



Full
Code



Advanced Programming

Book

Tkinter

Sqlite3

Matplotlib

Table of Contents

[install Visual Studio Code \(VSCode \) and run your first Python program :](#)

[Step 1 : Download and Install Visual Studio Code](#)

[Step 2 : Install Python](#)

[Step 3 : Install the Python Extension for VSCode](#)

[Step 4 : Create a Python File](#)

[Vocabulary Master](#)

[Budget Tracker](#)

[To Do List Manager](#)

[Weather Forecasts](#)

[Quiz Game Multiple Choices](#)

[Pomodoro Timer](#)

[Advanced Pomodoro Timer](#)

[FlashCard](#)

[Image Editor](#)

[Notes](#)

[Habit Tracker](#)

[Password Manager](#)

install Visual Studio Code (VSCode) and run your first Python program:

Step 1: Download and Install Visual Studio Code

1. Visit the official Visual Studio Code website at <https://code.visualstudio.com/>.
2. Click on the "Download for Windows" button if you're using Windows, or select the appropriate download link for your operating system (Windows, macOS, or Linux).
3. Follow the installation instructions for your operating system. The installation process is straightforward and typically involves clicking "Next" or "Install" with the default settings.

← → ⌂ code.visualstudio.com

Visual Studio Code Docs Updates Blog API Extensions FAQ Learn

[Version 1.91](#) is now available! Read about the new features and fixes from J

Free. Built on open source. Runs everywhere.

Code Editing. Redefined.

[Download for Windows](#)

[Web](#), [Insiders edition](#), or [other platforms](#)

The screenshot shows the official Visual Studio Code website. At the top, there's a navigation bar with links for 'Docs', 'Updates', 'Blog', 'API', 'Extensions', 'FAQ', and 'Learn'. A banner at the top right announces 'Version 1.91' with a link to its details. Below the banner, a large heading reads 'Code Editing. Redefined.' with a 'Download for Windows' button underneath. To the right of the heading is a screenshot of the VS Code interface. It shows the Explorer sidebar on the left with a tree view of a 'MY-APP' project containing components like 'components', 'button', and 'button.css'. The main editor area on the right displays a TypeScript file named 'button.ts' with the following code:

```
1 interface Button {
2   onClick(): void;
3   text: string;
4 }
5 const Button: React.FC<Button> = ({text}) =>
6   return <button>{text}</button>;
7 ;
8
9 export default Button;
```

A modal dialog is overlaid on the editor, prompting the user to 'Create a new button'. There are 'Accept' and 'Discard' buttons at the bottom of the dialog. At the bottom of the interface screenshot, there are tabs for 'PROBLEMS', 'OUTPUT', and 'TERMINAL', along with some log entries.

Step 2: Install Python

Before you can run **Python** programs in VSCode, you need to have **Python** installed on your computer. If you don't have **Python** installed, follow these steps:

1. Visit the official **Python** website at <https://www.Python.org/downloads/>.
2. Download the latest version of **Python** for your operating system. Be sure to download **Python** 3.x, as **Python** 2.x is no longer supported.
3. Run the **Python** installer and make sure to check the box that says "Add **Python** to PATH" during installation. This will make it easier to run **Python** from the command line.

← → ⌂ python.org/downloads/

Python PSF Docs PyPI

 python™ Donate

About Downloads Documentation Community Success

Download the latest version for Windows

[Download Python 3.12.4](#)

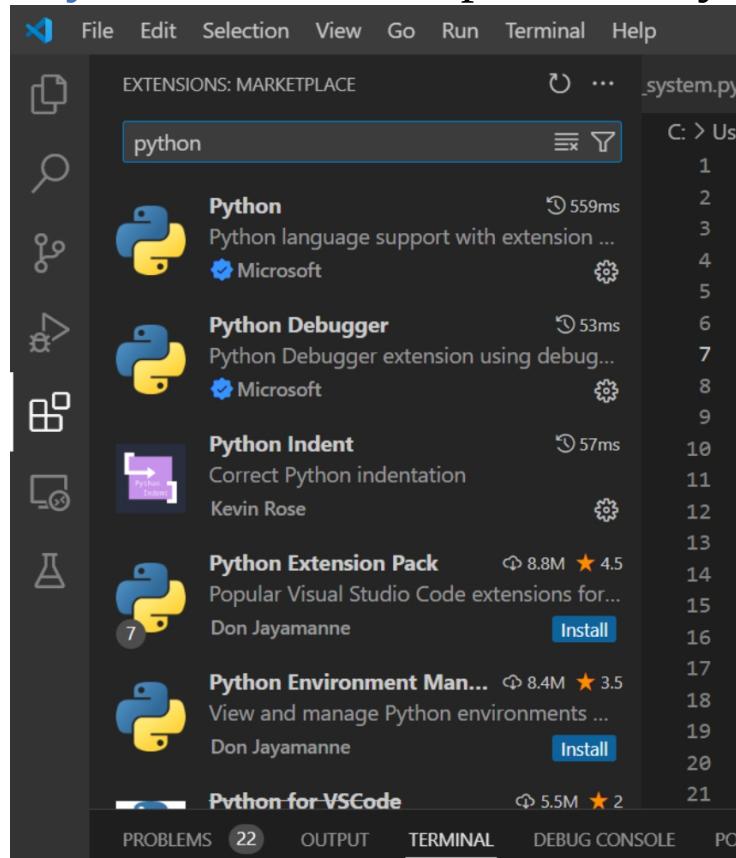
Looking for Python with a different OS? Python for [Windows](#), [Linux/UNIX](#), [macOS](#), [Other](#)

Want to help test development versions of Python 3.13? [Prereleases](#), [Docker images](#)



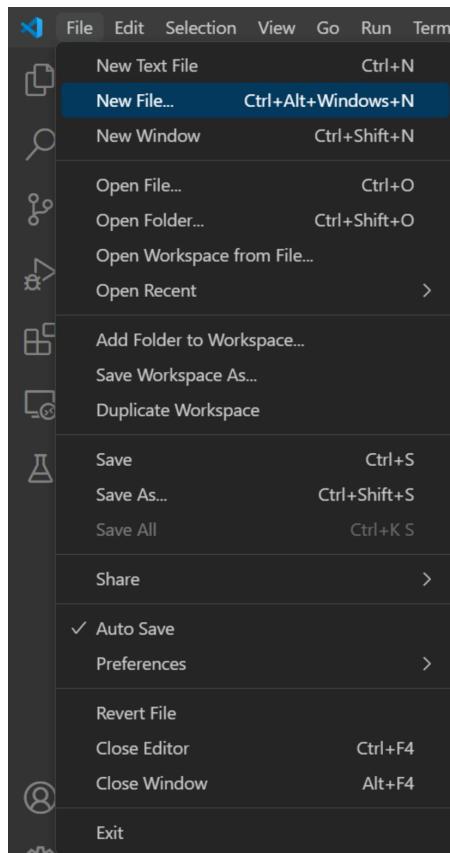
Step 3: Install the Python Extension for VSCode

1. Open Visual Studio Code.
2. Go to the Extensions view by clicking on the square icon in the left sidebar or pressing **Ctrl+Shift+X**.
3. Search for "**Python**" in the search bar at the top of the Extensions view.
4. Click the "Install" button next to the "**Python**" extension provided by Microsoft.



Step 4: Create a Python File

1. Open VSCode.
2. Click on "File" in the top menu, then select "New File."
3. Save the new file with a **.py** extension, for example, "hello.py."



Step 5: Write Your First [Python](#) Program

Inside your "hello.py" file, write a simple [Python](#) program. For example:

```
print("Hello, World!")
```

Step 6: Run Your [Python](#) Program

Go to “Run” menu and click “Run Without Debugging” or press **CTRL+F5**

You should see the output "**Hello, World!**" displayed in the **terminal panel** at the bottom of the VSCode window.

Congratulations! You've successfully installed Visual Studio Code, installed [Python](#), created your first [Python](#) file, and executed a [Python](#) program using VSCode. You can now continue writing and running [Python](#) code within the VSCode environment.

File Edit Selection View Go Run Terminal Help

employee_management.py

```
C: > Users > Suchat > household_ex
1 import tkinter as
2 from tkinter import
3 import matplotlib
4 from matplotlib import
5 import json
6 from datetime import
7
8
9 class ExpenseTrack:
10     def __init__(self):
11         self.root = tk.Tk()
12         self.root.title("Expense Tracker")
13
14         # Style configuration
15         self.style = ttk.Style()
16         self.style.theme_use("classic")
17         self.style.configure("Treeview", font=("Ari
18         self.style.configure("Treeview.Heading", font=
19         self.style.configure("Treeview.Column", font=
20
21     # Dictiona
```

Start Debugging F5

Run Without Debugging Ctrl+F5

Stop Debugging Shift+F5

Restart Debugging Ctrl+Shift+F5

Open Configurations

Add Configuration...

