An Interactive Syntax Highlighting Editor in Python with Dynamic Theme and Autocomplete Features

Yateeka Goyal
Department of Computer Science
Georgia State University
Email: ygoyal2@student.gsu.edu

***Abstract*—This paper presents a lightweight, extensible Python-based text editor that includes real-time syntax highlighting, customizable themes, bracket matching, autocomplete functionality, and a built-in line number display. The editor is developed using Tkinter, making it highly accessible and ideal for educational and light development purposes. This project demonstrates a strong understanding of GUI programming, lexical parsing, and the Python language's dynamic capabilities. The complete source code and additional resources for this project are available at: https://github.com/Yateeka/PLC.**

I. Introduction

Modern integrated development environments (IDEs) often include syntax highlighting and intelligent features that assist developers in code readability and debugging. However, such environments may be resource-intensive and are often inaccessible to beginners who require simplicity and educational guidance. This project addresses that gap by creating a syntax highlighting editor using Python's Tkinter module. The application is tailored for beginner-friendly usage while incorporating advanced features like theme switching, autocomplete for Python keywords, and live status updates.

While comprehensive IDEs provide sophisticated capabilities, they often come with steep learning curves and heavy system requirements. This can be a deterrent for new programmers, especially those learning Python in academic settings or on low-powered devices. The proposed editor focuses on essential development support tools within a single, easy-to-use interface. The lightweight nature of the editor ensures it can run seamlessly on nearly any system that supports Python, making it ideal for students, educators, and hobbyists. Moreover, by using native libraries only, the editor avoids dependency-related installation issues, promoting portability and minimalism. This project aims to replicate key functionality from full-featured editors, such as dynamic syntax highlighting and intuitive theming, in a condensed and educationally rich environment that encourages experimentation and learning. It also introduces users to concepts like real-time event handling, lexical token matching, and GUI layout management without overwhelming them with advanced configuration or setup steps.

II. Problem Definition and Complexity

The primary goal of this project is to design and implement a text editor that offers core functionalities of an IDE while remaining lightweight and simple. It involves:

- Real-time syntax highlighting using regular expressions.
- Implementation of bracket matching.
- Autocomplete window for Python keywords.
- Support for multiple visual themes.
- Line number synchronization.

This problem is non-trivial, especially considering that standard GUI libraries like Tkinter are not inherently designed for advanced text-editing features. Tkinter, by default, provides basic text widgets that lack native support for syntax-aware features. To overcome this, the editor integrates event-driven bindings with regex-based token recognition to dynamically apply syntax coloring in real time.

A core challenge lies in creating a responsive experience without relying on external libraries like Pygments or Scintilla. For instance, to highlight Python keywords and built-in functions, the editor tokenizes the content of the text widget after each keypress and applies style tags. Similarly, line numbers are dynamically rendered in a separate Text widget and kept synchronized with the main editor window through scroll and key events.

Additionally, implementing autocomplete within a pure-Tkinter interface requires manually constructing a popup window, positioning it relative to the cursor, and binding it to both keyboard and mouse events for user interaction. Bracket matching logic must account for nested contexts and prevent interference with user typing flow.

If needed, a visual architecture diagram can be added here to illustrate the interplay between components like the text area, highlighter, theme manager, line numbers, and autocomplete popup.

III. System Design and Architecture

The application is designed around a modular and event-driven architecture. Each component interacts through well-defined bindings and shared state, allowing flexibility and extensibility. The main components include:

- **TextEditor Class:** Acts as the controller for the entire application. It initializes the layout, configures bindings, manages themes, status bar updates, and integrates each subcomponent into the user interface.
- **ScrolledText Widget:** The primary code editing area, built using a 'ScrolledText' widget from Tkinter. It supports undo, redo, and keyboard events, and is where syntax highlighting is applied.

