Criterion C

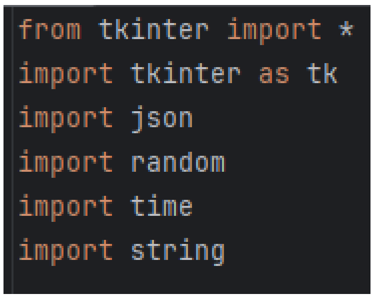
Programming Techniques & Tools

The project was developed using **Python**, chosen for its simplicity and extensive libraries. The **Tkinter** library was used to create the GUI, while **JSON** was used for storing user credentials and word libraries. The core functionalities included:

- **User authentication** with login/registration
- **Random word generation** using a unique seed system
- **Solver logic** to determine correct word placements
- **GUI implementation** for word input and feedback
- **Additional features** like a **timer and dark mode toggle**

The program utilizes several libraries to ensure functionality (*Figure 1*):

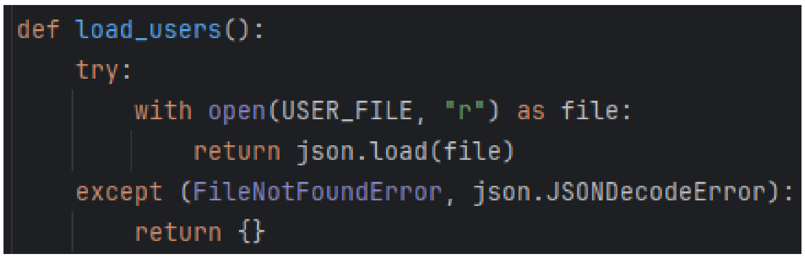
Tkinter is used for creating the graphical user interface, JSON handles the storage of user and game data, Random and String facilitate the generation of game seeds, and Time is employed to track player performance.



```python
from tkinter import *
import tkinter as tk
import json
import random
import time
import string
```

Figure 1

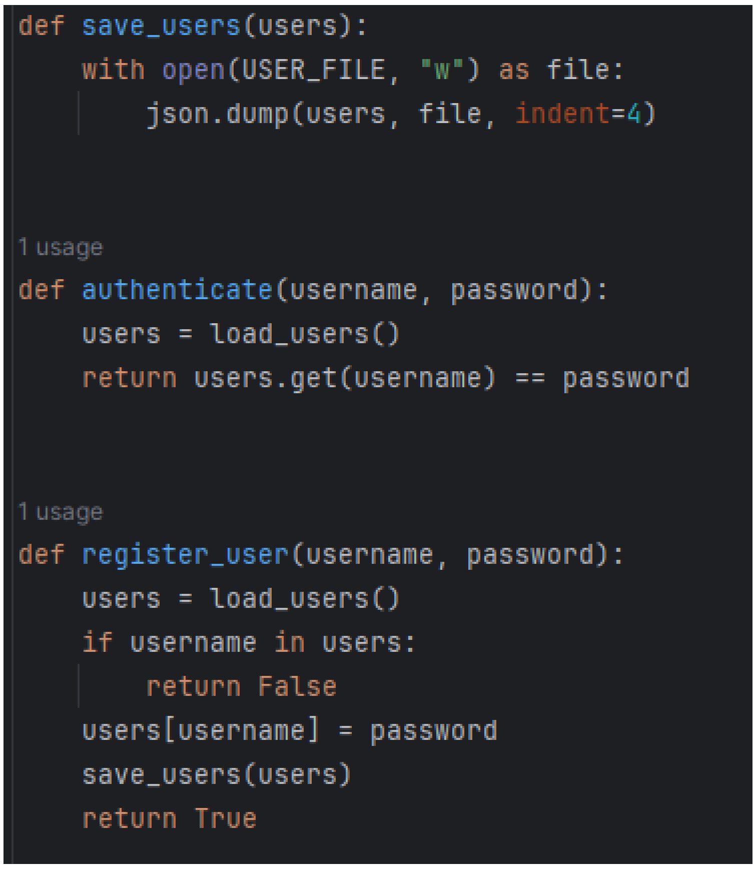
The user authentication system enables users to register and log in. This is handled through a JSON file (users.json), which stores usernames and their associated passwords. The system first attempts to load the user database (*Figure 2*), handling potential errors such as missing files or corrupted data.



```python
def load_users():
    try:
        with open(USER_FILE, "r") as file:
            return json.load(file)
    except (FileNotFoundError, json.JSONDecodeError):
        return {}
```

Figure 2

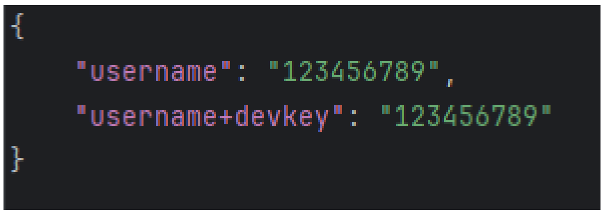
When a user registers, their username is checked against existing entries to prevent duplicates, and new credentials are added to the database. Logging in verifies if the provided credentials match the stored values. (*Figure 3*)



```python
def save_users(users):
    with open(USER_FILE, "w") as file:
        json.dump(users, file, indent=4)


1 usage
def authenticate(username, password):
    users = load_users()
    return users.get(username) == password


1 usage
def register_user(username, password):
    users = load_users()
    if username in users:
        return False
    users[username] = password
    save_users(users)
    return True
```

Figure 3

Developers have a unique identifier, the +devkey suffix, which grants them additional privileges within the software. (*Figure 4*)



```json
{
    "username": "123456789",
    "username+devkey": "123456789"
}
```

Figure 4

Game generation is driven by a seed-based system that ensures each unique set of words corresponds to a specific identifier. The total number of possible word selections is immense, requiring a compact method to encode each game uniquely. Given that there are 2,309 accepted words and 16 words are selected per game, the number of possible unique games is extraordinarily large. To encode every possible variation efficiently, a 179-bit number would be required, which is too long for practical use. Instead, a Base32 encoding system (using characters A-V and digits 0-9) reduces this to a more manageable 36-character seed. This ensures that every possible game state can be quickly identified while keeping the seed length reasonable. (*Figure 5, 6*)

Binary (179 Characters):