Step Over F10

Step Into F11

Step Out Shift+F11

Continue F5

Toggle Breakpoint F9

New Breakpoint >

Enable All Breakpoints ('Arial')

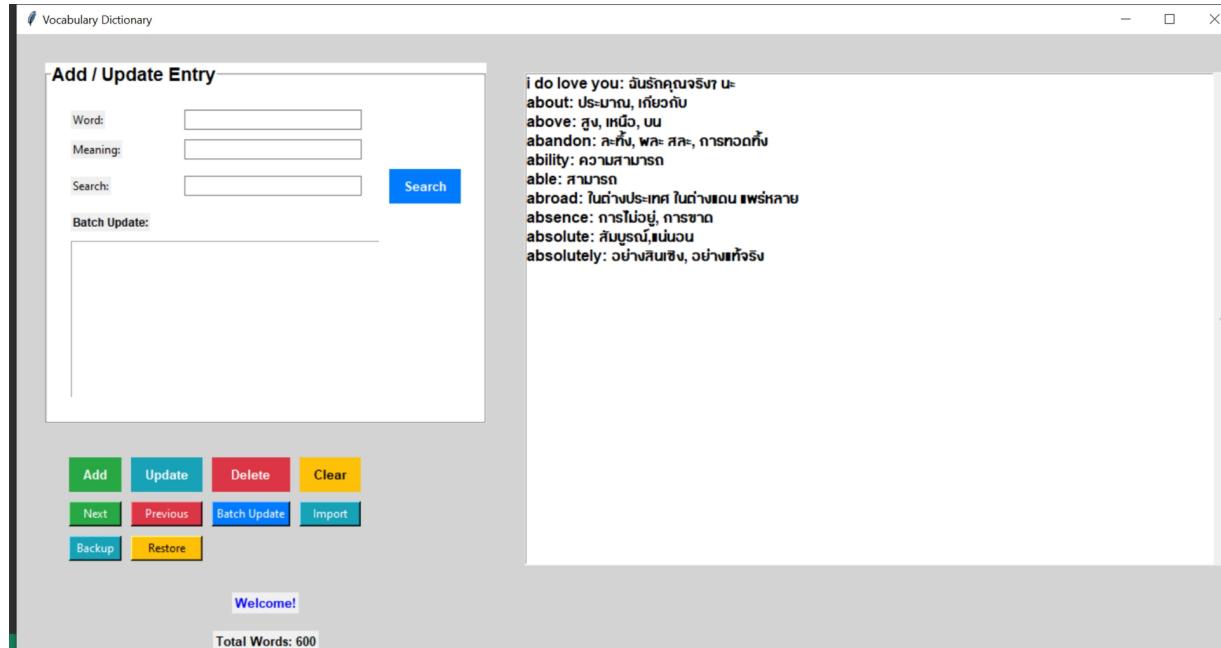
Disable All Breakpoints ('))

Remove All Breakpoints ('))

Install Additional Debuggers...

PROBLEMS 22 OUTPUT TERMINAL DEBUG CONSOLE PORTS

Vocabulary Master



```
import tkinter as tk
from tkinter import ttk
from tkinter.filedialog import askopenfilename
import sqlite3
import shutil
import datetime

# Database setup functions
```

```
def create_connection():
    try:
        conn = sqlite3.connect('vocabulary.db')
        return conn
    except sqlite3.Error as e:
        print(f"SQLite error: {e}")
        return None
```

```
def create_table():
    conn = create_connection()
    if conn is None:
```

```
print("Failed to create database connection.")
return

cursor = conn.cursor()
cursor.execute("""
    CREATE TABLE IF NOT EXISTS vocabulary (
        word TEXT PRIMARY KEY,
        meaning TEXT NOT NULL
    )
""")
conn.commit()
conn.close()
```

```
def add_word_to_db(word, meaning):
    conn = create_connection()
    cursor = conn.cursor()
    cursor.execute("""
        INSERT OR REPLACE INTO vocabulary (word, meaning)
        VALUES (?, ?)
    """, (word, meaning))
    conn.commit()
    conn.close()
```

```
def delete_word_from_db(word):
    conn = create_connection()
    cursor = conn.cursor()
    cursor.execute("""
        DELETE FROM vocabulary WHERE word = ?
    """, (word,))
    conn.commit()
    conn.close()
```

```
def get_all_words_from_db(limit=None, offset=0):
    conn = create_connection()
    cursor = conn.cursor()
```

```
query = 'SELECT word, meaning FROM vocabulary'
if limit is not None:
    query += ' LIMIT ? OFFSET ?'
    cursor.execute(query, (limit, offset))
else:
    cursor.execute(query)
rows = cursor.fetchall()
conn.close()
return rows
```

```
def search_word_from_db(query):
    conn = create_connection()
    cursor = conn.cursor()
    cursor.execute("""
        SELECT word, meaning FROM vocabulary
        WHERE word LIKE ? OR meaning LIKE ?
    ", (f'%{query}%', f'%{query}%'))
    rows = cursor.fetchall()
    conn.close()
    return rows
```

```
# Functions for GUI operations
```

```
def add_word():
    word = entry_word.get().strip()
    meaning = entry_meaning.get().strip()
    if word and meaning:
        add_word_to_db(word, meaning)
        status_label.config(
            text=f"Added '{word}' to vocabulary.", foreground='green')
        update_word_list()
    else:
        status_label.config(
            text="Both word and meaning are required.", foreground='red')
```

```
def update_word():
```

```

selected = listbox_words.curselection()
if selected:
    index = selected[0]
    selected_item = listbox_words.get(index)
    old_word = selected_item.split(':')[0].strip() # คำศัพท์เดิม
    new_word = entry_word.get().strip() # คำศัพท์ใหม่
    new_meaning = entry_meaning.get().strip() # ความหมายใหม่

    if new_word and new_meaning:
        # ลบคำศัพท์เดิม
        delete_word_from_db(old_word)
        # เพิ่มคำศัพท์ใหม่
        add_word_to_db(new_word, new_meaning)
        status_label.config(
            text=f"Updated '{old_word}' to '{new_word}'.",
            foreground='green')
        update_word_list()
    else:
        status_label.config(
            text="Both word and meaning are required for update.",
            foreground='red')
else:
    status_label.config(
        text="Select an item to update.",
        foreground='red')

```

```

# Modify delete_word function to handle multiple selections
def delete_word():
    selected_indices = listbox_words.curselection()
    if selected_indices:
        for index in selected_indices:
            selected_item = listbox_words.get(index)
            word = selected_item.split(':')[0].strip()
            delete_word_from_db(word)
            status_label.config(
                text=f"Deleted selected words from vocabulary.",
                foreground='green')
            update_word_list()