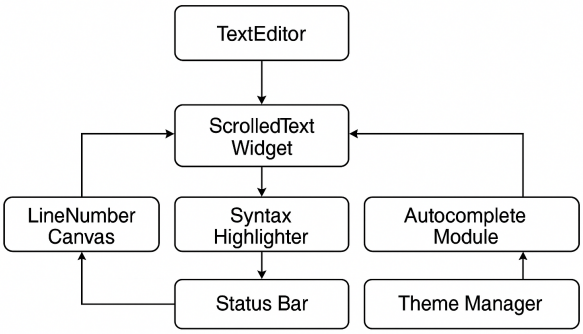

Figure 1 Architecture Diagram of the Editor

Fig. 1: System architecture showing key components of the editor

- **LineNumberCanvas:** A specialized 'Text' widget used exclusively for displaying line numbers. It dynamically adjusts as the user types or scrolls, maintaining alignment with the main editor.
- **Syntax Highlighter:** Uses regular expressions to identify patterns for Python syntax (e.g., keywords, strings, numbers, comments). The system highlights these elements in real-time by applying tags with corresponding colors.
- **Autocomplete Module:** A dynamic popup that displays keyword suggestions based on the current cursor context. It listens for keyboard input and presents suggestions that match partially typed Python keywords.
- **Theme Manager:** Controls the background, text, insertion cursor, and line number colors. Themes such as "Dracula," "Nord," and "Solarized" are predefined and can be switched through the menu or with a keyboard shortcut.
- **Status Bar:** Continuously shows the current cursor position (line and column). It updates with every keypress or mouse click, helping users track where they are in the document.

Each module is self-contained but communicates with others through events and shared variables. This makes the application robust and adaptable to additional features, such as multi-tab support or debugging panes. The architecture promotes ease of debugging, understanding, and future scalability.

IV. Implementation and Functionality

The implementation of the syntax highlighting editor is carried out in a modular and structured manner, with several classes and functions interacting to provide a seamless experience. At the core is the **TextEditor** class, which serves as the application controller. This class initializes the interface, binds events, manages theme switching, and coordinates with all other components of the editor.

Syntax highlighting is achieved through regular expression-based parsing. The editor scans the content of the text widget in real time and assigns visual tags to different syntactic elements such as keywords, strings, comments, numbers, and booleans. These tags define font colors that help the user visually distinguish between different code components. Each pattern is defined through regex and matched against the input buffer whenever the text is modified.

The **apply_syntax_highlighting** function is responsible for applying these patterns. It first clears any existing tags and then re-applies them based on the current contents. This includes differentiation between single-line strings, multi-line strings, numbers, logical constants (like `True` and `False`), `None`, built-in functions, and collection brackets. The function also distinguishes Python operators and highlights them accordingly.

A separate class, **LineNumberCanvas**, implements the display of line numbers using another `Text` widget. It synchronizes scrolling with the main text area and updates line numbers dynamically based on content changes.

To provide contextual help during coding, the **handle_autocomplete** function detects partially typed Python keywords and displays them in a floating `Listbox`. This is rendered in a popup `Toplevel` widget which appears just below the cursor. If a user selects a suggestion using the keyboard or mouse, it is inserted into the text.

Theme support is implemented using a dictionary where each theme defines attributes such as background color, text color, insertion point color, and line number color. The **apply_theme** function dynamically applies the selected theme by configuring the visual attributes of the text widget and line number canvas.

The bracket matching feature enhances usability by automatically inserting the corresponding closing bracket when an opening bracket is typed. The implementation is event-driven and modifies the text buffer while preserving the cursor position.

Other features include file operations (open/save), undo/redo, keyboard shortcuts, and a dynamic status bar that displays the current cursor position. All these components come together under the main window created by the **TextEditor** class, which integrates Tkinter widgets and binds them to appropriate methods for functionality.

The architecture promotes real-time interaction, responsiveness, and ease of use, making it ideal for both educational environments and simple development workflows.

V. Conceptual Understanding

The project demonstrates a deep understanding of several foundational programming and software design concepts critical to the construction of interactive applications.

