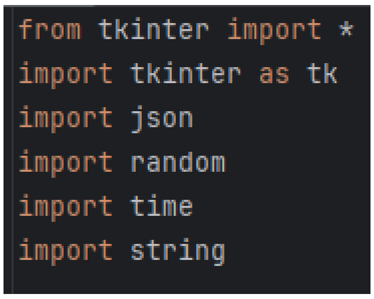
# Criterion C

**Programming Techniques & Tools**

The project was developed using Python due to its extensive library, ease of use, and strong community support. Its compatibility with libraries such as Tkinter and JSON made it ideal for building both the front-end and back-end efficiently. Tkinter enabled the creation of a lightweight, customizable GUI, while JSON offers simple, readable data storage for users and seeds.

To support core functionality, the program also uses (*Figure 1*):

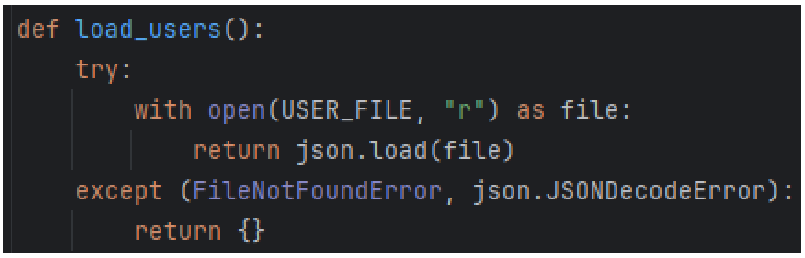
- *random* and *string* to generate unpredictable Base-32 game seeds
- *time* to track player performance to the millisecond



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

*Figure 1*

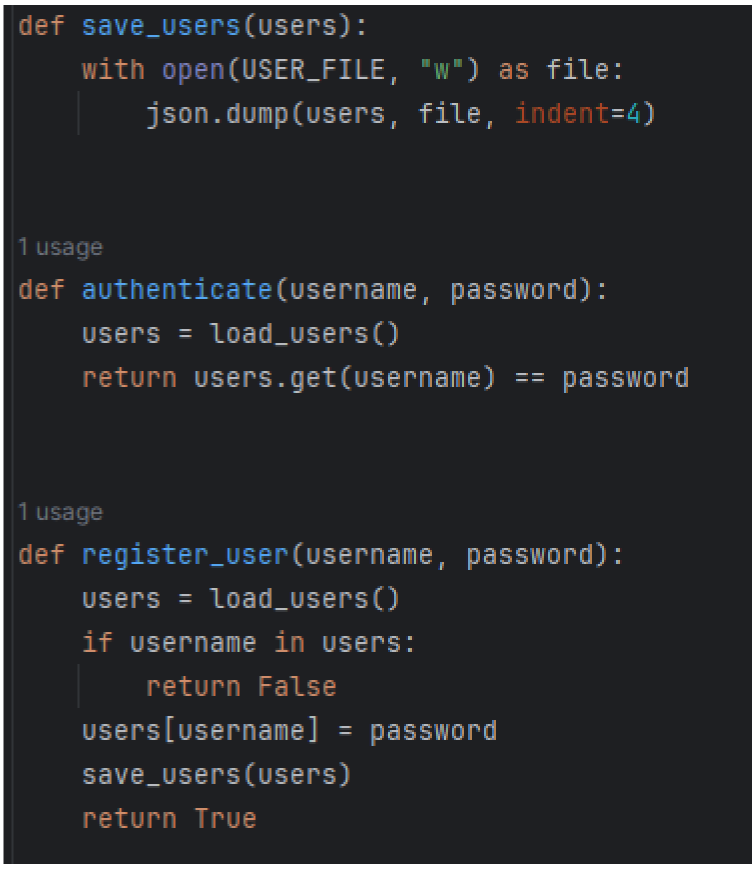
The user authentication system enables users to register and log in using a local JSON file *users.json*, which stores usernames and passwords as key-value pairs. The system first attempts to load the *users.json*, using error handling to account for missing files or invalid JSON formatting. (*Figure 2*)