```

```
    else:
        status_label.config(text="Select items to delete.",
foreground='red')

def show_all():
    update_word_list()

def search_word():
    query = entry_search.get().strip()
    if query:
        words = search_word_from_db(query)
        listbox_words.delete(0, tk.END)
        if words:
            for word, meaning in words:
                listbox_words.insert(tk.END, f"{word}: {meaning}")
        else:
            listbox_words.insert(tk.END, "No matching words found.")
    else:
        listbox_words.insert(tk.END, "Please enter a search query.")
```

```
def clear_fields():
    entry_word.delete(0, tk.END)
    entry_meaning.delete(0, tk.END)
    entry_search.delete(0, tk.END)
    status_label.config(text="Fields cleared.", foreground='blue')
```

```
def update_word_list(page=0):
    global current_page
    current_page = page
    words = get_page_data(current_page)
    listbox_words.delete(0, tk.END)
    if words:
        for word, meaning in words:
            listbox_words.insert(tk.END, f"{word}: {meaning}")
    else:
```

```
listbox_words.insert(tk.END, "No words found.")  
update_word_count()
```

```
def get_page_data(page):  
    offset = page * items_per_page  
    return get_all_words_from_db(items_per_page, offset)
```

```
def next_page():  
    global current_page  
    current_page += 1  
    update_word_list(current_page)
```

```
def previous_page():  
    global current_page  
    if current_page > 0:  
        current_page -= 1  
    update_word_list(current_page)
```

```
def batch_update():  
    words_to_update = entry_batch_update.get("1.0",  
    tk.END).strip().split('\n')  
    for line in words_to_update:  
        if ':' in line:  
            word, meaning = line.split(':', 1)  
            word = word.strip()  
            meaning = meaning.strip()  
            if word and meaning:  
                add_word_to_db(word, meaning)  
    status_label.config(text="Batch update completed.",  
foreground='green')  
    update_word_list()
```

```
def import_from_file():  
    file_path = askopenfilename(filetypes=[("Text files", "*.txt")])  
    if file_path:
```

```
with open(file_path, 'r', encoding='utf-8') as file:
    for line in file:
        if ':' in line:
            word, meaning = line.strip().split(':', 1)
            word = word.strip()
            meaning = meaning.strip()
            if word and meaning:
                add_word_to_db(word, meaning)
status_label.config(text="Import completed.", foreground='green')
update_word_list()
```

```
def backup_database():
    timestamp =
datetime.datetime.now().strftime('%Y%m%d_%H%M%S')
    backup_path = f'vocabulary_backup_{timestamp}.db'
    shutil.copy('vocabulary.db', backup_path)
    status_label.config(
        text=f"Backup created: {backup_path}", foreground='green')
```

```
def restore_database():
    file_path = askopenfilename(filetypes=[("Database files", "*.db")])
    if file_path:
        shutil.copy(file_path, 'vocabulary.db')
        status_label.config(text="Database restored.", foreground='green')
        update_word_list()
```

```
def update_word_count():
    total_words = len(get_all_words_from_db())
    count_label.config(text=f"Total Words: {total_words}")
```

```
# GUI setup
root = tk.Tk()
root.title("Vocabulary Dictionary")
root.geometry("1280x720+0+0") # Adjusted size for better fitting
root.resizable(True, True)
```

```
create_table() # Ensure the table exists when the program starts

# Global Variables
current_page = 0
items_per_page = 10

# Frames
frame_main = tk.Frame(root, padx=20, pady=20, bg='lightgray')
frame_main.pack(fill=tk.BOTH, expand=True)

frame_entry = tk.LabelFrame(frame_main, text="Add / Update Entry",
                            padx=20, pady=20, bg='white', font=('Helvetica',
                            14, 'bold'))
frame_entry.grid(row=0, column=0, padx=10, pady=10, sticky='ns')

frame_buttons = tk.Frame(frame_main, padx=20, pady=10,
                         bg='lightgray')
frame_buttons.grid(row=1, column=0, padx=10, pady=10, sticky='ew')

frame_listbox = tk.Frame(frame_main, padx=20, pady=10,
                         bg='lightgray')
frame_listbox.grid(row=0, column=1, rowspan=2, padx=10, pady=10,
                    sticky='nsew')

# Labels and Entries
ttk.Label(frame_entry, text="Word:").grid(
    row=0, column=0, sticky='w', padx=5, pady=5)
entry_word = ttk.Entry(frame_entry, width=30)
entry_word.grid(row=0, column=1, padx=5, pady=5)

ttk.Label(frame_entry, text="Meaning:").grid(
    row=1, column=0, sticky='w', padx=5, pady=5)
entry_meaning = ttk.Entry(frame_entry, width=30)
entry_meaning.grid(row=1, column=1, padx=5, pady=5)

ttk.Label(frame_entry, text="Search:").grid(
```

```

    row=2, column=0, sticky='w', padx=5, pady=5)
entry_search = ttk.Entry(frame_entry, width=30)
entry_search.grid(row=2, column=1, padx=5, pady=5)

search_button = tk.Button(frame_entry, text="Search",
command=search_word, bg='#007bff',
fg='white', font=('Helvetica', 10, 'bold'), padx=10,
pady=5, relief='flat')
search_button.grid(row=2, column=2, padx=5, pady=5)

# Buttons
add_button = tk.Button(frame_buttons, text="Add",
command=add_word, bg='#28a745',
fg='white', font=('Helvetica', 10, 'bold'), padx=10,
pady=5, relief='flat')
add_button.grid(row=0, column=0, padx=5, pady=5, sticky='ew')

update_button = tk.Button(frame_buttons, text="Update",
command=update_word, bg='#17a2b8',
fg='white', font=('Helvetica', 10, 'bold'), padx=10,
pady=5, relief='flat')
update_button.grid(row=0, column=1, padx=5, pady=5, sticky='ew')

delete_button = tk.Button(frame_buttons, text="Delete",
command=delete_word, bg='#dc3545',
fg='white', font=('Helvetica', 10, 'bold'), padx=10,
pady=5, relief='flat')
delete_button.grid(row=0, column=2, padx=5, pady=5, sticky='ew')

clear_button = tk.Button(frame_buttons, text="Clear",
command=clear_fields, bg='#ffc107',
fg='black', font=('Helvetica', 10, 'bold'), padx=10,
pady=5, relief='flat')
clear_button.grid(row=0, column=3, padx=5, pady=5, sticky='ew')

# Listbox and Scrollbar
listbox_words = tk.Listbox(frame_listbox, width=80,

```

```
        height=15, selectmode=tk.EXTENDED,
fg='black', font=('Helvetica', 12, 'bold'))
listbox_words.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

scrollbar = tk.Scrollbar(
    frame_listbox, orient=tk.VERTICAL,
command=listbox_words.yview)

scrollbar=tk.Scrollbar(
    frame_listbox,orient=tk.HORIZONTAL,
command=listbox_words.xview)

scrollbar.pack(side=tk.RIGHT, fill=tk.Y)
listbox_words.config(yscrollcommand=scrollbar.set)
# Pagination buttons
next_button = tk.Button(frame_buttons, text="Next",
                        command=next_page, bg='#28a745', fg='white')
next_button.grid(row=1, column=0, padx=5, pady=5, sticky='ew')

previous_button = tk.Button(
    frame_buttons, text="Previous", command=previous_page,
bg='#dc3545', fg='white')
previous_button.grid(row=1, column=1, padx=5, pady=5, sticky='ew')

# Batch Update
# Add Batch Update Label
ttk.Label(frame_entry, text="Batch Update:", font=('Helvetica', 9,
'bold')).grid(
    row=3, column=0, sticky='w', padx=5, pady=5, columnspan=2)

# Batch Update Text Entry
entry_batch_update = tk.Text(frame_entry, width=40, height=10)
entry_batch_update.grid(row=4, column=0, padx=5, pady=5,
columnspan=2)

# Batch Update Button
batch_update_button = tk.Button(
```

```
    frame_buttons, text="Batch Update", command=batch_update,
    bg="#007bff", fg='white')
batch_update_button.grid(row=1, column=2, padx=5, pady=5,
sticky='ew')

# Import and Backup
import_button = tk.Button(frame_buttons, text="Import",
                           command=import_from_file, bg="#17a2b8",
                           fg='white')
import_button.grid(row=1, column=3, padx=5, pady=5, sticky='ew')

backup_button = tk.Button(frame_buttons, text="Backup",
                           command=backup_database, bg="#17a2b8",
                           fg='white')
backup_button.grid(row=2, column=0, padx=5, pady=5, sticky='ew')

restore_button = tk.Button(frame_buttons, text="Restore",
                           command=restore_database, bg='#ffc107',
                           fg='black')
restore_button.grid(row=2, column=1, padx=5, pady=5, sticky='ew')

# Status and Count Labels
status_label = tk.Label(frame_main, text="Welcome!",
                        font=('Helvetica', 10, 'bold'), fg='blue')
status_label.grid(row=2, column=0, padx=10, pady=8)

count_label = tk.Label(frame_main, text="Total Words: 0",
                       font=('Helvetica', 10, 'bold'), fg='black')
count_label.grid(row=3, column=0, padx=10, pady=10)

update_word_list()

root.mainloop()
```

Complete Guide for Setting Up and Running the Vocabulary Dictionary Application

Installation and Setup

1. Install Python:

- Download and install **Python** from the [official website](#). Make sure to check the box to add **Python** to your system PATH during installation.

2. Install Required Libraries:

- The application requires the tkinter library for the graphical user interface (GUI) and sqlite3 for database management. These libraries are included with **Python** by default. However, if you need to ensure they are installed or need additional libraries, use the following commands in your terminal or command prompt:

bash

Code:

pip install tkinter

3. Set Up the Development Environment in Visual Studio Code (VSCode):

- **Install VSCode:** Download and install VSCode from the [official website](#).