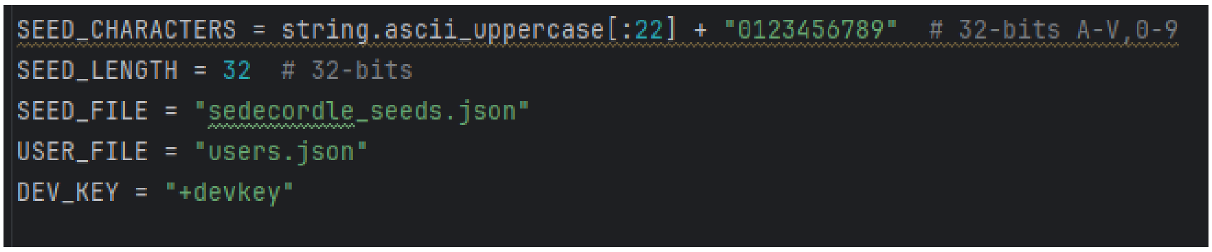
> 1011001101001101110010001111100101000001100011011101110100111100110111011001111111000000111001011100010001101011110111010001101010101000011010110100000111011010100101101010101000010101

Hexadecimal (45 Characters):

> 59A6E47CA0C6EE9E6ECFC1CB88D7BA3550D6876A5AA15

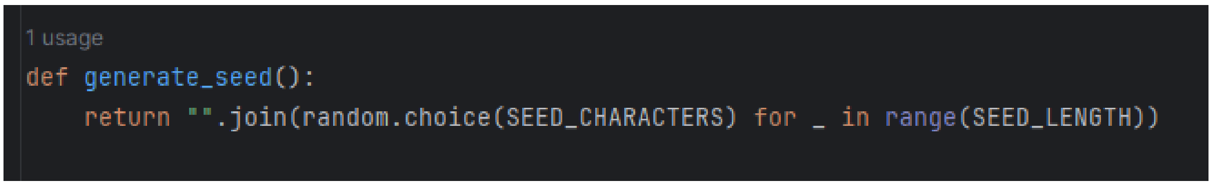
Base32 (36 Characters):

> B6JE8V50ORN9SRMFO75OHLTQ6L8DD1RABAGL



```
SEED_CHARACTERS = string.ascii_uppercase[:22] + "0123456789"  # 32-bits A-V,0-9
SEED_LENGTH = 32  # 32-bits
SEED_FILE = "sedecordle_seeds.json"
USER_FILE = "users.json"
DEV_KEY = "+devkey"
```

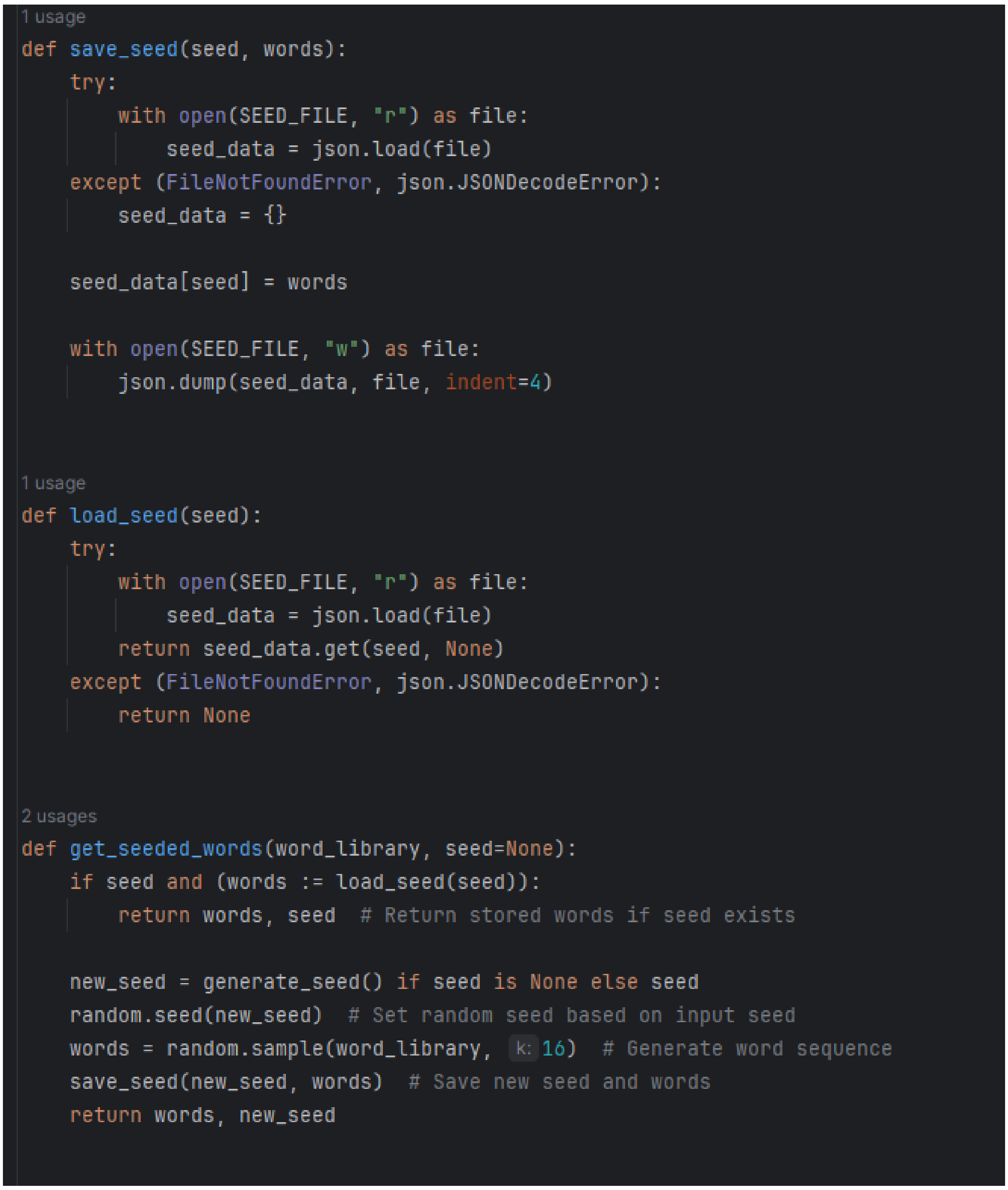
Figure 5



```
def generate_seed():
    return "".join(random.choice(SEED_CHARACTERS) for _ in range(SEED_LENGTH))
```

