. Tkinter, being part of the Python standard library, simplifies the process of GUI development and provides a foundation for building desktop applications. The chosen technologies align with the project's goal of creating a lightweight and platform-independent text editor.

**FEASIBILITY USED:**

A feasibility study is a detailed analysis that considers all of the critical aspects of a proposed project in order to determine the likelihood of it succeeding.

Success in business may be defined primarily by [return on investment](https://www.investopedia.com/terms/r/returnoninvestment.asp), meaning that the project will generate enough profit to justify the investment. However, many other important factors may be identified on the plus or minus side, such as community reaction and environmental impact.

Although feasibility studies can help project managers determine the risk and return of pursuing a plan of action, several steps should be considered before moving forward.

KEY TAKEAWAYS

* A company may conduct a feasibility study when it's considering launching a new business, adding a new product line, or acquiring a rival.
* A feasibility study assesses the potential for success of the proposed plan or project by defining its expected costs and projected benefits in detail.
* It's a good idea to have a contingency plan on hand in case the original project is found to be infeasible.

## **Understanding a Feasibility Study:**

A feasibility study is an assessment of the practicality of a proposed plan or project. A feasibility study analyzes the viability of a project to determine whether the project or venture is likely to succeed. The study is also designed to identify potential issues and problems that could arise while pursuing the project.

As part of the feasibility study, project managers must determine whether they have enough of the right people, financial resources, and technology. The study must also determine the return on investment, whether this is measured as a financial gain or a benefit to society, as in the case of a nonprofit project.

The feasibility study might include a [cash flow](https://www.investopedia.com/terms/c/cashflow.asp) analysis, measuring the level of cash generated from revenue versus the project's [operating costs](https://www.investopedia.com/ask/answers/101314/what-are-differences-between-operating-expenses-and-overhead-expenses.asp). A [risk assessment](https://www.investopedia.com/terms/r/risk-assessment.asp) must also be completed to determine whether the return is enough to offset the risk of undergoing the venture.

**Tip:** When doing a feasibility study, it’s always good to have a contingency plan that is ready to test as a viable alternative if the first plan fails.

## **Benefits of a Feasibility Study**

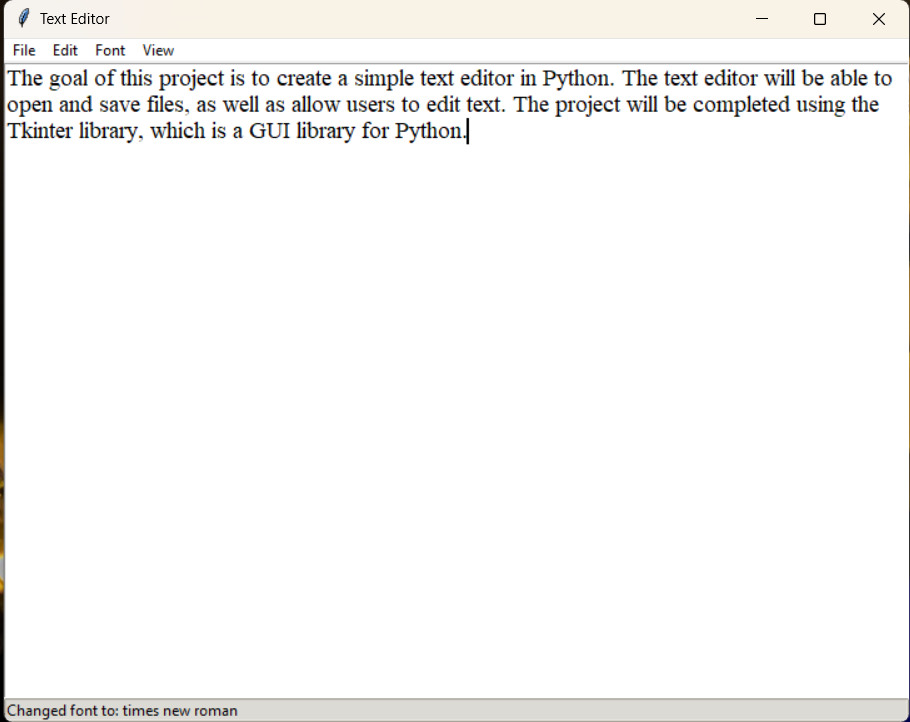
There are several benefits to feasibility studies, including helping project managers discern the pros and cons of undertaking a project before investing a significant amount of time and [capital](https://www.investopedia.com/terms/c/capital.asp) into it.