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

*Figure 2*

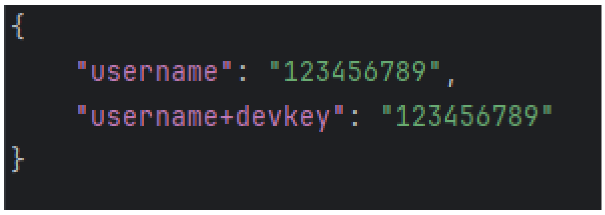
During registration, the system checks for duplicate usernames by comparing inputs against existing entries. If unique, the new credentials are stored in *users.json*. Login validation verifies whether the input matches the saved password. (*Figure 3*)



```
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*

Developer accounts are identified using the *+devkey* suffix in the username, which enables privileged features withing the software. This simple identifier is parsed during login to toggle dev-only functionality (*Figure 4*)



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

*Figure 4*

Game generation is powered by a seed-based system, where each unique combination of 16 words maps to a specific alphanumeric identifier. With 2,309 accepted words and 16 selected per game, the number of permutations exceeds $10^{43}$, making a traditional list-based approach impractical. Instead, the system encodes word selections using a compact Base-32 format, reducing a 179-bit binary value to a 36-character string. (*Figure 5, 6*)

Binary (179 Characters):

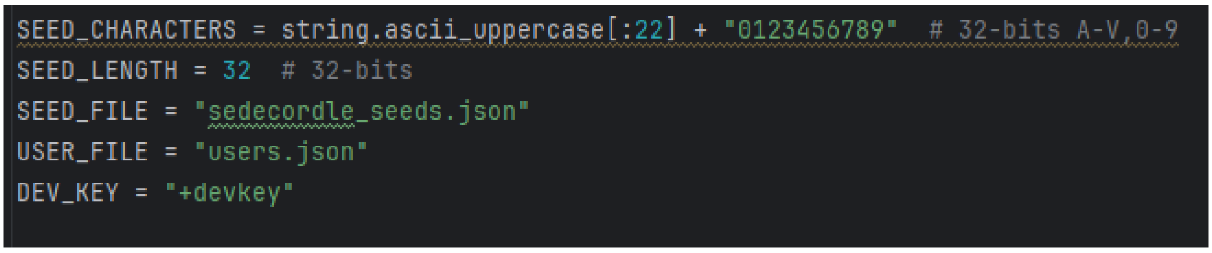
> 10110011010011011100100011111001010000011000110111011101001111001101110
> 11001111111000001110010111000100011010111101110100011010101010000110101 1
> 01000011101101010010110101010000101 01

Hexadecimal (45 Characters):

> 59A6E47CA0C6EE9E6ECFC1CB88D7BA3550D6876A5AA15

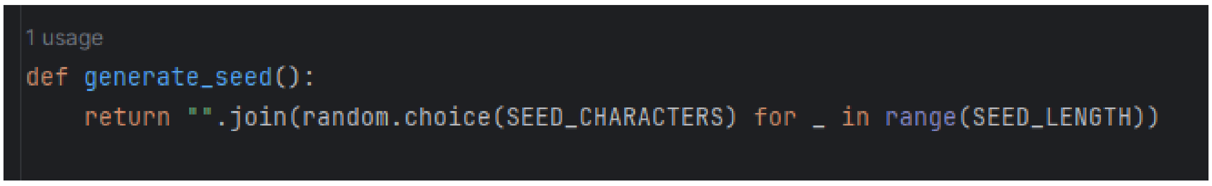
Base32 (36 Characters):

> 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"
```