Figure 6

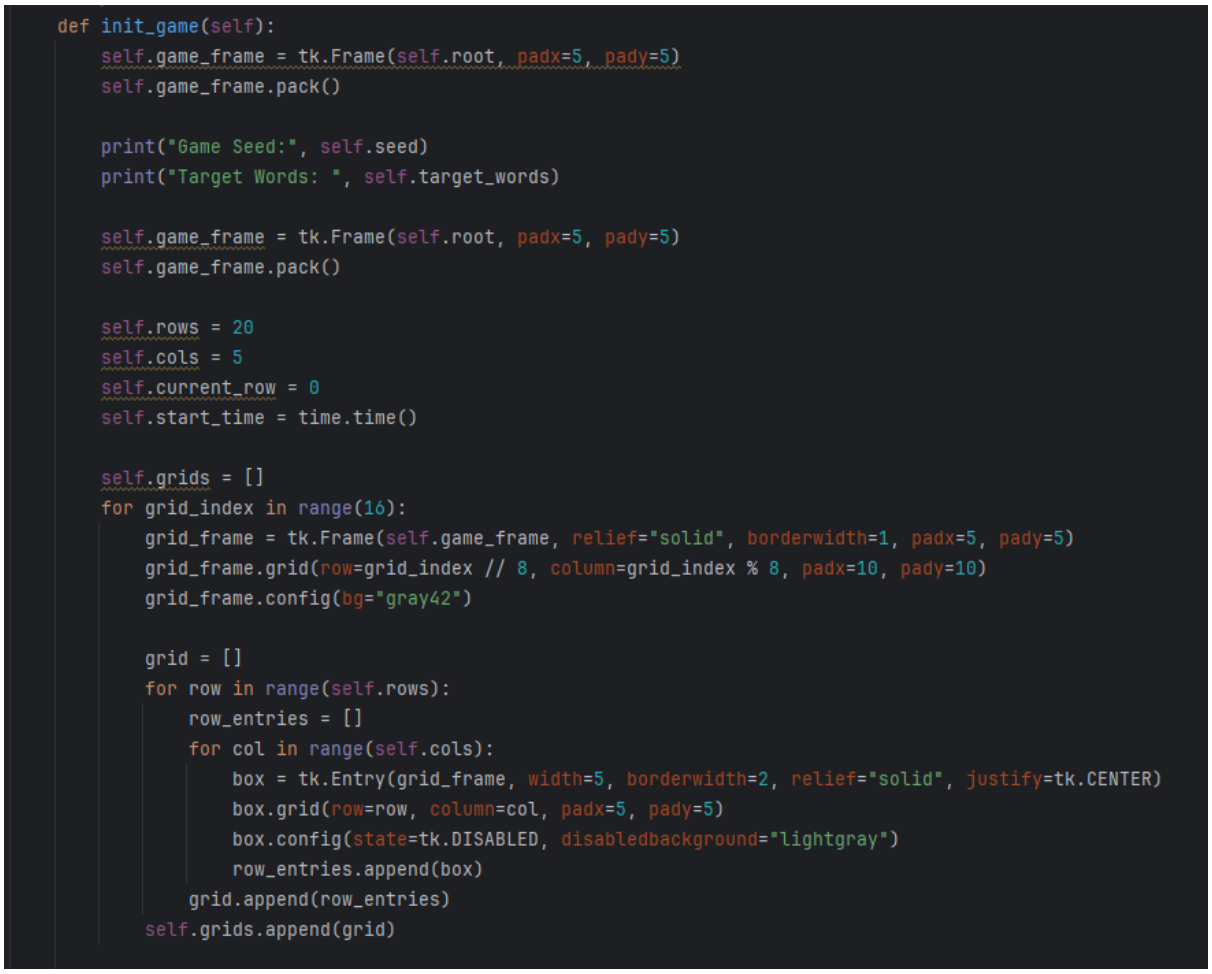
The seed is generated by randomly selecting characters from the Base32 character set, ensuring uniform distribution and a high degree of randomness (*Figure 5*). When generating a new game, the system first checks if the provided seed has been previously used by attempting to load stored word selections from the sedecordle_seeds.json file. If the seed already exists, the corresponding words are retrieved to maintain consistency. If the seed is new, 16 words are randomly sampled from the word library, saved, and associated with the seed. The randomization process ensures fairness, and the structure of the word list remains unchanged for a given seed. (*Figure 7*)



```
1 usage
def save_seed(seed, words):
    try:
        with open(SEED_FILE, "r") as file:
            seed_data = json.load(file)
    except (FileNotFoundError, json.JSONDecodeError):
        seed_data = {}

    seed_data[seed] = words

    with open(SEED_FILE, "w") as file:
        json.dump(seed_data, file, indent=4)


1 usage
def load_seed(seed):
    try:
        with open(SEED_FILE, "r") as file:
            seed_data = json.load(file)
        return seed_data.get(seed, None)
    except (FileNotFoundError, json.JSONDecodeError):
        return None


2 usages
def get_seeded_words(word_library, seed=None):
    if seed and (words := load_seed(seed)):
        return words, seed  # Return stored words if seed exists

    new_seed = generate_seed() if seed is None else seed
    random.seed(new_seed)  # Set random seed based on input seed
    words = random.sample(word_library, k: 16)  # Generate word sequence
    save_seed(new_seed, words)  # Save new seed and words
    return words, new_seed
```