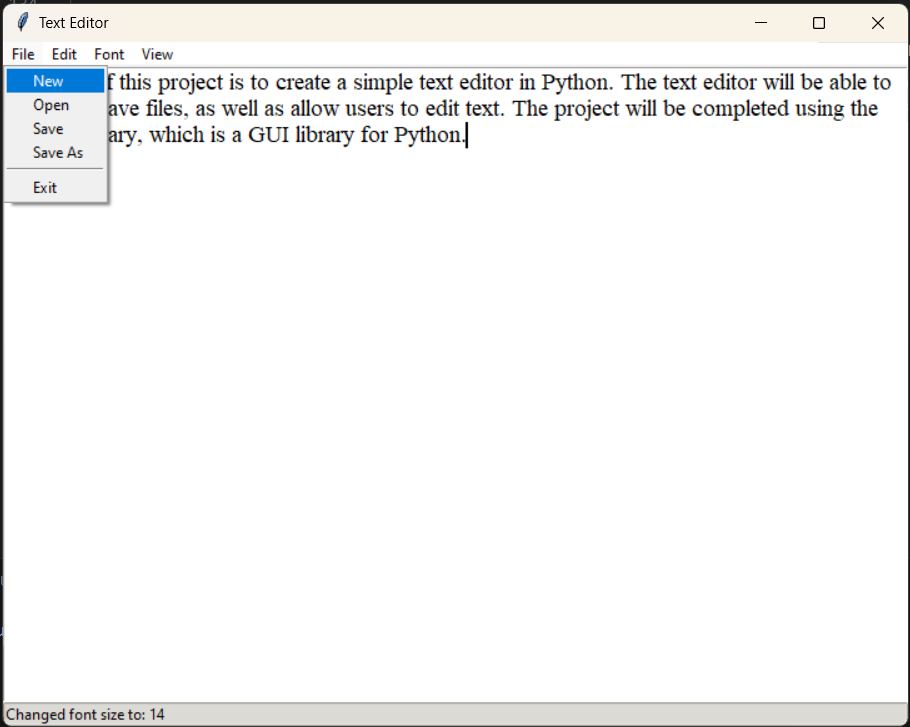
Feasibility studies can also provide a company's management team with crucial information that could prevent them from entering into a [risky](https://www.investopedia.com/terms/r/risk.asp) business venture.