- **Open Your Project:** Open VSCode, and use File > Open Folder... to open the folder where your **Python** script is located.
- **Install Python Extension:** Search for the "**Python**" extension in the Extensions view (Ctrl+Shift+X) and install it to enable **Python**-specific features and linting.
- **Create a Virtual Environment** (optional but recommended):
 - Open a terminal in VSCode (Terminal > New Terminal) and run:

bash

Code:

Python -m venv **env**

- Activate the virtual environment:
 - On Windows:

bash

Code:

.\env\Scripts\activate

- On macOS/Linux:

bash

Code:

`source env/bin/activate`

4. Install Additional Dependencies:

- If there are additional dependencies, you can install them using pip:

bash

Code:

`pip install -r requirements.txt`

- Create a requirements.txt file in your project folder and add any extra libraries your project might need.

How to Run the Application

1. Run the Script:

- Open a terminal in VSCode (Terminal > New Terminal) and navigate to the directory where your script is located.
- Run the script using **Python**:

bash

Code:

Python your_script_name.py

2. Using the Application:

- When you run the script, a GUI window will appear where you can:
 - Add, update, and delete vocabulary words.

- Search for words and their meanings.
- Perform batch updates.
- Import vocabulary from text files.
- Backup and restore the vocabulary database.
- Navigate through pages of vocabulary entries.

Deep Dive into the Code

1. Imports and Initial Setup:

- **Libraries:**
 - tkinter and ttk: Used for creating the GUI.
 - askopenfilename from tkinter.filedialog: Allows file selection dialogs.
 - sqlite3: Handles database operations.
 - shutil: Provides file operations like copying.
 - datetime: Handles date and time functions.

Python

Code:

```
import tkinter as tk
from tkinter import ttk
from tkinter.filedialog import askopenfilename
import sqlite3
import shutil
import datetime
```

2. Database Functions:

- **create_connection:** Establishes a connection to the SQLite database.

- **create_table**: Creates the vocabulary table if it does not exist.
- **add_word_to_db**: Inserts or updates a word in the database.
- **delete_word_from_db**: Deletes a word from the database.
- **get_all_words_from_db**: Retrieves all words with optional pagination.
- **search_word_from_db**: Searches for words or meanings based on a query.

Python

Code:

```
def create_connection():
    try:
        conn = sqlite3.connect('vocabulary.db')
        return conn
    except sqlite3.Error as e:
        print(f"SQLite error: {e}")
        return None
```

3. GUI Functions:

- **Add Word**: Adds a new word to the database and updates the GUI.
- **Update Word**: Updates an existing word's details.

- **Delete Word:** Deletes selected words.
- **Show All:** Displays all words.
- **Search Word:** Searches for words and displays results.
- **Clear Fields:** Clears input fields and status messages.
- **Update Word List:** Updates the displayed list of words with pagination support.
- **Batch Update:** Updates multiple words at once from a text area.
- **Import From File:** Imports words from a text file into the database.
- **Backup Database:** Creates a backup of the current database.
- **Restore Database:** Restores the database from a backup.

Python

Code:

```
def add_word():
    word = entry_word.get().strip()
    meaning = entry_meaning.get().strip()
    if word and meaning:
        add_word_to_db(word, meaning)
        status_label.config(
```

```
    text=f"Added '{word}' to vocabulary.",  
foreground='green')  
    update_word_list()  
else:  
    status_label.config(  
        text="Both word and meaning are required.",  
foreground='red')
```

4. GUI Setup:

- **Window Setup:** Initializes the main window with dimensions and title.
- **Frames and Layout:** Organizes widgets into frames for better layout management.
- **Widgets:**
 - **Labels and Entries:** For user input.
 - **Buttons:** For performing actions like adding, updating, deleting, etc.
 - **Listbox and Scrollbar:** Displays the list of vocabulary entries with scrolling capability.
- **Pagination Buttons:** Navigate through pages of vocabulary entries.
- **Batch Update and Import/Backup Buttons:** For batch operations and file handling.

Python

Code:

```
# GUI setup
root = tk.Tk()
root.title("Vocabulary Dictionary")
root.geometry("1280x720+0+0") # Adjusted size for
better fitting
root.resizable(True, True)
```

Frames:

- frame_main, frame_entry, frame_buttons, and frame_listbox are used to structure the GUI into manageable sections.

Widgets:

- **Entry Widgets:** For user inputs like word, meaning, and search queries.
- **Buttons:** To trigger actions like adding, updating, and deleting words.
- **Listbox:** Displays the list of vocabulary entries.
- **Scrollbar:** Provides scrolling functionality for the listbox.

Pagination and Batch Operations:

- **Pagination:** Functions next_page and previous_page handle page navigation.

- **Batch Update:** Handles multiple updates from a text input.

Python

Code:

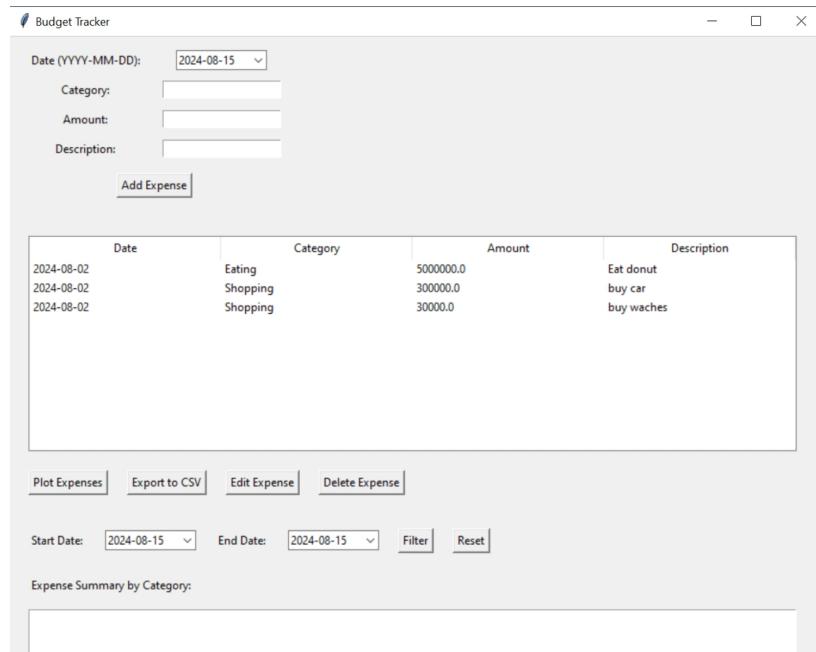
```
# Pagination buttons
next_button = tk.Button(frame_buttons, text="Next",
                        command=next_page, bg="#28a745",
                        fg='white')
next_button.grid(row=1, column=0, padx=5, pady=5,
                  sticky='ew')

previous_button = tk.Button(
    frame_buttons, text="Previous",
    command=previous_page, bg="#dc3545", fg='white')
previous_button.grid(row=1, column=1, padx=5,
                      pady=5, sticky='ew')
```

Summary

This vocabulary dictionary application leverages SQLite for data storage and tkinter for a user-friendly GUI. It provides functionalities for managing vocabulary entries, including adding, updating, deleting, searching, and batch updating words. It also supports importing from files, backing up, and restoring the database. This setup ensures that users can effectively manage their vocabulary list with an intuitive interface and robust backend operations.