Figure 7

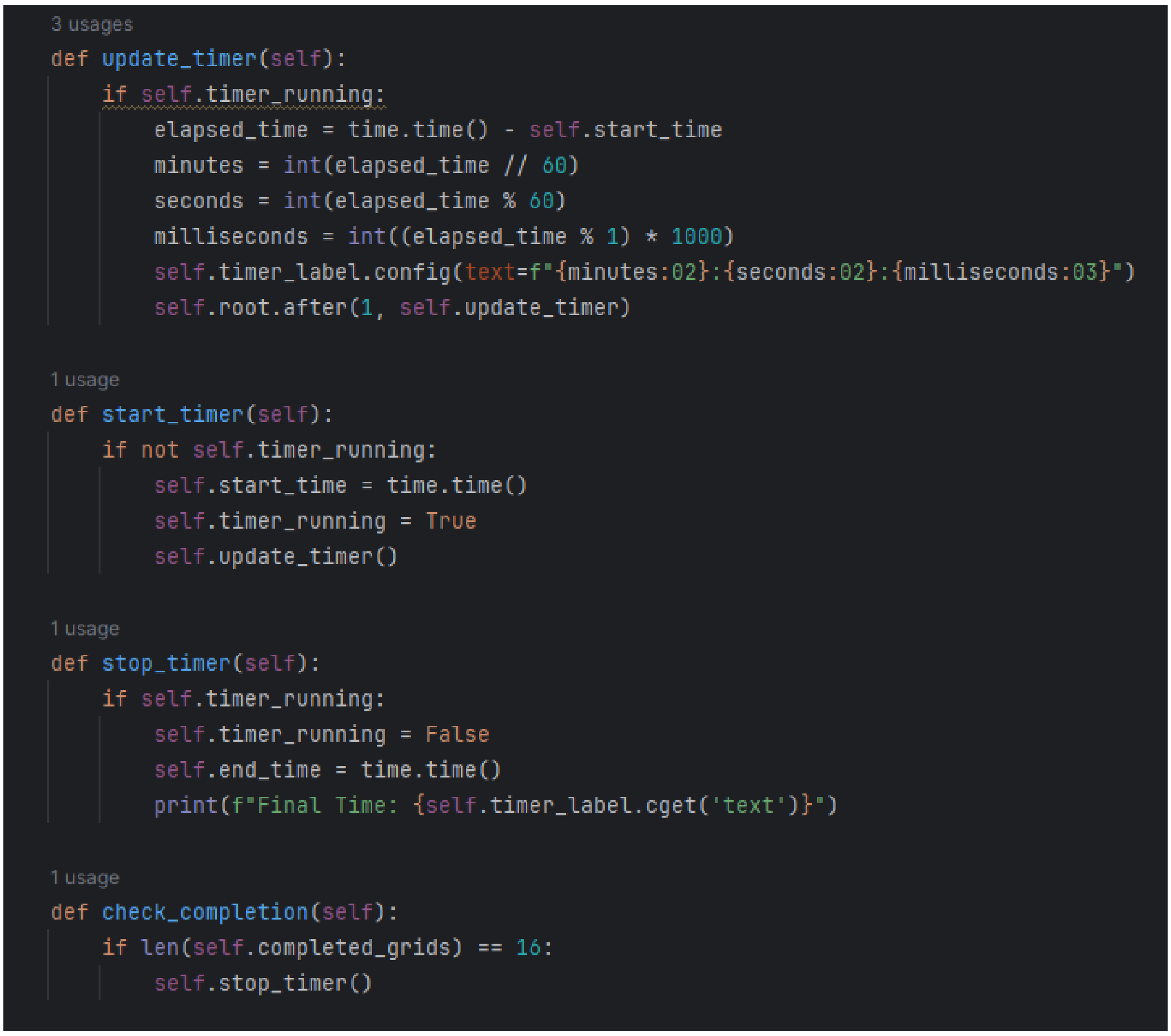
The graphical user interface (GUI) was built using Tkinter, with a structured layout to ensure usability. Upon startup, the login screen is displayed, prompting users to enter their credentials or register a new account. After successful authentication, the game initializes with multiple grids, each corresponding to an active word. The layout consists of entry boxes where users input their guesses, with each entry dynamically updating to reflect correctness. The system disables unused entry fields to prevent unnecessary inputs, guiding the user through the solving process. (*Figure 8*)



```python
def init_game(self):
    self.game_frame = tk.Frame(self.root, padx=5, pady=5)
    self.game_frame.pack()

    print("Game Seed:", self.seed)
    print("Target Words: ", self.target_words)

    self.game_frame = tk.Frame(self.root, padx=5, pady=5)
    self.game_frame.pack()

    self.rows = 20
    self.cols = 5
    self.current_row = 0
    self.start_time = time.time()

    self.grids = []
    for grid_index in range(16):
        grid_frame = tk.Frame(self.game_frame, relief="solid", borderwidth=1, padx=5, pady=5)
        grid_frame.grid(row=grid_index // 8, column=grid_index % 8, padx=10, pady=10)
        grid_frame.config(bg="gray42")

        grid = []
        for row in range(self.rows):
            row_entries = []
            for col in range(self.cols):
                box = tk.Entry(grid_frame, width=5, borderwidth=2, relief="solid", justify=tk.CENTER)
                box.grid(row=row, column=col, padx=5, pady=5)
                box.config(state=tk.DISABLED, disabledbackground="lightgray")
                row_entries.append(box)
            grid.append(row_entries)
        self.grids.append(grid)
```

Figure 8

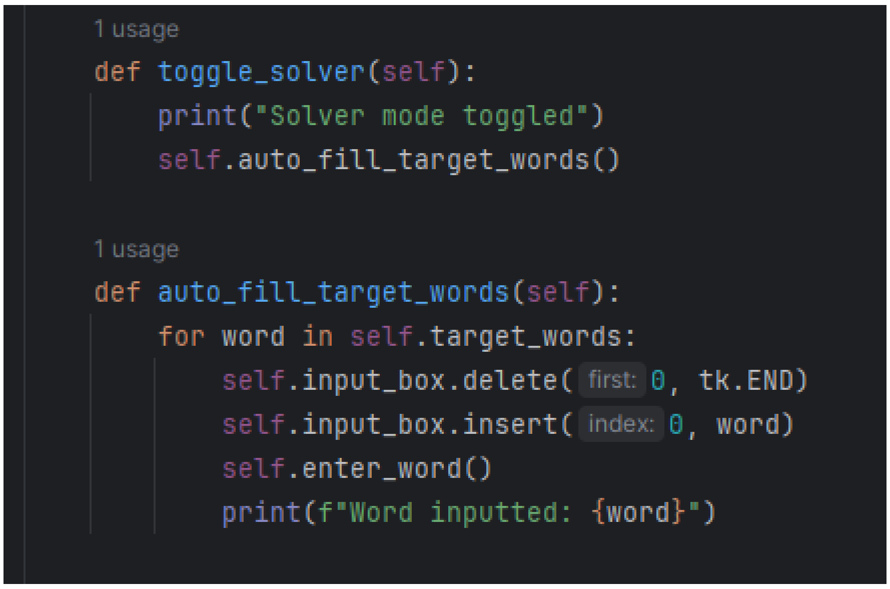
A timer tracks the user's progress throughout the game. When the first word is entered, the timer begins counting in minutes, seconds, and milliseconds. The timer updates in real-time, ensuring an accurate measurement of performance. Once all words have been correctly identified, the timer stops, and the final time is displayed. This adds an element of challenge and progression tracking to the game. (*Figure 9*)



```
3 usages
def update_timer(self):
    if self.timer_running:
        elapsed_time = time.time() - self.start_time
        minutes = int(elapsed_time // 60)
        seconds = int(elapsed_time % 60)
        milliseconds = int((elapsed_time % 1) * 1000)
        self.timer_label.config(text=f"{minutes:02}:{seconds:02}:{milliseconds:03}")
        self.root.after(1, self.update_timer)

1 usage
def start_timer(self):
    if not self.timer_running:
        self.start_time = time.time()
        self.timer_running = True
        self.update_timer()

1 usage
def stop_timer(self):
    if self.timer_running:
        self.timer_running = False
        self.end_time = time.time()
        print(f"Final Time: {self.timer_label.cget('text')}")

1 usage
def check_completion(self):
    if len(self.completed_grids) == 16:
        self.stop_timer()
```