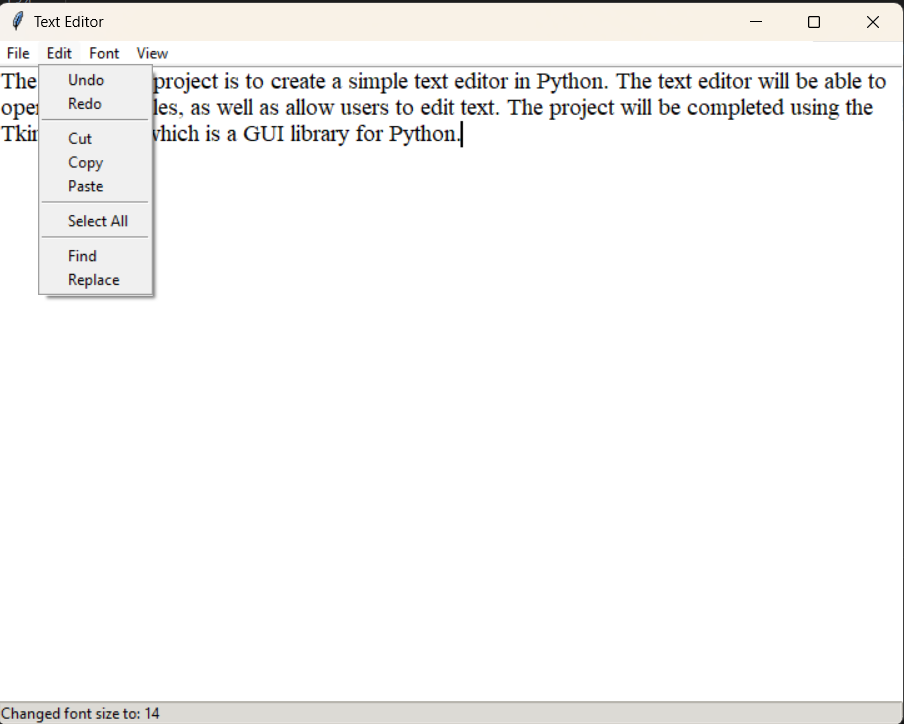
Such studies help companies determine how they will grow. They will know more about how they will operate, what the potential obstacles are, who the competition is, and what the market is.

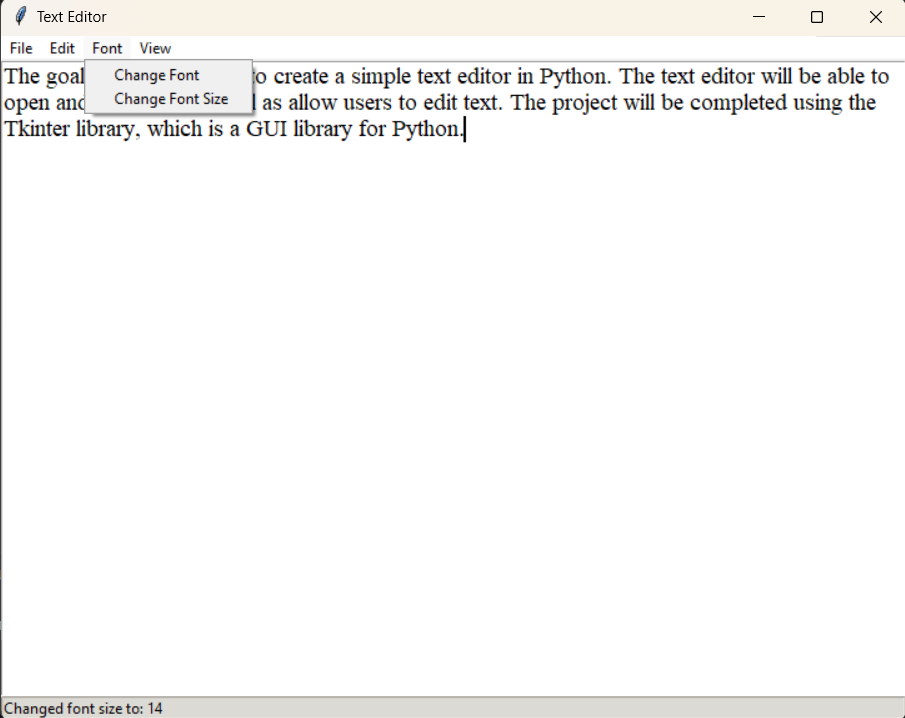
Feasibility studies also help convince investors and bankers that investing in a particular project or business is a wise choice.

**OUTPUT ANALYSIS:**

****

****

****

****

**CONCLUSION:**

The development of the Python text editor using Tkinter has been a rewarding journey, resulting in a functional and user-friendly application. Throughout the project, several key achievements and considerations have come to light.