Budget Tracker



```
import tkinter as tk
from tkinter import ttk, messagebox
import sqlite3
from matplotlib.figure import Figure
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from tkcalendar import DateEntry
import csv
from datetime import datetime

def init_db():
    conn = sqlite3.connect('expenses.db')
    c = conn.cursor()
    c.execute("""
        CREATE TABLE IF NOT EXISTS expenses (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            date TEXT,
            category TEXT,
            amount REAL
        )
    """)
```

```
        amount REAL,
        description TEXT
    )
")
conn.commit()
conn.close()

class BudgetTrackerApp(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("Budget Tracker")
        self.geometry("850x800+200+-20")

        # Initialize the database
        init_db()

        self.selected_expense_id = None

        self.create_widgets()
        self.populate_treeview()

    def create_widgets(self):
        # Create frames for different sections
        self.form_frame = tk.Frame(self)
        self.form_frame.grid(row=0, column=0, sticky="nsew", padx=10,
pady=10)

        self.treeview_frame = tk.Frame(self)
        self.treeview_frame.grid(
            row=1, column=0, sticky="nsew", padx=10, pady=10)
```

```
self.plot_frame = tk.Frame(self)

# Form for new expense
self.date_label = tk.Label(self.form_frame, text="Date (YYYY-MM-DD):")
self.date_label.grid(row=0, column=0, padx=10, pady=5)
self.date_entry = DateEntry(self.form_frame, date_pattern='yyyy-mm-dd')
self.date_entry.grid(row=0, column=1, padx=10, pady=5)

self.category_label = tk.Label(self.form_frame, text="Category:")
self.category_label.grid(row=1, column=0, padx=10, pady=5)
self.category_entry = tk.Entry(self.form_frame)
self.category_entry.grid(row=1, column=1, padx=10, pady=5)

self.amount_label = tk.Label(self.form_frame, text="Amount:")
self.amount_label.grid(row=2, column=0, padx=10, pady=5)
self.amount_entry = tk.Entry(self.form_frame)
self.amount_entry.grid(row=2, column=1, padx=10, pady=5)

self.description_label = tk.Label(self.form_frame,
text="Description:")
self.description_label.grid(row=3, column=0, padx=10, pady=5)
self.description_entry = tk.Entry(self.form_frame)
self.description_entry.grid(row=3, column=1, padx=10, pady=5)

self.add_button = tk.Button(
    self.form_frame, text="Add Expense",
    command=self.add_expense)
self.add_button.grid(row=4, column=0, columnspan=2, pady=10)

# Filter by date range
```

```
self.filter_frame = tk.Frame(self)
self.filter_frame.grid(
    row=2, column=0, sticky="nsew", padx=10, pady=10)

    self.start_date_label = tk.Label(self.filter_frame, text="Start
Date:")
        self.start_date_label.grid(row=0, column=0, padx=10, pady=5)
        self.start_date_entry = DateEntry(
            self.filter_frame, date_pattern='yyyy-mm-dd')
        self.start_date_entry.grid(row=0, column=1, padx=10, pady=5)

    self.end_date_label = tk.Label(self.filter_frame, text="End Date:")
        self.end_date_label.grid(row=0, column=2, padx=10, pady=5)
        self.end_date_entry = DateEntry(
            self.filter_frame, date_pattern='yyyy-mm-dd')
        self.end_date_entry.grid(row=0, column=3, padx=10, pady=5)

    self.filter_button = tk.Button(
        self.filter_frame, text="Filter", command=self.filter_expenses)
    self.filter_button.grid(row=0, column=4, padx=10, pady=5)

    self.reset_button = tk.Button(
        self.filter_frame, text="Reset",
command=self.populate_treeview)
    self.reset_button.grid(row=0, column=5, padx=10, pady=5)

# Treeview for displaying expenses
self.tree = ttk.Treeview(self.treeview_frame, columns=(
    "Date", "Category", "Amount", "Description"),
show='headings')
    self.tree.heading("Date", text="Date")
    self.tree.heading("Category", text="Category")
```

```
self.tree.heading("Amount", text="Amount")
self.tree.heading("Description", text="Description")
self.tree.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

self.tree.bind('<Double-1>', self.on_tree_select)
self.tree.bind('<<TreeviewSelect>>', self.on_tree_select)

# Plot button and navigation controls
self.plot_button = tk.Button(
    selftreeview_frame, text="Plot Expenses",
command=self.show_plot)
self.plot_button.pack(side=tk.LEFT, padx=10, pady=10)

self.export_button = tk.Button(
    selftreeview_frame, text="Export to CSV",
command=self.export_to_csv)
self.export_button.pack(side=tk.LEFT, padx=10, pady=10)

self.edit_button = tk.Button(
    selftreeview_frame, text="Edit Expense",
command=self.edit_expense)
self.edit_button.pack(side=tk.LEFT, padx=10, pady=10)

self.delete_button = tk.Button(
    selftreeview_frame, text="Delete Expense",
command=self.delete_expense)
self.delete_button.pack(side=tk.LEFT, padx=10, pady=10)

self.back_button = tk.Button(
    selfplot_frame, text="Back to Form",
command=self.show_form)
self.back_button.pack(side=tk.LEFT, padx=10, pady=10)
```

```
self.canvas = None

# Summary Frame
self.summary_frame = tk.Frame(self)
self.summary_frame.grid(
    row=3, column=0, sticky="nsew", padx=10, pady=3)

self.summary_label = tk.Label(
    self.summary_frame, text="Expense Summary by Category:")
self.summary_label.pack(anchor="w", padx=10, pady=5)

self.summary_text = tk.Text(
    self.summary_frame, height=10, state="disabled")
self.summary_text.pack(fill=tk.BOTH, expand=True, padx=10,
pady=10)

def add_expense(self):
    date = self.date_entry.get()
    category = self.category_entry.get()
    amount = self.amount_entry.get()
    description = self.description_entry.get()

    if not date or not category or not amount or not description:
        messagebox.showwarning("Input Error", "All fields are
required.")
        return

    try:
        amount = float(amount)
    except ValueError:
```

```
    messagebox.showwarning("Input Error", "Amount must be a
number.")
    return

    conn = sqlite3.connect('expenses.db')
    c = conn.cursor()
    c.execute('INSERT INTO expenses (date, category, amount,
description) VALUES (?, ?, ?, ?)',
              (date, category, amount, description))
    conn.commit()
    conn.close()

    self.populate_treeview()
    self.update_summary()
    self.clear_form()

def populate_treeview(self):
    for i in self.tree.get_children():
        self.tree.delete(i)

    conn = sqlite3.connect('expenses.db')
    c = conn.cursor()
    c.execute('SELECT id, date, category, amount, description FROM
expenses')
    for row in c.fetchall():
        self.tree.insert("", "end", values=row[1:], iid=row[0])
    conn.close()

def clear_form(self):
    # Set to current date instead of empty
    self.date_entry.set_date(datetime.now())
    self.category_entry.delete(0, tk.END)
```

```
    self.amount_entry.delete(0, tk.END)
    self.description_entry.delete(0, tk.END)
    self.selected_expense_id = None

def filter_expenses(self):
    start_date = self.start_date_entry.get()
    end_date = self.end_date_entry.get()

    for i in self.tree.get_children():
        self.tree.delete(i)

    conn = sqlite3.connect('expenses.db')
    c = conn.cursor()
    c.execute('SELECT id, date, category, amount, description FROM expenses WHERE date BETWEEN ? AND ?',
              (start_date, end_date))
    for row in c.fetchall():
        self.tree.insert("", "end", values=row[1:], iid=row[0])
    conn.close()

def on_tree_select(self, event):
    selected_item = self.tree.selection()
    if selected_item:
        self.selected_expense_id = selected_item[0]
        # Debug print
        print(f"Selected expense ID: {self.selected_expense_id}")
        conn = sqlite3.connect('expenses.db')
        c = conn.cursor()
        c.execute('SELECT date, category, amount, description FROM expenses WHERE id = ?',
                  (self.selected_expense_id,))
        expense = c.fetchone()
```

```
conn.close()

    self.date_entry.set_date(expense[0])
    self.category_entry.delete(0, tk.END)
    self.category_entry.insert(0, expense[1])
    self.amount_entry.delete(0, tk.END)
    self.amount_entry.insert(0, expense[2])
    self.description_entry.delete(0, tk.END)
    self.description_entry.insert(0, expense[3])

def edit_expense(self):
    if not self.selected_expense_id:
        messagebox.showwarning(
            "Selection Error", "Please select an expense to edit.")
        return

    date = self.date_entry.get()
    category = self.category_entry.get()
    amount = self.amount_entry.get()
    description = self.description_entry.get()

    if not date or not category or not amount or not description:
        messagebox.showwarning("Input Error", "All fields are
required.")
        return

    try:
        amount = float(amount)
    except ValueError:
        messagebox.showwarning("Input Error", "Amount must be a
number.")
    return
```

```
conn = sqlite3.connect('expenses.db')
c = conn.cursor()
c.execute('UPDATE expenses SET date = ?, category = ?, amount
= ?, description = ? WHERE id = ?',
          (date, category, amount, description,
           self.selected_expense_id))
conn.commit()
conn.close()

self.populate_treeview()
self.update_summary()
self.clear_form()

def delete_expense(self):
    if not self.selected_expense_id:
        messagebox.showwarning(
            "Selection Error", "Please select an expense to delete.")
    return

# Debug print
print(f"Deleting expense ID: {self.selected_expense_id}")

conn = sqlite3.connect('expenses.db')
c = conn.cursor()
c.execute('DELETE FROM expenses WHERE id = ?',
          (self.selected_expense_id,))
conn.commit()
conn.close()

self.populate_treeview()
```

```
self.update_summary()
self.clear_form()

def show_plot(self):
    self.plot_frame.grid(row=0, column=0, rowspan=4,
                         sticky="nsew", padx=10, pady=10)
    self.form_frame.grid_forget()
    selftreeview_frame.grid_forget()
    self.filter_frame.grid_forget()
    self.summary_frame.grid_forget()

    conn = sqlite3.connect('expenses.db')
    c = conn.cursor()
    c.execute('SELECT category, SUM(amount) FROM expenses
GROUP BY category')
    data = c.fetchall()
    conn.close()

    categories = [row[0] for row in data]
    amounts = [row[1] for row in data]

    fig = Figure(figsize=(8, 6))
    ax = fig.add_subplot(111)
    ax.pie(amounts, labels=categories, autopct='%.1f%%',
startangle=90)
    ax.set_title("Expenses by Category")

    if self.canvas:
        self.canvas.get_tk_widget().pack_forget()

    self.canvas = FigureCanvasTkAgg(fig, master=self.plot_frame)
```

```
self.canvas.draw()
self.canvas.get_tk_widget().pack()

def show_form(self):
    self.plot_frame.grid_forget()
    self.form_frame.grid(row=0, column=0, sticky="nsew", padx=10,
pady=10)
    selftreeview_frame.grid(
        row=1, column=0, sticky="nsew", padx=10, pady=10)
    self.filter_frame.grid(
        row=2, column=0, sticky="nsew", padx=10, pady=10)
    self.summary_frame.grid(
        row=3, column=0, sticky="nsew", padx=10, pady=10)

def export_to_csv(self):
    conn = sqlite3.connect('expenses.db')
    c = conn.cursor()
    c.execute('SELECT date, category, amount, description FROM
expenses')
    data = c.fetchall()
    conn.close()

    with open('expenses.csv', 'w', newline="") as f:
        writer = csv.writer(f)
        writer.writerow(['Date', 'Category', 'Amount', 'Description'])
        writer.writerows(data)

    messagebox.showinfo("Export Successful",
                        "Expenses exported to expenses.csv")

def update_summary(self):
    conn = sqlite3.connect('expenses.db')
```

```
c = conn.cursor()
    c.execute('SELECT category, SUM(amount) FROM expenses
GROUP BY category')
    data = c.fetchall()
    conn.close()

    self.summary_text.config(state="normal")
    self.summary_text.delete(1.0, tk.END)
    for category, amount in data:
        self.summary_text.insert(tk.END, f"{category}: {amount}\n")
    self.summary_text.config(state="disabled")

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

Code Breakdown

1. Imports:

- tkinter and tkinter.ttk: Used for creating the GUI.
- sqlite3: Provides an interface for interacting with the SQLite database.
- matplotlib: Used for creating visual plots.
- tkcalendar.DateEntry: Allows users to pick dates from a calendar widget.
- csv: For exporting data to CSV files.

- datetime: For handling date and time.

2. Database Initialization (init_db function): Python