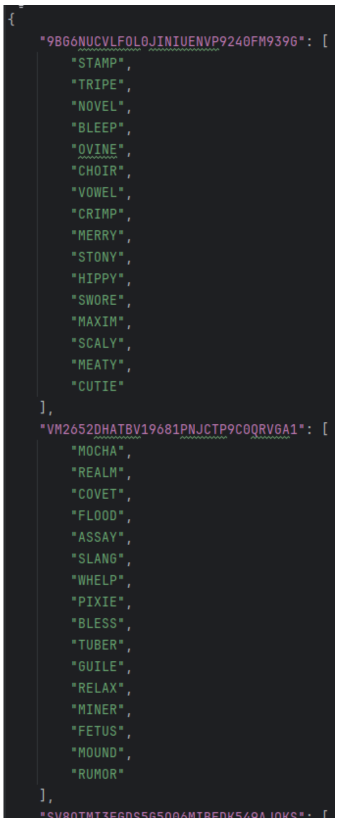
Figure 9

The game includes a solver mode that instantly fills in the correct words, providing immediate solutions. This feature is exclusively accessible to clients with a developer key, ensuring restricted use. The solver mode is implemented through a dedicated button within the game interface, which only appears for developer accounts, as seen in the conditional logic that checks self.is_developer before rendering the button (*Figure 11*). When activated, the solver function sequentially inputs all target words into the game, automating the process through the auto_fill_target_words method. This method iterates through self.target_words, clearing the input field, inserting the correct word, and triggering the input event, effectively bypassing the manual entry process. The game stores these words using unique seed-based identifiers in JSON format, mapping each seed to a predefined set of 16 words, ensuring consistency across sessions(*Figure 12*). By leveraging this structure, the solver provides an efficient, rapid way to retrieve correct words instantly while maintaining the integrity of the word-generation system.



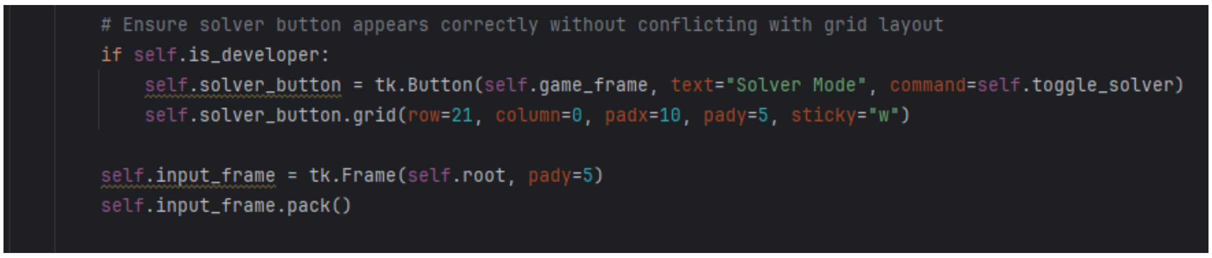
```python
1 usage
def toggle_solver(self):
    print("Solver mode toggled")
    self.auto_fill_target_words()

1 usage
def auto_fill_target_words(self):
    for word in self.target_words:
        self.input_box.delete( first: 0, tk.END)
        self.input_box.insert( index: 0, word)
        self.enter_word()
        print(f"Word inputted: {word}")
```

Figure 11



```json
{
    "9BG6NUCVLFOL0JINIUENVP9240FM939G": [
        "STAMP",
        "TRIPE",
        "NOVEL",
        "BLEEP",
        "OVINE",
        "CHOIR",
        "VOWEL",
        "CRIMP",
        "MERRY",
        "STONY",
        "HIPPY",
        "SWORE",
        "MAXIM",
        "SCALY",
        "MEATY",
        "CUTIE"
    ],
    "VM2652DHATBV19681PNJCTP9C0QRVGA1": [
        "MOCHA",
        "REALM",
        "COVET",
        "FLOOD",
        "ASSAY",
        "SLANG",
        "WHELP",
        "PIXIE",
        "BLESS",
        "TUBER",
        "GUILE",
        "RELAX",
        "MINER",
        "FETUS",
        "MOUND",
        "RUMOR"
    ],
```

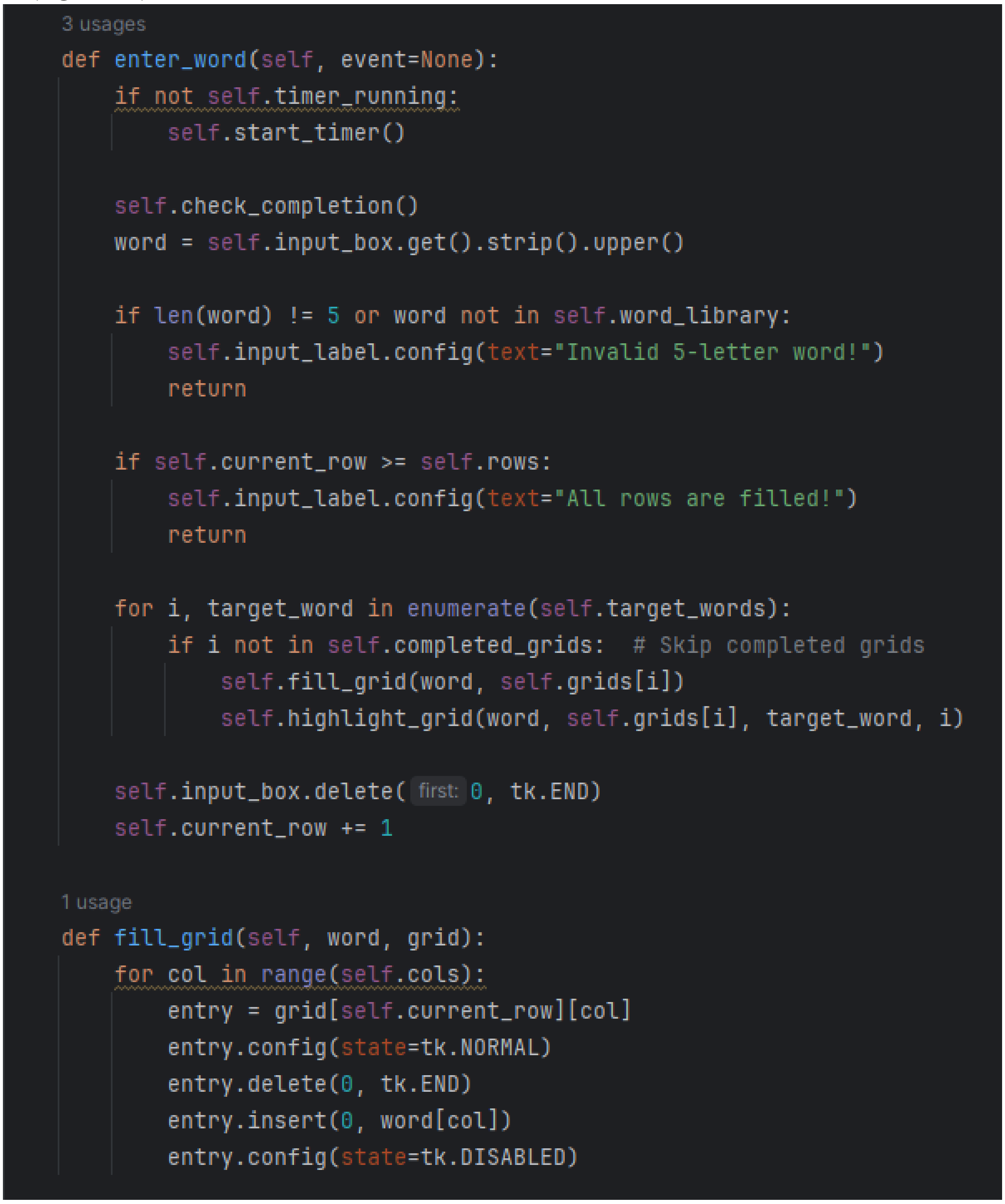
Figure 10



```python
# Ensure solver button appears correctly without conflicting with grid layout
if self.is_developer:
    self.solver_button = tk.Button(self.game_frame, text="Solver Mode", command=self.toggle_solver)
    self.solver_button.grid(row=21, column=0, padx=10, pady=5, sticky="w")

self.input_frame = tk.Frame(self.root, pady=5)
self.input_frame.pack()
```