**Key Achievements**

**User-Friendly Interface:**

The Tkinter-based graphical user interface provides a simple and intuitive platform for users to create, edit, and save text documents.

**Essential Text Editing Features:**

The implementation of fundamental text editing functionalities, such as cut, copy, paste, undo, and redo, ensures a seamless user experience.

**File Operations:**

The text editor supports essential file operations, including creating new files, opening existing ones, saving, and saving as. The integration of file dialogs simplifies the file-handling process.

**Search and Replace Functionality:**

The inclusion of search and replace features enhances the usability of the text editor, allowing users to efficiently find and modify text within documents.

**Font Customization:**

Users can personalize their editing experience by changing the font family and size, adding a layer of customization to the text editor.

**Status Bar and Feedback:**

The status bar provides valuable feedback to users, offering information about ongoing operations and enhancing the overall user experience.

**Challenges and Lessons Learned**

**Font Handling Challenges:**

The implementation of font handling introduced challenges, particularly when dealing with existing fonts and font sizes. Overcoming these challenges required careful consideration of Tkinter's font module.

**User Feedback:**

Incorporating user feedback played a crucial role in refining the application. Future iterations could benefit from additional user testing and feedback loops.

**Future Enhancements**

**Advanced Editing Features:**

Future versions could explore the implementation of more advanced text editing features, such as syntax highlighting, code folding, or line numbering.

**Enhanced Usability:**

Further improvements to the user interface and interaction patterns could enhance the overall usability and accessibility of the text editor.

**Collaborative Editing**:

Exploring features for collaborative editing or integration with version control systems could extend the functionality for users working on collaborative projects.

**Conclusion**

In conclusion, the Python text editor project has achieved its primary goal of providing a simple and functional tool for text editing. The project's success lies in its adherence to usability principles, incorporation of user feedback, and the implementation of essential features. As the project continues to evolve, it holds the potential for further enhancements and contributions to the broader community of Python developers.

Feel free to customize the conclusion based on the specific outcomes and experiences of your project. If there are notable achievements, challenges, or future plans that differ from the template, be sure to highlight them accordingly.

**TEST ANALYSIS / RESULT ANALYSIS:**

The testing phase of the Python text editor project played a crucial role in ensuring the reliability, functionality, and usability of the application. The following sections provide an analysis of the testing process, highlighting key scenarios, results, and potential areas for improvement.

**Testing Scenarios**

**User Interface Testing:**

The graphical user interface (GUI) underwent thorough testing to ensure its responsiveness, clarity, and user-friendly design. Testing included interactions with various screen resolutions and platforms (Windows, macOS, Linux).

**Functional Testing:**

Essential text editing functionalities, including cut, copy, paste, undo, redo, and file operations (open, save, save as), were systematically tested to ensure they performed as expected.

**File Operations Testing:**

File-related operations, such as opening and saving files, were tested with various file types, sizes, and content structures to confirm the text editor's robustness.

**Search and Replace Testing:**

The search and replace functionalities were tested with different text patterns and document sizes to verify the accuracy of the search algorithm and the reliability of the replacement process.

**Font Handling Testing:**

Testing involved changing fonts and font sizes dynamically to assess the text editor's ability to handle font-related configurations without compromising stability.

**Usability Testing:**

Usability testing involved real users interacting with the text editor to identify any potential pain points, confusion, or areas for improvement in terms of user experience.

**Testing Results**

**User Interface:**

The GUI was found to be intuitive and responsive across various platforms. No significant issues were identified in terms of visual appeal or user interaction.

**Functionalities:**

Essential text editing functionalities performed as expected, providing users with a seamless editing experience. Users were able to execute common tasks without encountering critical issues.

**File Operations:**

File operations, including opening, saving, and saving as, were reliable. The file dialogues worked effectively, allowing users to navigate and select files with ease.