```
def init_db():
    conn = sqlite3.connect('expenses.db')
    c = conn.cursor()
    c.execute("""
        CREATE TABLE IF NOT EXISTS expenses (
            id INTEGER PRIMARY KEY
            AUTOINCREMENT,
            date TEXT,
            category TEXT,
            amount REAL,
            description TEXT
        )
    """)
    conn.commit()
    conn.close()
```

- **Purpose:** Initializes the SQLite database and creates the expenses table if it does not already exist.
- **Table Schema:**
 - id: Primary key, auto-incremented integer.

- date: Date of the expense.
- category: Category of the expense.
- amount: Amount spent.
- description: Description of the expense.

3. Application Class (BudgetTrackerApp):

The BudgetTrackerApp class inherits from tk.Tk and represents the main application window. It includes various methods and components for handling user interactions and data.

Constructor (`__init__`):

Python

```
def __init__(self):
    super().__init__()
    self.title("Budget Tracker")
    self.geometry("850x800+200+-20")
    init_db()
    self.selected_expense_id = None
    self.create_widgets()
    self.populate_treeview()
```

- **Window Configuration:** Sets the title and size of the main window.
- **Database Initialization:** Calls `init_db` to ensure the database is ready.
- **Widget Creation and Data Population:** Calls `create_widgets` to build the GUI and `populate_treeview` to display existing expenses.

Creating Widgets (`create_widgets`):

Python

```
def create_widgets(self):
    # Create frames for different sections
    self.form_frame = tk.Frame(self)
```

```
    self.form_frame.grid(row=0, column=0,
sticky="nsew", padx=10, pady=10)

    selftreeview_frame = tk.Frame(self)
    selftreeview_frame.grid(row=1, column=0,
sticky="nsew", padx=10, pady=10)

    self.plot_frame = tk.Frame(self)

    # Form for new expense
    self.date_label = tk.Label(self.form_frame,
text="Date (YYYY-MM-DD):")
    self.date_label.grid(row=0, column=0, padx=10,
pady=5)
    self.date_entry = DateEntry(self.form_frame,
date_pattern='yyyy-mm-dd')
    self.date_entry.grid(row=0, column=1, padx=10,
pady=5)

    self.category_label = tk.Label(self.form_frame,
text="Category:")
    self.category_label.grid(row=1, column=0,
padx=10, pady=5)
    self.category_entry = tk.Entry(self.form_frame)
    self.category_entry.grid(row=1, column=1,
padx=10, pady=5)

    self.amount_label = tk.Label(self.form_frame,
text="Amount:")
```

```
    self.amount_label.grid(row=2, column=0, padx=10,
pady=5)
    self.amount_entry = tk.Entry(self.form_frame)
    self.amount_entry.grid(row=2, column=1, padx=10,
pady=5)

    self.description_label = tk.Label(self.form_frame,
text="Description:")
    self.description_label.grid(row=3, column=0,
padx=10, pady=5)
    self.description_entry = tk.Entry(self.form_frame)
    self.description_entry.grid(row=3, column=1,
padx=10, pady=5)

    self.add_button = tk.Button(self.form_frame,
text="Add Expense", command=self.add_expense)
    self.add_button.grid(row=4, column=0,
columnspan=2, pady=10)

# Filter by date range
    self.filter_frame = tk.Frame(self)
    self.filter_frame.grid(row=2, column=0,
sticky="nsew", padx=10, pady=10)

    self.start_date_label = tk.Label(self.filter_frame,
text="Start Date:")
    self.start_date_label.grid(row=0, column=0,
padx=10, pady=5)
```

```
    self.start_date_entry = DateEntry(self.filter_frame,
date_pattern='yyyy-mm-dd')
    self.start_date_entry.grid(row=0, column=1,
padx=10, pady=5)

    self.end_date_label = tk.Label(self.filter_frame,
text="End Date:")
    self.end_date_label.grid(row=0, column=2,
padx=10, pady=5)

    self.end_date_entry = DateEntry(self.filter_frame,
date_pattern='yyyy-mm-dd')
    self.end_date_entry.grid(row=0, column=3,
padx=10, pady=5)

    self.filter_button = tk.Button(self.filter_frame,
text="Filter", command=self.filter_expenses)
    self.filter_button.grid(row=0, column=4, padx=10,
pady=5)

    self.reset_button = tk.Button(self.filter_frame,
text="Reset", command=self.populate_treeview)
    self.reset_button.grid(row=0, column=5, padx=10,
pady=5)

    # Treeview for displaying expenses
    self.tree = ttk.Treeview(self.treeview_frame,
columns=("Date", "Category", "Amount",
"Description"), show='headings')
```

```
    self.tree.heading("Date", text="Date")
    self.tree.heading("Category", text="Category")
    self.tree.heading("Amount", text="Amount")
    self.tree.heading("Description", text="Description")
    self.tree.pack(fill=tk.BOTH, expand=True,
    padx=10, pady=10)

        self.tree.bind('<Double-1>', self.on_tree_select)
        self.tree.bind('<<TreeviewSelect>>',
self.on_tree_select)