Figure 12

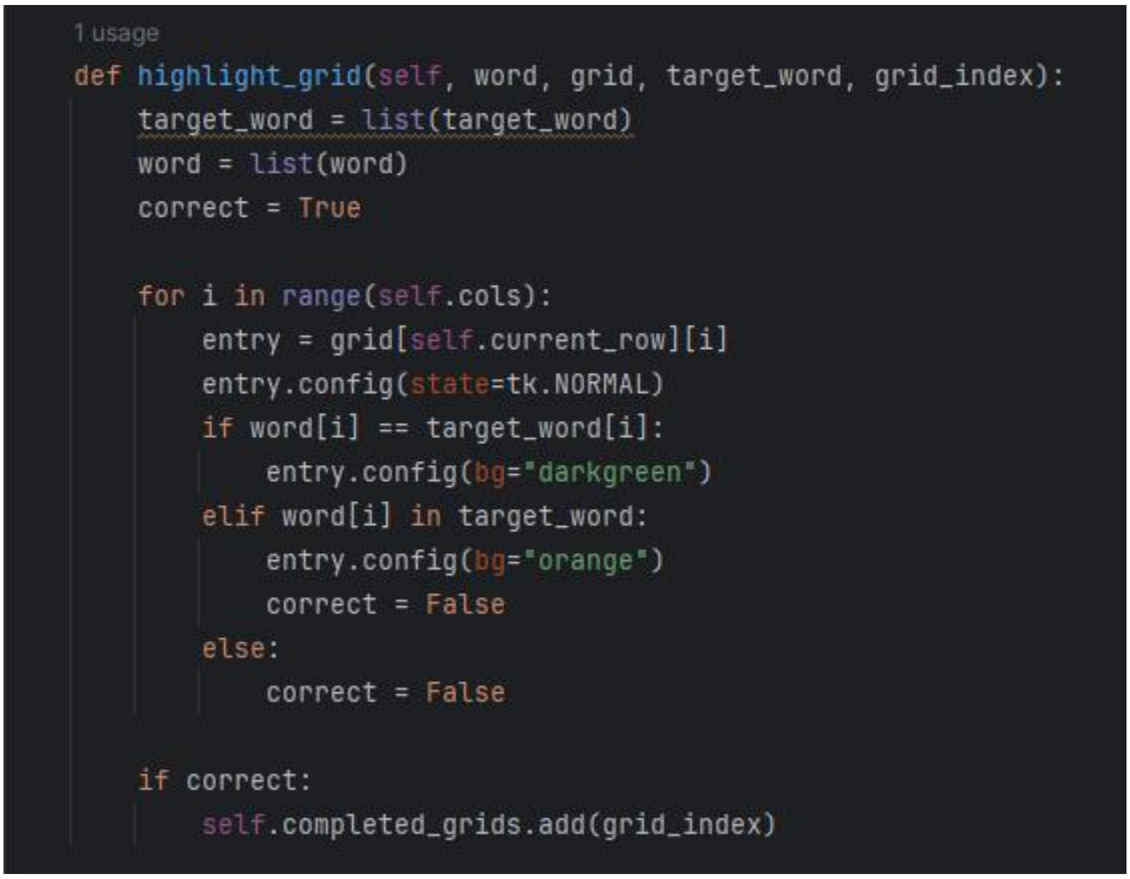
Word input validation ensures that only valid five-letter words from the word library are accepted. If an invalid word is entered, an error message is displayed, preventing incorrect inputs from affecting the game state. The system checks whether the entered word exists in the predefined word list, ensuring strict adherence to the game's rules. (*Figure 13*)



```
3 usages
def enter_word(self, event=None):
    if not self.timer_running:
        self.start_timer()

    self.check_completion()
    word = self.input_box.get().strip().upper()

    if len(word) != 5 or word not in self.word_library:
        self.input_label.config(text="Invalid 5-letter word!")
        return

    if self.current_row >= self.rows:
        self.input_label.config(text="All rows are filled!")
        return

    for i, target_word in enumerate(self.target_words):
        if i not in self.completed_grids:  # Skip completed grids
            self.fill_grid(word, self.grids[i])
            self.highlight_grid(word, self.grids[i], target_word, i)

    self.input_box.delete( first: 0, tk.END)
    self.current_row += 1

1 usage
def fill_grid(self, word, grid):
    for col in range(self.cols):
        entry = grid[self.current_row][col]
        entry.config(state=tk.NORMAL)
        entry.delete(0, tk.END)
        entry.insert(0, word[col])
        entry.config(state=tk.DISABLED)
```