- **GUI Programming with Event-Driven Logic:** The editor relies heavily on event bindings in Tkinter to handle real-time user inputs such as keypresses, mouse movements, and file operations. Event-driven programming ensures a responsive user interface where every interaction dynamically updates the application's state.
- **Regular Expressions for Token Classification:** Syntax highlighting is accomplished through the strategic use of regular expressions, which parse and identify key

syntactic elements in the text buffer. This demonstrates a strong grasp of pattern matching and dynamic text processing.

- **Object-Oriented Programming for Modularity:** Each major feature, such as line numbering, autocomplete, and theming, is encapsulated in its own class or modular function. This use of OOP principles ensures that the code remains organized, reusable, and maintainable.
- **Data Structures such as Dictionaries and Lists:** Theme management, autocomplete suggestions, and syntax mapping utilize Python's dictionaries and lists to efficiently organize and retrieve data.

VI. Innovation and Creativity

The editor introduces several unique features that go beyond standard text editing applications:

- **Bracket-Matching Mechanism Integrated with Syntax Coloring:** When a user types an opening bracket, the corresponding closing bracket is inserted automatically. This significantly enhances the coding experience and reduces syntax errors.
- **Live Theme Switcher Between 7 Different Schemes:** Users can switch between multiple professionally designed themes, including Dark Mode, Dracula, Nord, and Solarized, without restarting the application.
- **Autocomplete Suggestions for Python Keywords:** The editor suggests completions for partially typed keywords in a floating window, improving coding speed and reducing errors.
- **Informative Sample Code Snippet on Startup:** Upon launching, the editor loads a fully formatted Python script demonstrating syntax highlighting, allowing users to immediately understand the tool's capabilities.

VII. Documentation

The project is thoroughly documented both in-line and externally:

- Each complex function, such as syntax highlighting and autocomplete handling, is accompanied by detailed comments explaining the internal logic.
- A built-in Help Menu within the application provides users with keyboard shortcuts and feature descriptions.
- This report acts as a complete external documentation manual, describing the architecture, design decisions, and usage instructions.

VIII. Presentation and Demonstration

During the demonstration, each key feature of the editor was showcased with real-time interactions and supporting screenshots:

Real-Time Syntax Highlighting

Initially, the editor loads plain text without any highlighting. As the user types or opens a Python file, syntax elements such as keywords, built-ins, comments, and numbers are dynamically color-coded.

(a) Before highlighting

(b) After highlighting

Fig. 2: Editor demonstrating real-time syntax highlighting

Bracket Matching

When an opening parenthesis, square bracket, or curly brace is typed, the corresponding closing bracket is automatically inserted.

Fig. 3: Automatic insertion of closing brackets in action.

Theme Switching

Users demonstrated toggling between different themes such as ”Light” and ”Dracula” to customize their editing environment.