**Search and Replace:**

The search and replace functionalities demonstrated accuracy in locating and replacing text instances within documents, contributing to an efficient editing process.

**Font Handling:**

Font handling functionalities were generally successful, although some challenges were encountered when working with specific existing fonts and sizes. Future iterations may address these edge cases.

**Usability:**

Usability testing provided valuable insights into user interactions. Overall, users found the text editor to be straightforward and easy to use. Some minor suggestions were noted for further usability enhancements.

**Areas for Improvement**

**Font Handling Edge Cases:**

Further investigation and testing may be required to address specific edge cases related to font handling, especially when dealing with a wide range of existing fonts and sizes.

**Usability Refinements:**

Implementing minor refinements based on user feedback could further enhance the overall usability of the text editor.

**Conclusion**

The testing phase has been instrumental in validating the reliability and usability of the Python text editor. While the majority of scenarios performed well, identified areas for improvement will be addressed in future iterations to ensure continuous enhancement and user satisfaction.

Adjust the content based on the specific testing scenarios, outcomes, and areas for improvement observed in your project. If there are unique aspects of your testing process or specific challenges encountered, be sure to highlight them in this analysis.

**REFERENCES:**

WEBSITES:-

1. http://wikipedia.org
2. https://www.javatpoint.com
3. https://www.geeksforgeeks.org
4. https://www.programiz.com
5. <https://github.com>

**FUTURE SCOPE:**

The future scope of the Python text editor project using Tkinter is broad and can be extended in various directions to enhance functionality, usability, and compatibility. Here are some potential areas for future development:

**Syntax Highlighting:**

Implement syntax highlighting for different programming languages. This enhancement can improve the user experience for developers working with code in the text editor.

**Code Folding:**

Introduce code folding capabilities to allow users to collapse and expand sections of code. This feature is particularly useful for handling large codebases.

**Line Numbering:**

Include an option for line numbering, aiding users in navigating through documents and locating specific lines quickly.

**Themes and Customization:**

Introduce themes and customization options to allow users to personalize the look and feel of the text editor according to their preferences.

**Advanced Search and Replace:**

Enhance the search and replace functionality to include advanced options, such as case-sensitive searches, regular expressions, and batch replacements.

**Collaborative Editing:**

Explore the possibility of implementing collaborative editing features, enabling multiple users to work on the same document simultaneously. Integration with version control systems like Git could also be considered.

**Spell Checking:**

Integrate spell-checking capabilities to help users identify and correct spelling errors in their documents.

**Accessibility Improvements:**

Implement accessibility features to ensure that the text editor is usable by individuals with different abilities. This includes considerations for screen readers and keyboard navigation.

**File Comparison:**

Introduce a file comparison feature, allowing users to compare two versions of a document side by side.

**Integration with Cloud Storage:**

Implement integration with cloud storage services (e.g., Dropbox, Google Drive) to enable users to save and retrieve documents directly from the cloud.

**Performance Optimization:**

Conduct performance optimization to improve the text editor's responsiveness and reduce load times, especially when handling large files.

**Internationalization (i18n):**

Make the text editor accessible to users from different linguistic backgrounds by introducing internationalization support.

**Cross-Platform Enhancements:**

Ensure continuous compatibility and optimization for different operating systems, addressing any platform-specific challenges or improvements.

**User Feedback Mechanism:**

Establish a mechanism for users to provide feedback directly within the application, facilitating continuous improvement based on user suggestions.

**Documentation and Tutorials:**

Create comprehensive documentation and tutorials to guide users on the features and functionalities of the text editor, fostering a supportive user community.

**Bug Fixes and Stability:**

Address any identified bugs or stability issues to ensure a reliable and error-free user experience.

By incorporating these future enhancements, the Python text editor can evolve into a versatile and feature-rich tool, catering to a broader range of users and use cases. The development roadmap should be guided by user feedback, industry trends, and the project's overall objectives.