Figure 13

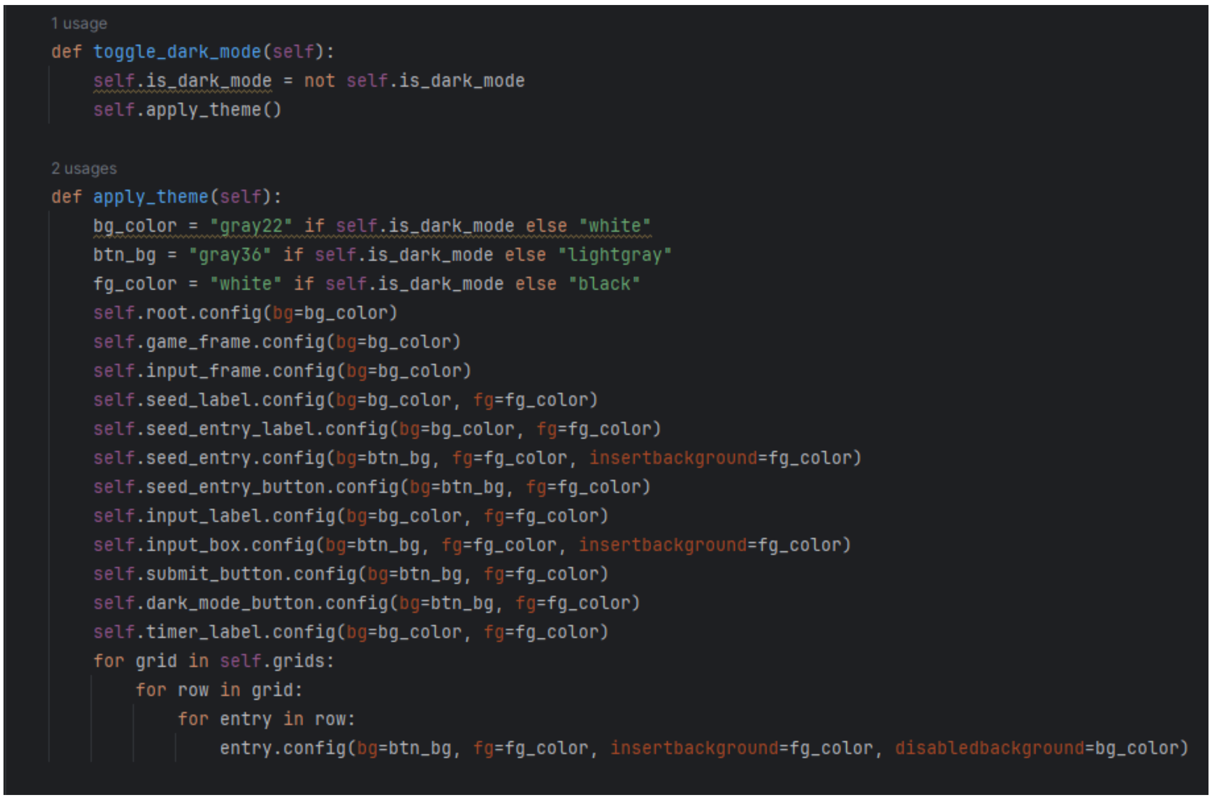
The grid visualization dynamically updates based on input accuracy. Each letter is compared against the target word in the respective grid. If the letter is in the correct position, the background colour turns green. If the letter is present but in the wrong position, the background turns orange. Incorrect letters remain unchanged. If all letters in a word are correct, the grid is marked as completed, and no further modifications are allowed. (*Figure 14*)



```
1 usage
def highlight_grid(self, word, grid, target_word, grid_index):
    target_word = list(target_word)
    word = list(word)
    correct = True

    for i in range(self.cols):
        entry = grid[self.current_row][i]
        entry.config(state=tk.NORMAL)
        if word[i] == target_word[i]:
            entry.config(bg="darkgreen")
        elif word[i] in target_word:
            entry.config(bg="orange")
            correct = False
        else:
            correct = False

    if correct:
        self.completed_grids.add(grid_index)
```

Figure 14

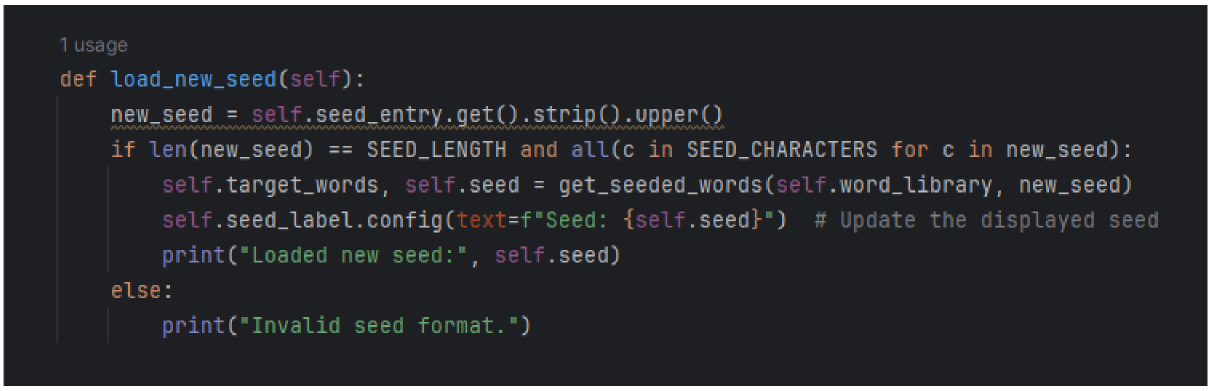
A toggleable dark mode is included for user preference. When activated, the entire interface undergoes a colour transformation to darker shades while ensuring text remains legible. The implementation dynamically applies the colour scheme to all UI elements, maintaining uniformity across buttons, labels, and input fields. (*Figure 15*)



```
1 usage
def toggle_dark_mode(self):
    self.is_dark_mode = not self.is_dark_mode
    self.apply_theme()

2 usages
def apply_theme(self):
    bg_color = "gray22" if self.is_dark_mode else "white"
    btn_bg = "gray36" if self.is_dark_mode else "lightgray"
    fg_color = "white" if self.is_dark_mode else "black"
    self.root.config(bg=bg_color)
    self.game_frame.config(bg=bg_color)
    self.input_frame.config(bg=bg_color)
    self.seed_label.config(bg=bg_color, fg=fg_color)
    self.seed_entry_label.config(bg=bg_color, fg=fg_color)
    self.seed_entry.config(bg=btn_bg, fg=fg_color, insertbackground=fg_color)
    self.seed_entry_button.config(bg=btn_bg, fg=fg_color)
    self.input_label.config(bg=bg_color, fg=fg_color)
    self.input_box.config(bg=btn_bg, fg=fg_color, insertbackground=fg_color)
    self.submit_button.config(bg=btn_bg, fg=fg_color)
    self.dark_mode_button.config(bg=btn_bg, fg=fg_color)
    self.timer_label.config(bg=bg_color, fg=fg_color)
    for grid in self.grids:
        for row in grid:
            for entry in row:
                entry.config(bg=btn_bg, fg=fg_color, insertbackground=fg_color, disabledbackground=bg_color)
```