    # Plot button and navigation controls
    self.plot_button = tk.Button(selftreeview_frame,
text="Plot Expenses", command=self.show_plot)
    self.plot_button.pack(side=tk.LEFT, padx=10,
pady=10)

    self.export_button = tk.Button(selftreeview_frame,
text="Export to CSV", command=self.export_to_csv)
    self.export_button.pack(side=tk.LEFT, padx=10,
pady=10)

    self.edit_button = tk.Button(selftreeview_frame,
text="Edit Expense", command=self.edit_expense)
    self.edit_button.pack(side=tk.LEFT, padx=10,
pady=10)

    self.delete_button = tk.Button(selftreeview_frame,
text="Delete Expense",
```

```

        command=self.delete_expense)
    self.delete_button.pack(side=tk.LEFT, padx=10,
                           pady=10)

    self.back_button = tk.Button(self.plot_frame,
                                text="Back to Form", command=self.show_form)
    self.back_button.pack(side=tk.LEFT, padx=10,
                           pady=10)

    self.canvas = None

    # Summary Frame
    self.summary_frame = tk.Frame(self)
    self.summary_frame.grid(row=3, column=0,
                           sticky="nsew", padx=10, pady=3)

    self.summary_label = tk.Label(self.summary_frame,
                                text="Expense Summary by Category:")
    self.summary_label.pack(anchor="w", padx=10,
                           pady=5)

    self.summary_text = tk.Text(self.summary_frame,
                               height=10, state="disabled")
    self.summary_text.pack(fill=tk.BOTH,
                           expand=True, padx=10, pady=10)

```

- **Frames:** Organizes the layout into different sections:

- **form_frame**: For entering new expenses.
- **treeview_frame**: Displays expenses in a table and provides options for plotting, exporting, editing, and deleting.
- **filter_frame**: Allows filtering expenses by date range.
- **plot_frame**: Displays the pie chart for expenses.
- **summary_frame**: Shows a textual summary of expenses by category.

Expense Management Methods:

- **add_expense**: Adds a new expense to the database and updates the GUI.
- **populate_treeview**: Fills the treeview with expenses from the database.
- **clear_form**: Clears the input fields and resets the selected expense ID.
- **filter_expenses**: Filters and displays expenses based on the selected date range.
- **on_tree_select**: Populates the form with details of the selected expense.
- **edit_expense**: Edits the selected expense.
- **delete_expense**: Deletes the selected expense.

- **show_plot**: Displays a pie chart of expenses by category.
- **show_form**: Switches back to the main form view.
- **export_to_csv**: Exports expenses to a CSV file.
- **update_summary**: Updates the summary of expenses by category.

4. Running the Application:

To run this application, follow these steps:

1. **Ensure Dependencies**: Install necessary libraries if they are not already installed. You can do this using **pip**:

bash::

pip install matplotlib tkcalendar

2. **Save the Code**: Save the code into a **Python** file, e.g., budget_tracker.py.
3. **Run the Code**: Execute the script using **Python**

bash::

Python budget_tracker.py

4. Using the Application:

- **Add Expenses**: Fill out the form and click "Add Expense."

- **Filter Expenses:** Select a date range and click "Filter."
- **Edit/Delete Expenses:** Double-click on an entry in the table or use the respective buttons.
- **Plot Data:** Click "Plot Expenses" to view the pie chart.
- **Export Data:** Click "Export to CSV" to save data as a CSV file.
- **Navigate:** Use the "Back to Form" button to return to the expense entry form.

By following these instructions, you should be able to effectively use and manage your budget tracking application. If you encounter any issues or need further modifications, feel free to ask!

4o mini

You said:

step by step install with vscode

ChatGPT said:

ChatGPT

To set up and run the budget tracker application using Visual Studio Code (VSCode), follow these step-by-step instructions:

Step 1: Install Visual Studio Code

1. Download and Install VSCode:

- Go to the [Visual Studio Code download page](#).
- Download the installer for your operating system (Windows, macOS, or Linux).
- Run the installer and follow the setup instructions.

Step 2: Install Python

1. Download and Install Python

- Go to the [Python download page](#).
- Download the latest version of **Python** (**Python** 3.x).
- Run the installer and ensure the option "Add **Python** to PATH" is checked.
Follow the setup instructions.

Step 3: Install Python Extensions in VSCode

1. Open VSCode:

- Launch Visual Studio Code from your applications or start menu.

2. Install Python Extension:

- Click on the Extensions view icon in the Sidebar (or press Ctrl+Shift+X).

- Search for "**Python**" and install the extension provided by Microsoft.

Step 4: Set Up a **Python** Virtual Environment

1. Open a Terminal in VSCode:

- Open the integrated terminal by selecting Terminal > New Terminal from the top menu or pressing **Ctrl+`**.

2. Create a Virtual Environment:

- Navigate to the folder where you want to save your project (or create a new folder for your project):

bash::

```
mkdir budget_tracker  
cd budget_tracker
```

- Create a virtual environment:

bash::

```
Python -m venv venv
```

3. Activate the Virtual Environment:

- On Windows:

bash::

```
.\venv\Scripts\activate
```

- On macOS/Linux:

bash::

```
source venv/bin/activate
```

Step 5: Install Required Packages

1. Install Required Packages:

- Ensure the virtual environment is activated.
- Install the necessary **Python** packages by running:

bash::

```
pip install matplotlib tkcalendar
```

Step 6: Create the **Python** Script

1. Create a New **Python** File:

- In VSCode, create a new file named budget_tracker.py in your project folder.

2. Copy the Code:

- Paste the provided **Python** code into budget_tracker.py and save the file.

Step 7: Run the Application

1. Run the Python Script:

- In the terminal, with the virtual environment activated, run the script:

bash::

Python budget_tracker.py

2. Interact with the Application:

- The application window should open. You can now use it to add expenses, filter data, view plots, and export data.

Optional: Debugging and Enhancements

1. Debugging:

- Set breakpoints in your code by clicking in the gutter next to line numbers.
- Use the Debug view (click on the Debug icon in the Sidebar or press Ctrl+Shift+D) to start debugging.

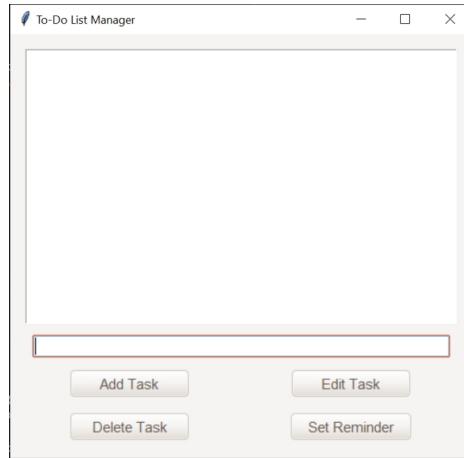
2. Enhancements:

- Customize VSCode settings and preferences by going to File > Preferences > Settings (or pressing Ctrl+,).

Additional Tips

- **Version Control:** Consider using Git for version control. Initialize a Git repository in your project folder and commit your code changes.
- **Documentation:** Use VSCode's Markdown support to create project documentation in a README.md file.

To Do List Manager



```
import tkinter as tk
from tkinter import messagebox, simpledialog
from tkinter import ttk
from tkthemes import ThemedTk
from tkcalendar import DateEntry
import datetime
import threading
import time

class ToDoListManager:
    def __init__(self, root):
        self.root = root
        self.root.title("To-Do List Manager")

        self.tasks = []

        self.frame = ttk.Frame(root, padding="10")
        self.frame.pack(fill=tk.BOTH, expand=True)

        self.task_listbox = tk.Listbox(
            self.frame, height=15, width=50, font=('Helvetica', 12))
```

```
    self.task_listbox.grid(row=0, column=0, columnspan=2, padx=5,
pady=5)

    self.entry_task = ttk.Entry(
        self.frame, width=48, font=('Helvetica', 12))
    self.entry_task.grid(row=1, column=0, columnspan=2, padx=5,
pady=5)

    self.button_add = ttk.Button(
        self.frame, text="Add Task", command=self.add_task)
    self.button_add.grid(row=2, column=0, padx=5, pady=5)

    self.button_edit = ttk.Button(
        self.frame, text="Edit Task", command=self.edit_task)
    self.button_edit.grid(row=2, column=1, padx=5, pady=5)

    self.button_delete = ttk.Button(
        self.frame, text="Delete Task", command=self.delete_task)
    self.button_delete.grid(row=3, column=0, padx=5, pady=5)

    self.button_reminder = ttk.Button(
        self.frame, text="Set Reminder", command=self.setReminder)
    self.button_reminder.grid(row=3, column=1, padx=5, pady=5)

    self.startReminderThread()

def add_task(self):
    task = self.entry_task.get()
    if task:
        self.tasks.append({"task": task, "reminder": None})
        self.updateTaskListbox()
```

```
    self.entry_task.delete(0, tk.END)
else:
    messagebox.showwarning("Warning", "You must enter a task.")