**Appendix**

import tkinter as tk

from tkinter import ttk, filedialog, simpledialog, font

class TextEditor:

    def \_\_init\_\_(self, root):

        self.root = root

        self.root.title("Text Editor")

        # Styling

        self.style = ttk.Style()

        self.style.theme\_use("clam")  # Choose a theme (aqua, clam, alt, default, etc.)

        # Text Widget

        self.text\_widget = tk.Text(self.root, wrap="word", undo=True, font=("Helvetica", 12))

        self.text\_widget.pack(expand="yes", fill="both")

        # Menu Bar

        self.menu\_bar = tk.Menu(root)

        self.root.config(menu=self.menu\_bar)

        # File Menu

        self.file\_menu = tk.Menu(self.menu\_bar, tearoff=0)

        self.menu\_bar.add\_cascade(label="File", menu=self.file\_menu)

        self.file\_menu.add\_command(label="New", command=self.new\_file)

        self.file\_menu.add\_command(label="Open", command=self.open\_file)

        self.file\_menu.add\_command(label="Save", command=self.save\_file)

        self.file\_menu.add\_command(label="Save As", command=self.save\_as\_file)

        self.file\_menu.add\_separator()

        self.file\_menu.add\_command(label="Exit", command=self.root.destroy)

        # Edit Menu

        self.edit\_menu = tk.Menu(self.menu\_bar, tearoff=0)

        self.menu\_bar.add\_cascade(label="Edit", menu=self.edit\_menu)

        self.edit\_menu.add\_command(label="Undo", command=self.text\_widget.edit\_undo)

        self.edit\_menu.add\_command(label="Redo", command=self.text\_widget.edit\_redo)

        self.edit\_menu.add\_separator()

        self.edit\_menu.add\_command(label="Cut", command=self.cut)

        self.edit\_menu.add\_command(label="Copy", command=self.copy)

        self.edit\_menu.add\_command(label="Paste", command=self.paste)

        self.edit\_menu.add\_separator()

        self.edit\_menu.add\_command(label="Select All", command=self.select\_all)

        self.edit\_menu.add\_separator()

        self.edit\_menu.add\_command(label="Find", command=self.find\_text)

        self.edit\_menu.add\_command(label="Replace", command=self.replace\_text)

        # Font Menu

        self.font\_menu = tk.Menu(self.menu\_bar, tearoff=0)

        self.menu\_bar.add\_cascade(label="Font", menu=self.font\_menu)

        self.font\_menu.add\_command(label="Change Font", command=self.change\_font)

        self.font\_menu.add\_command(label="Change Font Size", command=self.change\_font\_size)

        # View Menu

        self.view\_menu = tk.Menu(self.menu\_bar, tearoff=0)

        self.menu\_bar.add\_cascade(label="View", menu=self.view\_menu)

        self.view\_menu.add\_command(label="Toggle Status Bar", command=self.toggle\_status\_bar)

        # Status Bar

        self.status\_var = tk.StringVar()

        self.status\_bar = ttk.Label(root, textvariable=self.status\_var, anchor=tk.W, relief=tk.SUNKEN)

        self.status\_bar.pack(side=tk.BOTTOM, fill=tk.X)

        # Default status

        self.status\_var.set("Ready")

    def new\_file(self):

        self.text\_widget.delete("1.0", tk.END)

        self.root.title("Text Editor")

        self.update\_status("New file created")

    def open\_file(self):

        file\_path = filedialog.askopenfilename(defaultextension=".txt", filetypes=[("Text Files", "\*.txt"), ("All Files", "\*.\*")])

        if file\_path:

            with open(file\_path, "r") as file:

                file\_content = file.read()

                self.text\_widget.delete("1.0", tk.END)

                self.text\_widget.insert(tk.END, file\_content)

            self.root.title(f"Text Editor - {file\_path}")

            self.update\_status(f"Opened file: {file\_path}")

    def save\_file(self):