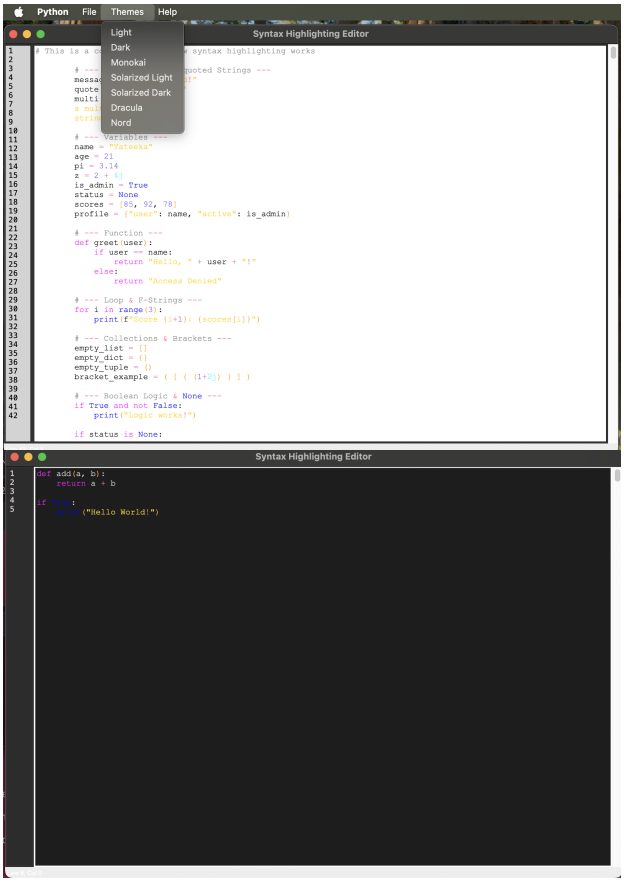

Fig. 4: Theme switching demonstration between Light and Dracula themes.

Autocomplete Functionality

Typing part of a Python keyword triggers a floating autocomplete window offering relevant suggestions.

Fig. 5: Autocomplete suggestions triggered by partial keyword typing.

Status Bar and Line Numbers

The editor dynamically updates the status bar with the current line and column, and synchronized line numbering provides improved navigation.

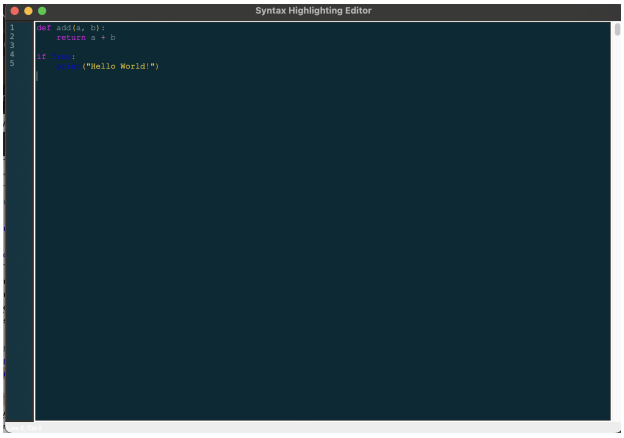

Fig. 6: Real-time line and column tracking with synchronized line numbers.

File Operations

Demonstrations included successfully opening and saving Python script files using the built-in file dialog operations.

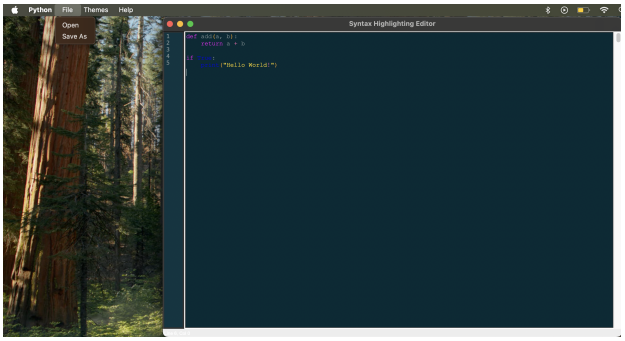

Fig. 7: Opening and saving files within the editor.

IX. Future Work

Looking ahead, the syntax highlighting editor can be extended in several ways to enhance its functionality and user experience. One promising direction is to support additional programming languages such as JavaScript and HTML, allowing the editor to serve a broader range of developers. Another enhancement would be to integrate a built-in terminal or Python execution console, enabling users to run their code directly from within the editor. Supporting multiple file editing through a tabbed interface would also improve workflow efficiency. Finally, incorporating AI-powered code suggestions could make the editor even smarter by providing real-time assistance based on context, further helping users write and debug code more efficiently.

X. Conclusion

This project successfully showcases the application of both core and advanced programming principles to build a user-friendly, feature-rich text editor. It is a functional prototype that can be extended to support additional languages, debugging features, or plug-in capabilities. The editor reflects a strong grasp of programming paradigms, real-time event processing, dynamic text manipulation, and graphical user interface design. Through thoughtful architecture and careful implementation, the project achieves its goal of delivering an educational, lightweight, and interactive development tool.