Figure 15

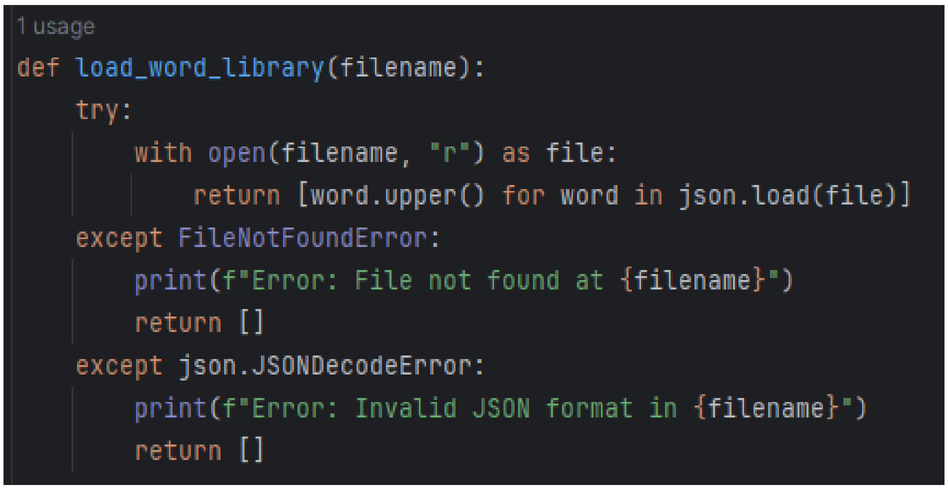
Seeds can also be manually loaded via the GUI. Users can input a custom seed, which is validated for correct length and character composition. If valid, the corresponding word set is loaded, allowing users to replay specific game variations. Invalid seeds are rejected, ensuring the integrity of the game. (*Figure 16*)



```
1 usage
def load_new_seed(self):
    new_seed = self.seed_entry.get().strip().upper()
    if len(new_seed) == SEED_LENGTH and all(c in SEED_CHARACTERS for c in new_seed):
        self.target_words, self.seed = get_seeded_words(self.word_library, new_seed)
        self.seed_label.config(text=f"Seed: {self.seed}")  # Update the displayed seed
        print("Loaded new seed:", self.seed)
    else:
        print("Invalid seed format.")
```

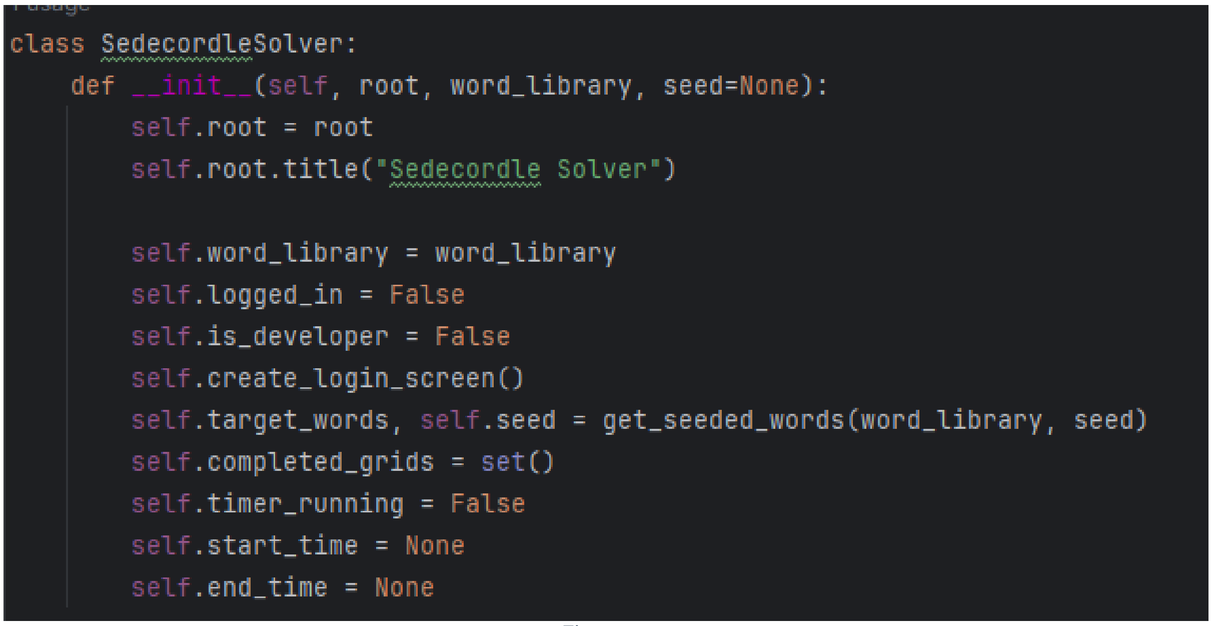
Figure 16

The game's main execution begins by loading the word library, stored as a JSON file(*Figure 19*). If the file is missing or corrupted, an error message is displayed, preventing further execution. Once the words are loaded, the game initializes, creating the GUI and setting up the necessary elements for user interaction.



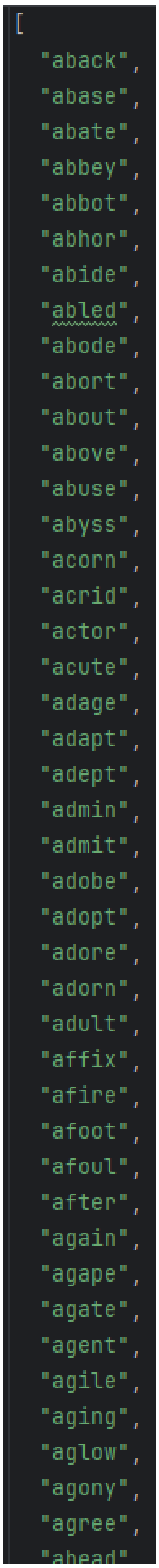
```
1 usage
def load_word_library(filename):
    try:
        with open(filename, "r") as file:
            return [word.upper() for word in json.load(file)]
    except FileNotFoundError:
        print(f"Error: File not found at {filename}")
        return []
    except json.JSONDecodeError:
        print(f"Error: Invalid JSON format in {filename}")
        return []
```

Figure 17



```
class SedecordleSolver:
    def __init__(self, root, word_library, seed=None):
        self.root = root
        self.root.title("Sedecordle Solver")

        self.word_library = word_library
        self.logged_in = False
        self.is_developer = False
        self.create_login_screen()
        self.target_words, self.seed = get_seeded_words(word_library, seed)
        self.completed_grids = set()
        self.timer_running = False
        self.start_time = None
        self.end_time = None
```

Figure 19



```
[
  "aback",
  "abase",
  "abate",
  "abbey",
  "abbot",
  "abhor",
  "abide",
  "abled",
  "abode",
  "abort",
  "about",
  "above",
  "abuse",
  "abyss",
  "acorn",
  "acrid",
  "actor",
  "acute",
  "adage",
  "adapt",
  "adept",
  "admin",
  "admit",
  "adobe",
  "adopt",
  "adore",
  "adorn",
  "adult",
  "affix",
  "afire",
  "afoot",
  "afoul",
  "after",
  "again",
  "agape",
  "agate",
  "agent",
  "agile",
  "aging",
  "aglow",
  "agony",
  "agree",
  "ahead"
```

Figure 18

Words: 1088