        file\_path = filedialog.asksaveasfilename(defaultextension=".txt", filetypes=[("Text Files", "\*.txt"), ("All Files", "\*.\*")])

        if file\_path:

            with open(file\_path, "w") as file:

                file.write(self.text\_widget.get("1.0", tk.END))

            self.root.title(f"Text Editor - {file\_path}")

            self.update\_status(f"Saved file: {file\_path}")

    def save\_as\_file(self):

        self.save\_file()

    def cut(self):

        self.text\_widget.event\_generate("<<Cut>>")

        self.update\_status("Cut")

    def copy(self):

        self.text\_widget.event\_generate("<<Copy>>")

        self.update\_status("Copy")

    def paste(self):

        self.text\_widget.event\_generate("<<Paste>>")

        self.update\_status("Paste")

    def select\_all(self):

        self.text\_widget.tag\_add(tk.SEL, "1.0", tk.END)

        self.text\_widget.mark\_set(tk.SEL\_FIRST, "1.0")

        self.text\_widget.mark\_set(tk.SEL\_LAST, tk.END)

        self.text\_widget.see(tk.SEL\_FIRST)

        self.update\_status("Selected all")

    def find\_text(self):

        query = simpledialog.askstring("Find", "Enter text to find:")

        if query:

            start\_pos = "1.0"

            while True:

                start\_pos = self.text\_widget.search(query, start\_pos, stopindex=tk.END)

                if not start\_pos:

                    break

                end\_pos = f"{start\_pos}+{len(query)}c"

                self.text\_widget.tag\_add(tk.SEL, start\_pos, end\_pos)

                start\_pos = end\_pos

            self.text\_widget.tag\_configure("sel", background="yellow")

            self.update\_status(f"Found occurrences of: {query}")

    def replace\_text(self):

        find\_query = simpledialog.askstring("Find", "Enter text to find:")

        if find\_query:

            replace\_query = simpledialog.askstring("Replace", f"Replace '{find\_query}' with:")

            if replace\_query:

                content = self.text\_widget.get("1.0", tk.END)

                updated\_content = content.replace(find\_query, replace\_query)

                self.text\_widget.delete("1.0", tk.END)

                self.text\_widget.insert(tk.END, updated\_content)

                self.update\_status(f"Replaced occurrences of: {find\_query}")

    def change\_font(self):

        font\_name = simpledialog.askstring("Font", "Enter font (e.g., Arial, Times New Roman):")

        if font\_name:

            current\_font = font.Font(self.text\_widget, self.text\_widget.cget("font"))

            current\_font.configure(family=font\_name)

            self.text\_widget.configure(font=current\_font)

            self.update\_status(f"Changed font to: {font\_name}")

    def change\_font\_size(self):

        size = simpledialog.askinteger("Font Size", "Enter font size:")

        if size:

            current\_font = font.Font(self.text\_widget, self.text\_widget.cget("font"))

            current\_font.configure(size=size)

            self.text\_widget.configure(font=current\_font)

            self.update\_status(f"Changed font size to: {size}")

    def toggle\_status\_bar(self):

        if self.status\_bar.winfo\_ismapped():

            self.status\_bar.pack\_forget()

            self.update\_status("Status Bar hidden")

        else:

            self.status\_bar.pack(side=tk.BOTTOM, fill=tk.X)

            self.update\_status("Status Bar shown")

    def update\_status(self, message):

        self.status\_var.set(message)

if \_\_name\_\_ == "\_\_main\_\_":

    root = tk.Tk()

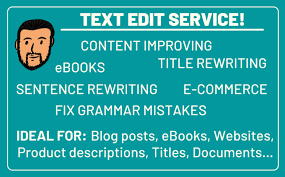
    app = TextEditor(root)

    root.mainloop()

**IMAGES:**

****

****

****

**"The process of editing is what I enjoy most - putting the pieces together and making sense out of them"**