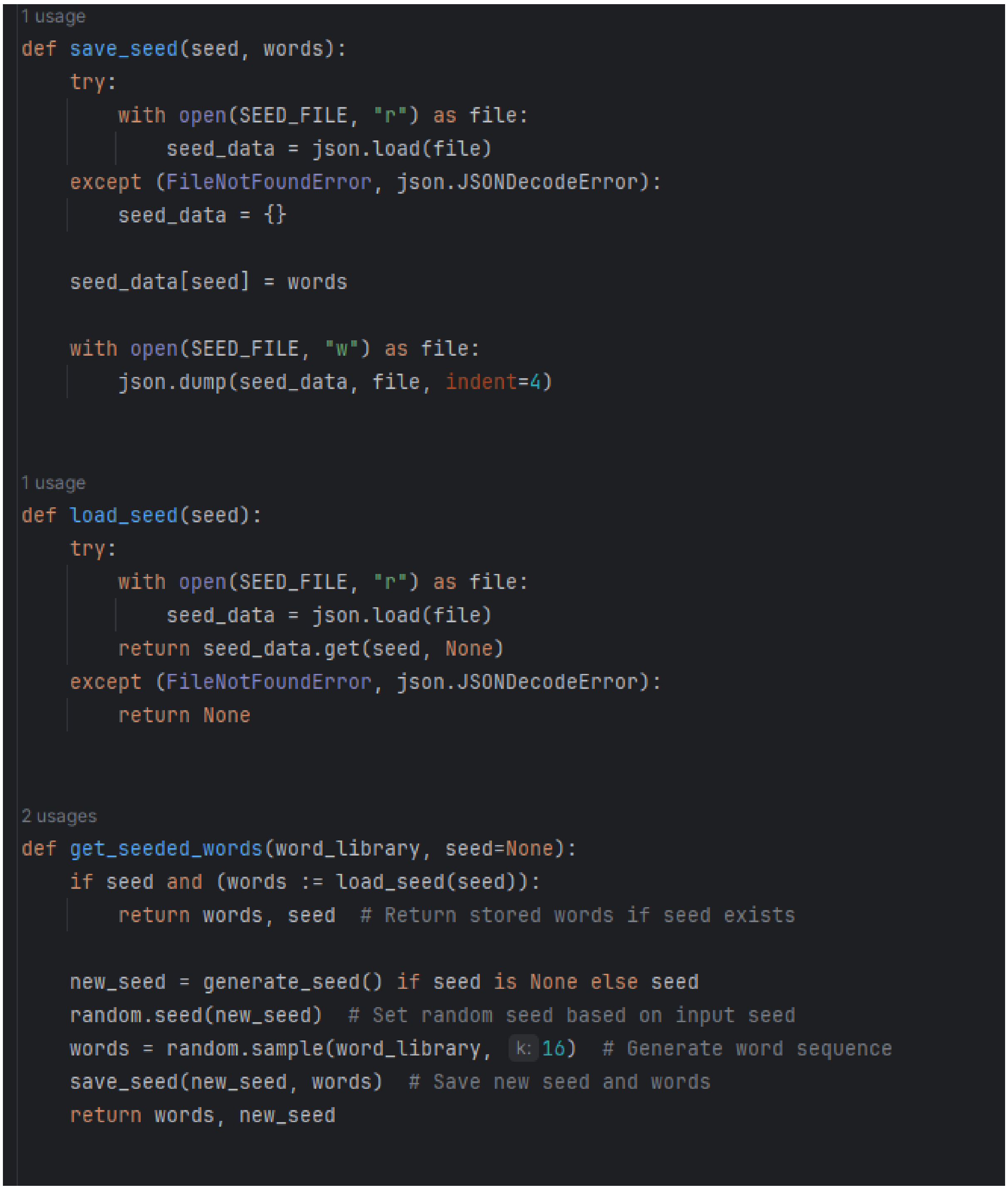
*Figure 5*



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

*Figure 6*

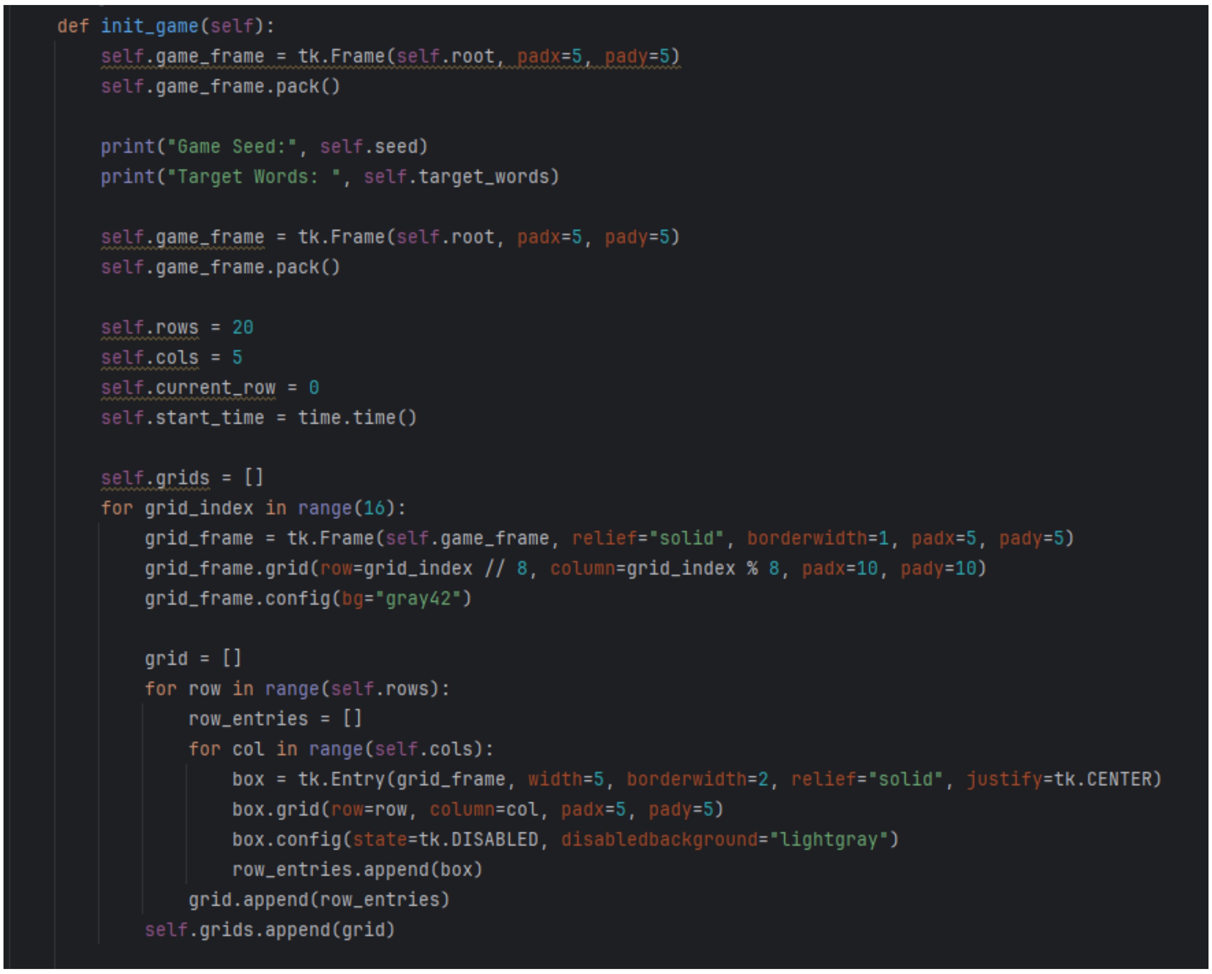
To ensure fairness and reproducibility, the program checks whether a provided seed has been used previously by attempting to load it from the *sedecordle_seeds*.json file. If found, the corresponding word set is retrieved. If the seed is new, 16 words are randomly selected from 5*_letter_word.json* using a seeded random generator, then saved and associated with the seed (*Figure 7*). This design prevents duplication and guarantees that the same seed will always produce the same game.



```python
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*

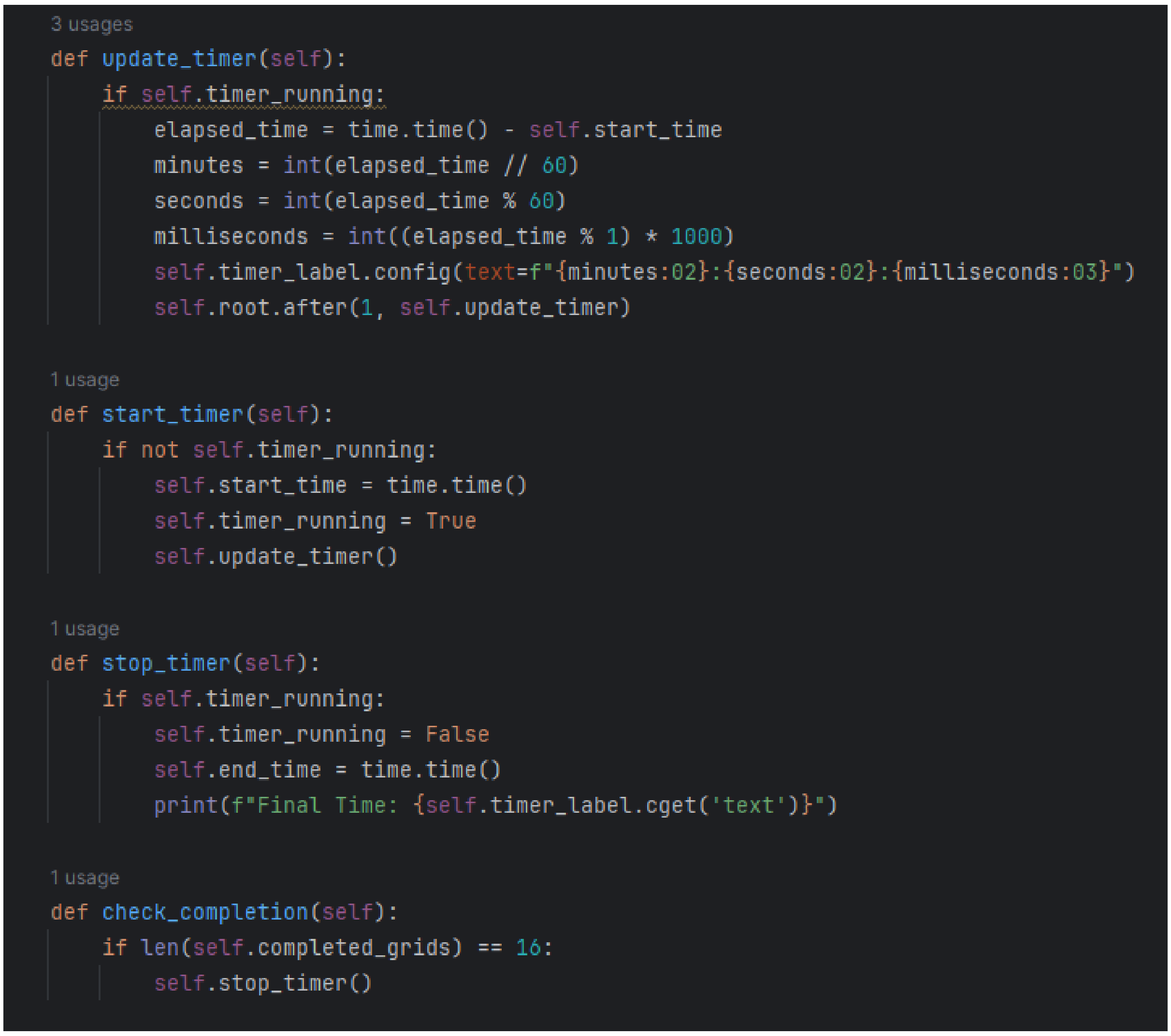
The graphical user interface was developed using Tkinter and structured using nested frames. The *init_game()* function creates 16 individual grids – each a *Frame* representing a target word – arranged using *.grid()* and *.pack()* for spacing and alignment. Within each grid, *Entry* widgets are initialised in a nested loop and stored in a 3D list *self.grids*. By default, all entry boxes are set to *DISABLED* using *.config()* , which prevent premature inputs. This structure allows dynamic updates over each cell's behaviour as the game progresses. (*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*

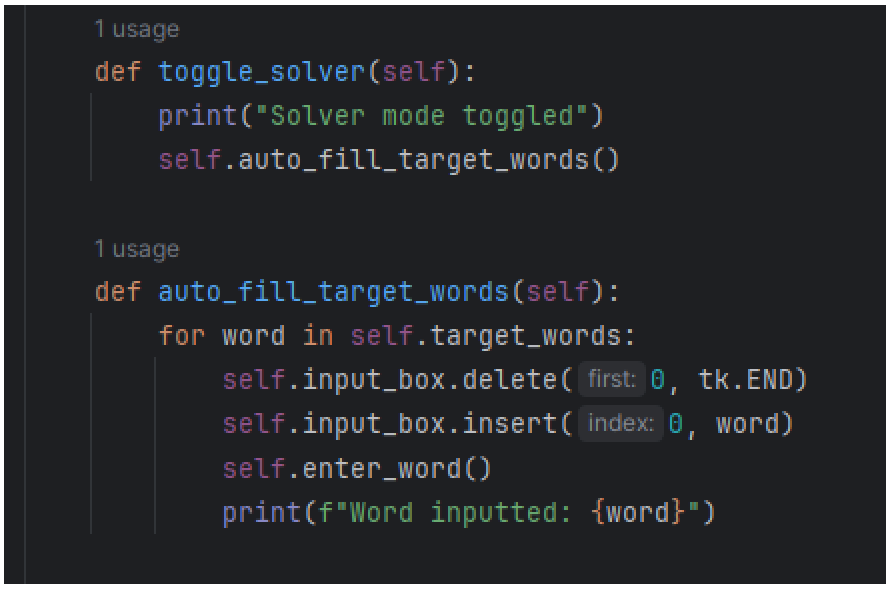
A timer system tracks the user's performance from the moment the first word is entered, updating live in minutes, seconds, and milliseconds. The *start_timer()* method sets the initial timestamp and flags the timer as running, while update_timer() uses *after(1, ...)* to refresh the timer every millisecond without blocking the UI. This loop calculates elapsed time and updates the display using formatted strings. Once all 16 words are completed, *check_comnpletion()* calls *stop_timer()* , which finalises the session. (*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()
```

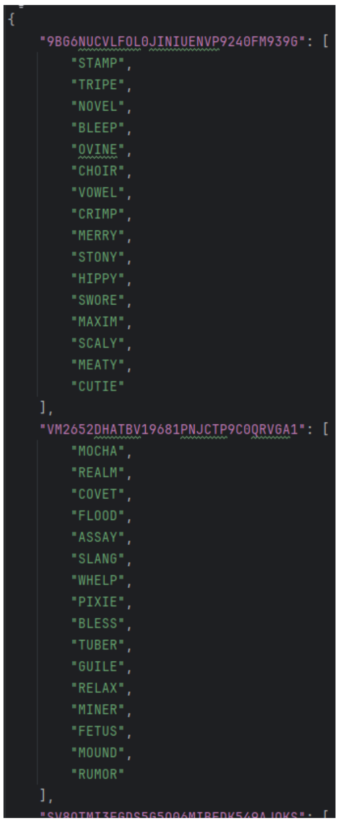
*Figure 9*

The solver mode is conditionally enabled for dev-accounts using an *if self.is_developer* check, which controls whether *solver_button* is rendered in the GUI (*Figure 12*). When clicked, this button triggers the *toggle_solver()* method, which calls *auto_fill_target_words()* to automate word entry (*Figure 11*). This function iterates over *self.target_words*, clears the current input field using *.delete(),* inserts the correct word via *.insert()* , and simulates manual entryu by calling *enter_word()*. Each word is printed to the console for debugging. These correct word sequences are mapped to unique Base32 seeds and stored in a JSON file for consistency (*Figure 10*).



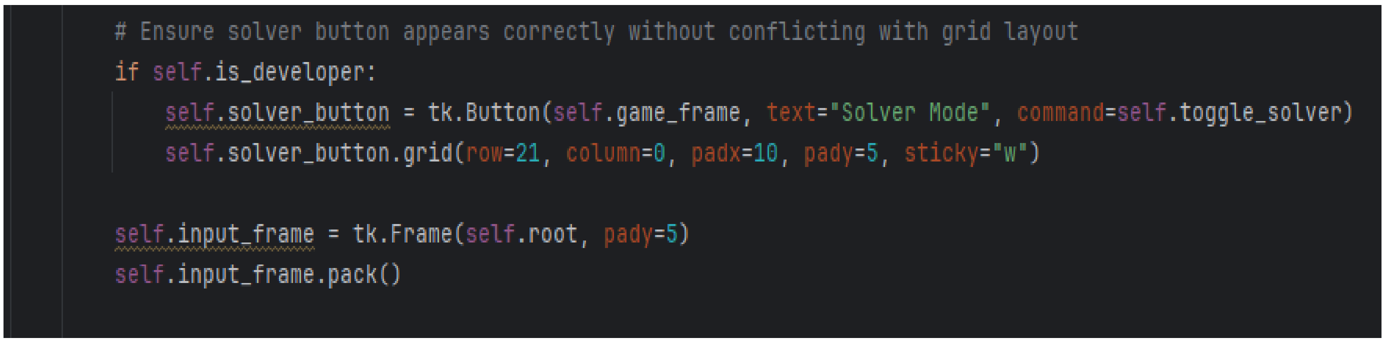
```python
def toggle_solver(self):
    print("Solver mode toggled")
    self.auto_fill_target_words()

def auto_fill_target_words(self):
    for word in self.target_words:
        self.input_box.delete(0, tk.END)
        self.input_box.insert(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"
    ],
```

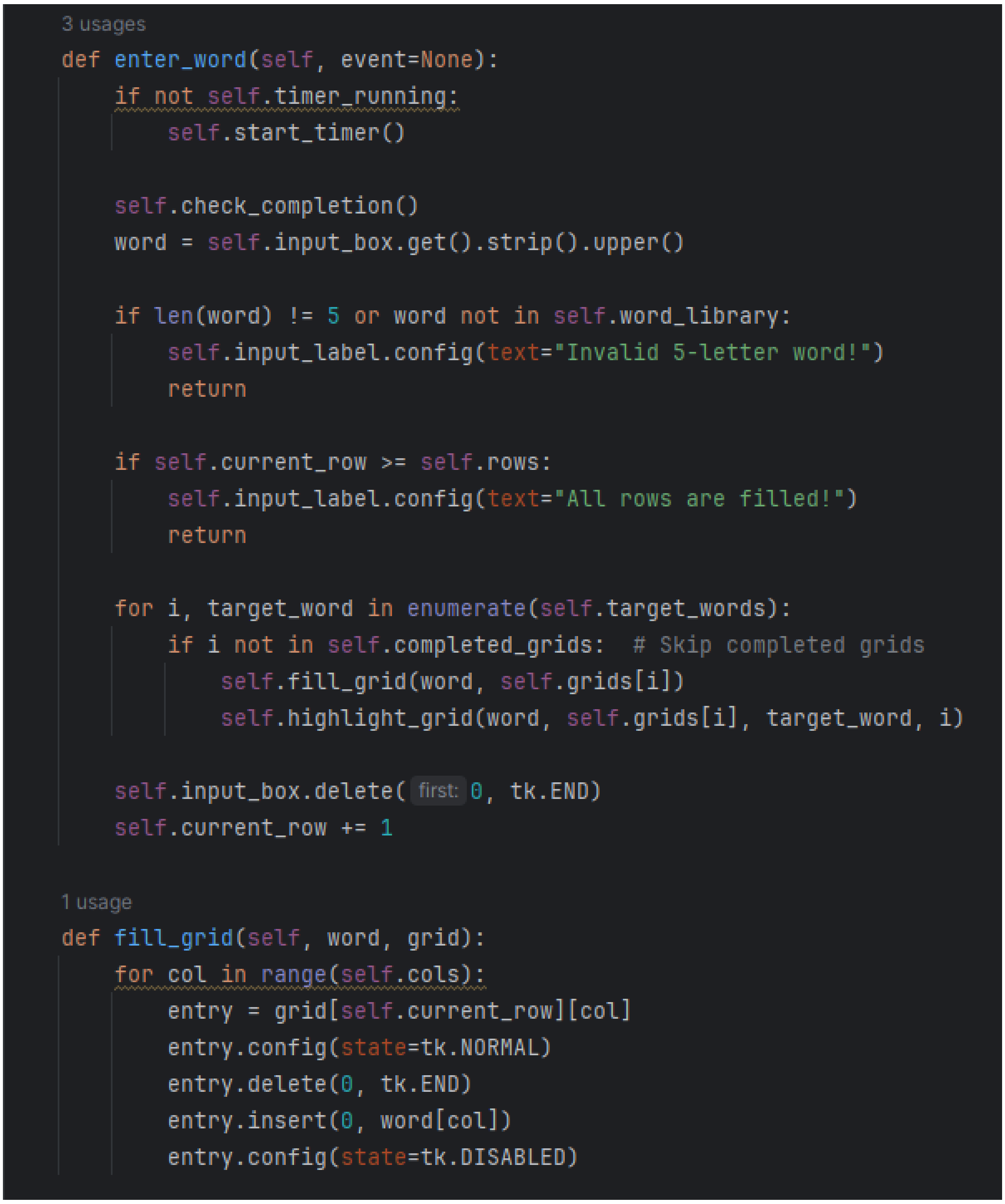
Figure 10



```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

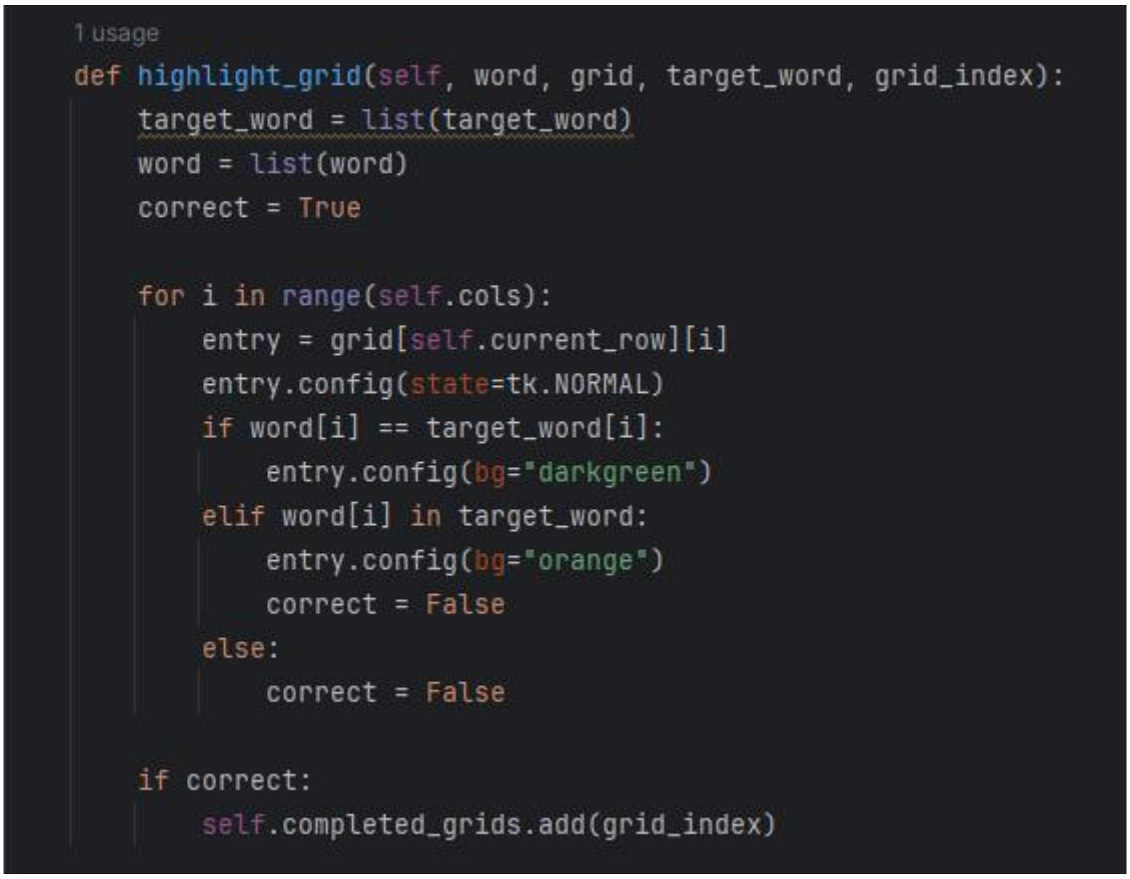
Word input is validated in the *enter_word()* method to ensure that only valid five-letter entries from 5_letter_words.json are accepted. The input is stripped, capitalized, and checked against both length and word list before proceeding. If invalid, *.config()* is used to update the input label with an error message, and the function exits early using *return*. If valid, the input is applied to all unfinished grids via *fill_grid()* and *highlight_grid(),* skipping any already-completed words. The input box is cleared using *.delete(),* and the row index is incremented. This method ensures strict rule enforcement and UI responsiveness. (*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*

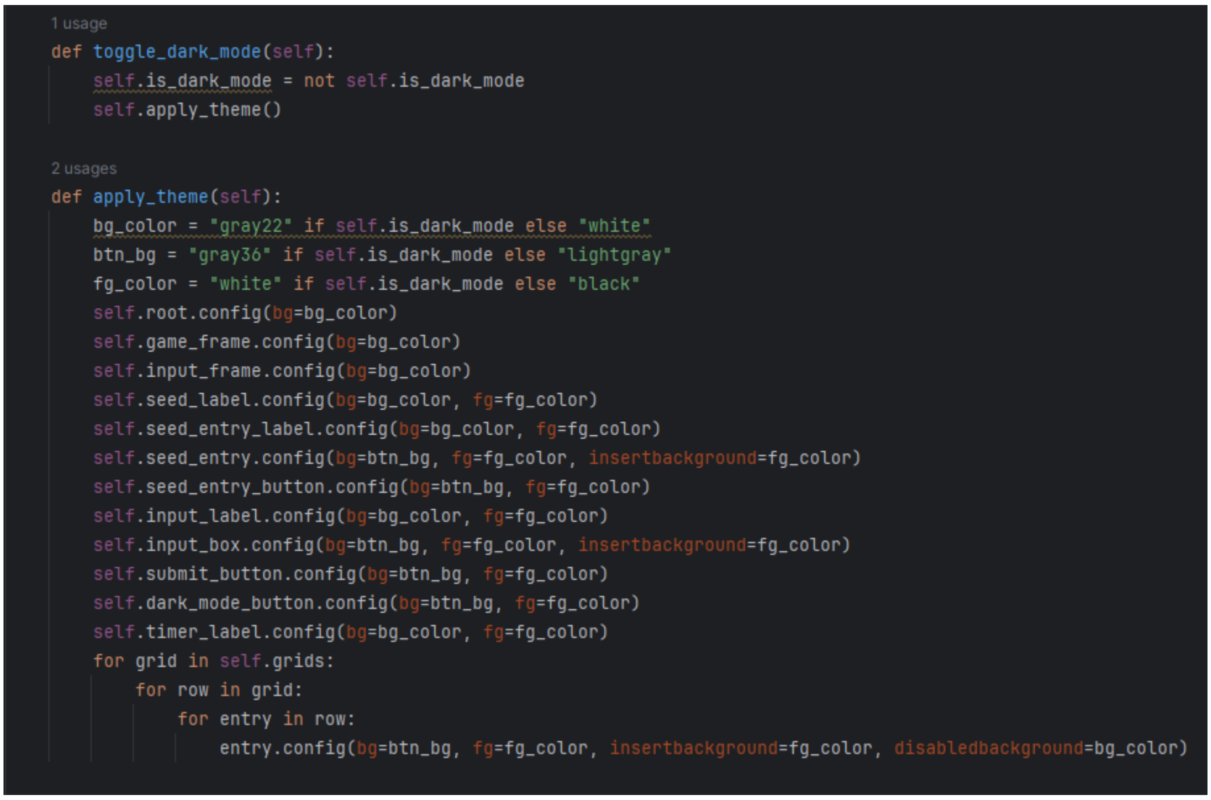
The *highlight_grid()* function visually compares each guessed letter to its corresponding target word character and updates the grid accordingly. Letters in the correct position are marked green, while letters that exist in the word but are misplaced are marked orange using *.config(bg=...)*. Each cell is accessed through a column-wise loop and temporarily enabled via *state=tk.NORMAL* before being coloured. If any letter is incorrect or misplaced, a correct flag is set to *False*. Once all five letters match, the grid index is added to *self.completed_grids*, preventing further edits. This method aids user understanding through real-time visual feedback and clear game state updates. (*Figure 14*)



```python
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*

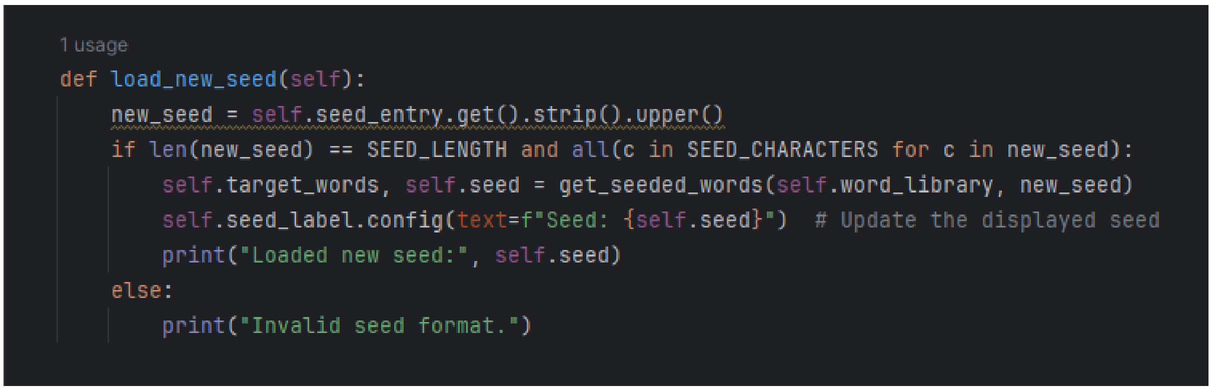
The *toggle_dark_mode()* method flips the *self.is_dark_mode* Boolean and calls *apply_theme(),* which dynamically updates the colour scheme of all interface elements based on the selected mode Background, foreground, and button colours are conditionally defined using inline if statements. The method then applies these settings using .*config()* to root labels, buttons, and entry fields. A nested loop iterates through every grid and entry box, ensuring uniform styling even for disabled fields. This implementation creates a responsive, full-UI transformation while preserving visual consistency between light and dark themes. (*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*

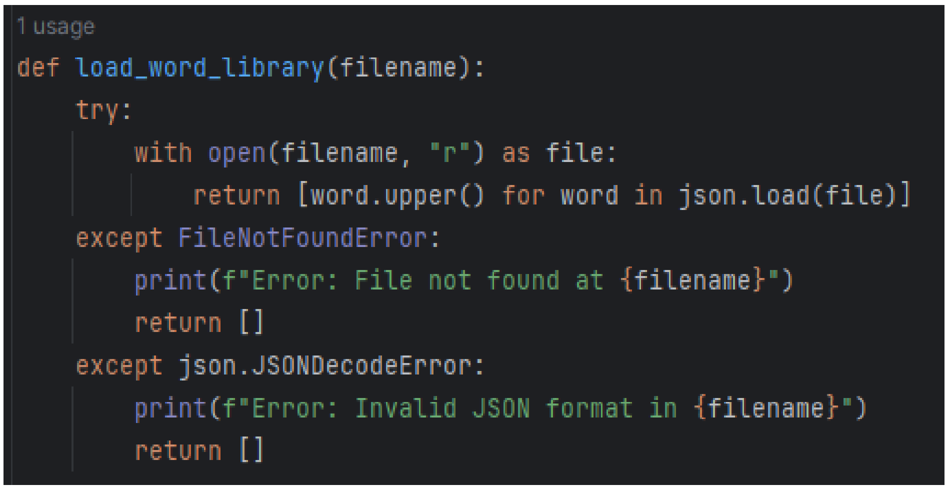
The *load_new_seed()* method allows users to input a custom seed manually through the GUI. The input is cleaned with *.strip().upper()* and validated by checking both length and character set using *all()* and *SEED_CHARACTERS*. If valid, the word set is retrieved using *get_seeded_words()* and assigned to *self.target_words*, while the interface is updated via *.config()* to display the active seed. If invalid, the function exits after printing an error message. This ensures that only properly formatted Base32-compatible seeds are accepted. (*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*

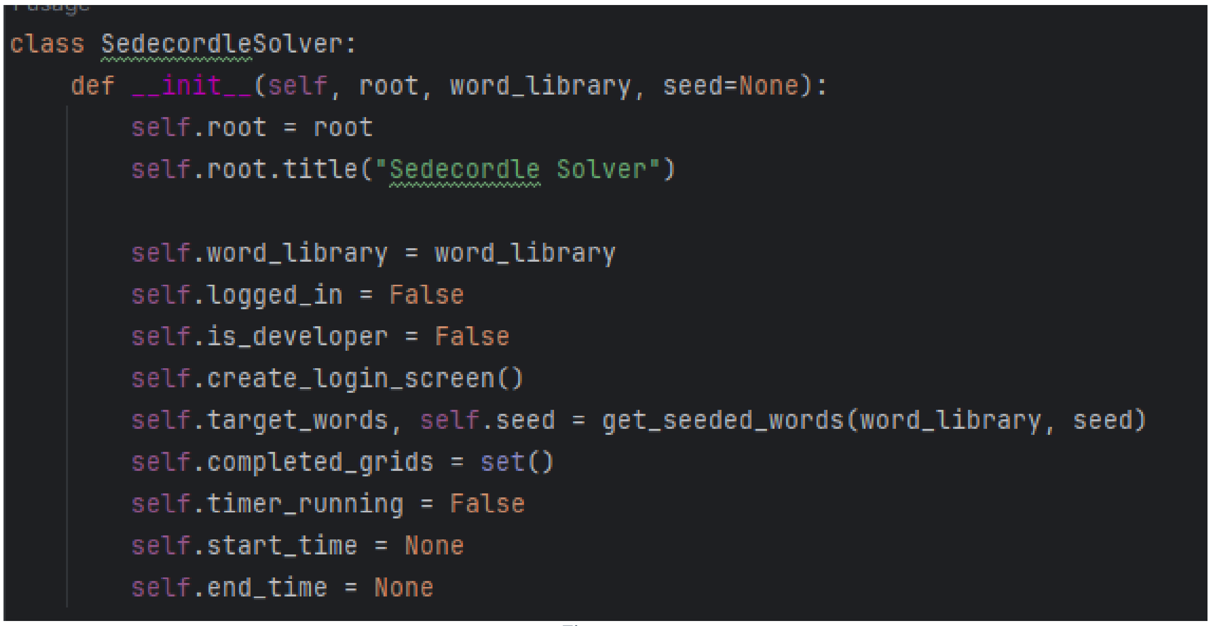
The program begins by loading the word library from *5_letter_words.json* using *load_word_library()*. This method reads and capitalizes each word, handling potential errors such as missing files or invalid JSON formatting via *try/except* blocks to ensure the game does not crash. (*Figure 17*)



```
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*

Once loaded, the *SedecordleSolver* class is initialized, setting up all necessary variables, including GUI elements, state flags, and seed-based word selection. Core attributes such as *self.word_library, self.logged_in, self.is_developer*, and *self.target_words* are established in *__init__*. (*Figure 19)*



```
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 18*

Words: 1018

## Citations:

*GeeksforGeeks. (2024, December 21). Python Tkinter. GeeksforGeeks. Retrieved May 26, 2024, from https://www.geeksforgeeks.org/python-gui-tkinter/*

*Lab 12. (n.d.). Retrieved June 23, 2024, from https://cs111.wellesley.edu/archive/cs111_fall14/public_html/labs/lab12/tkintercolor.html*

*SedecordleGame. (2021, April 13). Sedecordle: Solve 16 wordles at once. Sedecordle Game - 16 Words Wordle. Retrieved May 24, 2024, from https://sedecordlegame.org/*

*An alphabetized list of five-letter words used in Wordle as the puzzle clues. (n.d.). Gist. Retrieved June 9, 2024, from https://gist.github.com/slushman/34e60d6bc479ac8fc698df8c226e4264*

*Br34th3r. (n.d.). GitHub - br34th3r/PythonLoginAndRegister: Basic Login and Registration System to be Used by New Developers to Structure their own Modules. GitHub. Retrieved July 22, 2024, from https://github.com/br34th3r/PythonLoginAndRegister*

*Using JSON files to store and retrieve list data. (n.d.). Stack Overflow. Retrieved June 9, 2024, from https://stackoverflow.com/questions/62418874/using-json-files-to-store-and-retrieve-list-data*

*how do I make a Timer in Python. (n.d.). Stack Overflow. Retrieved June 23, 2024, from https://stackoverflow.com/questions/70058132/how-do-i-make-a-timer-in-python*

*GeeksforGeeks. (2024a, July 26). Random numbers in Python. GeeksforGeeks. Retrieved July 14, 2024, from https://www.geeksforgeeks.org/random-numbers-in-python/*

*ChatGPT, response to "How could I generate 32-bit (A-V, 0-9) "seeds" randomly in python?" OpenAI, September 26, 2023, https://www.chatgpt.com*

*ChatGPT, response to "How could I implement a developer-key function, that when included into the username during registration, gives users special access to additional features, in python?" OpenAI, September 26, 2023, https://www.chatgpt.com*