def edit_task(self):
    selected_task_index = self.task_listbox.curselection()
    if selected_task_index:
        new_task = simpledialog.askstring(
            "Edit Task", "Edit the selected task:",
            initialvalue=self.tasks[selected_task_index[0]]["task"])
        if new_task:
            self.tasks[selected_task_index[0]]["task"] = new_task
            self.update_task_listbox()
    else:
        messagebox.showwarning(
            "Warning", "You must select a task to edit.")

def delete_task(self):
    selected_task_index = self.task_listbox.curselection()
    if selected_task_index:
        del self.tasks[selected_task_index[0]]
        self.update_task_listbox()
    else:
        messagebox.showwarning(
            "Warning", "You must select a task to delete.")

def setReminder(self):
    selected_task_index = self.task_listbox.curselection()
    if selected_task_index:
        reminder_dialog = ReminderDialog(self.root)
        self.root.wait_window(reminder_dialog.top)
```

```
reminder_date = reminder_dialog.reminder_date
reminder_time = reminder_dialog.reminder_time

if reminder_date and reminder_time:
    reminder_datetime_str = f"{reminder_date}"
{reminder_time}"
try:
    reminder_datetime = datetime.datetime.strptime(
        reminder_datetime_str, "%Y-%m-%d %H:%M")
    self.tasks[selected_task_index[0]]
        ]["reminder"] = reminder_datetime
    self.update_task_listbox()
except ValueError:
    messagebox.showerror(
        "Error", "Invalid date or time format. Please try
again.")
else:
    messagebox.showwarning(
        "Warning", "You must select a task to set a reminder.")

def update_task_listbox(self):
    self.task_listbox.delete(0, tk.END)
    for task in self.tasks:
        reminder_str = f" (Reminder: {task['reminder']})" if
task['reminder'] else ""
        self.task_listbox.insert(tk.END, f"{task['task']}"
{reminder_str}")

def start_reminder_thread(self):
    reminder_thread = threading.Thread(
        target=self.check_reminders, daemon=True)
    reminder_thread.start()
```

```
def check_reminders(self):
    while True:
        current_time = datetime.datetime.now()
        for task in self.tasks:
            if task["reminder"] and current_time >= task["reminder"]:
                messagebox.showinfo(
                    "Reminder", f"Reminder for task: {task['task']} ")
                task["reminder"] = None # Reset reminder after
showing
        self.update_task_listbox()
        time.sleep(1) # Check reminders every second
class ReminderDialog:
    def __init__(self, parent):
        top = self.top = tk.Toplevel(parent)
        top.title("Set Reminder")

        ttk.Label(top, text="Select Reminder Date:").pack(pady=5)
        self.calendar = DateEntry(top, date_pattern="y-mm-dd")
        self.calendar.pack(pady=5)

        ttk.Label(top, text="Enter Reminder Time
(HH:MM):").pack(pady=5)
        self.time_entry = ttk.Entry(top)
        self.time_entry.pack(pady=5)

        self.button_set = ttk.Button(
            top, text="Set", command=self.set_reminder)
        self.button_set.pack(pady=5)

        self.reminder_date = None
        self.reminder_time = None
```

```
def setReminder(self):
    self.reminder_date = self.calendar.get_date()
    self.reminder_time = self.time_entry.get()
    self.top.destroy()

if __name__ == "__main__":
    root = ThemedTk(theme="radiance")
    app = ToDoListManager(root)
    root.mainloop()
```

How to Use the To-Do List Manager Program

The To-Do List Manager is a simple yet effective tool to help you organize and manage your tasks with an additional reminder feature. Below, you'll find a detailed guide on how to use the program.

Features

- 1. Add Task:** Allows you to add a new task to your to-do list.
- 2. Edit Task:** Enables you to modify an existing task.
- 3. Delete Task:** Lets you remove a selected task from the list.
- 4. Set Reminder:** Allows you to set a reminder for a selected task. You can specify both the date and time.

Using the Program

1. Adding a Task:

- Enter your task in the text entry box at the bottom of the main window.
- Click the "Add Task" button to add it to the list.
- Your new task will appear in the list box.

2. Editing a Task:

- Click on a task in the list box to select it.
- Click the "Edit Task" button.
- A dialog will appear allowing you to edit the selected task.
- Enter the new task details and click "OK" to save the changes.

3. Deleting a Task:

- Click on the task you want to remove in the list box.
- Click the "Delete Task" button.
- The selected task will be removed from the list.

4. Setting a Reminder:

- Click on a task in the list box to select it.
- Click the "Set Reminder" button.

- A new dialog will appear where you can select a reminder date and enter a time.
- Click "Set" to save the reminder.
- The reminder time will be displayed next to the task in the list.

5. Reminder Notifications:

- The program continuously checks for reminders in the background.
- If the current time matches a reminder's time, a notification will pop up.
- The reminder will be cleared from the list after it triggers.

Running the Program with Visual Studio Code (VSCode)

To run the To-Do List Manager program using VSCode, follow these steps:

Step 1: Install Visual Studio Code

1. Download and install [Visual Studio Code](#) if you haven't already.

Step 2: Install Python

1. Ensure that [Python](#) is installed on your system.
You can download it from [Python.org](#).

Step 3: Install Necessary Packages

1. Open a terminal in VSCode by selecting Terminal > New Terminal from the menu.
2. Install the required **Python** packages using **pip**. Run the following commands:

bash::

Code:

pip install tk

pip install ttkthemes

pip install tkcalendar

These commands install the tkinter, ttkthemes, and tkcalendar packages necessary for running the program.

Step 4: Open the Project in VSCode

1. Launch VSCode.
2. Open the folder containing your **Python** script by selecting File > Open Folder... and navigating to your project folder.
3. In the Explorer sidebar, locate and click on the ToDoListManager.py file to open it in the editor.

Step 5: Run the Program

1. Ensure your **Python** interpreter is correctly set. You can check or change the **Python** interpreter by clicking on the interpreter version in the

bottom left corner of VSCode and selecting the correct **Python** environment.

2. To run the script, open the integrated terminal in VSCode by selecting Terminal > New Terminal.
3. In the terminal, run the script by executing:

bash::

Code:

Python ToDoListManager.py

4. The application window should open, displaying the To-Do List Manager interface.

Troubleshooting

- **Python not recognized:** If you get an error indicating that **Python** is not recognized, ensure that **Python** is installed and added to your system's PATH environment variable.
- **Module not found:** If you encounter errors about missing modules, ensure that you have installed all necessary packages using **pip** as mentioned in Step 3.
- **VSCode settings:** If VSCode does not automatically detect the correct **Python** interpreter, you may need to manually select it by opening the Command Palette (Ctrl+Shift+P), typing **Python Select**

Interpreter, and choosing the correct **Python** environment.

By following these steps, you should be able to set up and run the To-Do List Manager program smoothly in Visual Studio Code. If you encounter any issues, make sure to check the console output for error messages, which can help diagnose and resolve common problems.

Let's break down the To-Do List Manager code step-by-step to understand how it works in detail:

1. Import Statements

Python
Code:

```
import tkinter as tk
from tkinter import messagebox, simpledialog
from tkinter import ttk
from ttkthemes import ThemedTk
from tkcalendar import DateEntry
import datetime
import threading
import time
```

- **tkinter**: Provides tools to create GUI elements in **Python**.

- **messagebox, simpledialog**: Modules from tkinter for displaying message boxes and simple input dialogs.
- **ttk**: Provides themed widgets for a more modern look.
- **ThemedTk**: A class from ttkthemes that allows you to use different themes.
- **DateEntry**: A widget from tkcalendar to select dates.
- **datetime**: For handling dates and times.
- **threading**: To run tasks in the background, allowing for non-blocking operations.
- **time**: Provides time-related functions, like sleeping to control how frequently reminders are checked.

2. ToDoListManager Class

Constructor (`__init__` Method)

Python

Code:

```
class ToDoListManager:
    def __init__(self, root):
        self.root = root
        self.root.title("To-Do List Manager")
        self.tasks = []
```

- **self.root**: The main window of the application.
- **self.tasks**: A list to store tasks, where each task is a dictionary containing task details and reminder information.

GUI Layout

Python Code:

```
self.frame = ttk.Frame(root, padding="10")
self.frame.pack(fill=tk.BOTH, expand=True)
```

- **ttk.Frame**: A container widget for organizing other widgets.
- **pack**: Places the frame in the root window and makes it expand to fill available space.

Python Code:

```
self.task_listbox = tk.Listbox(self.frame,
height=15, width=50, font=('Helvetica', 12))
self.task_listbox.grid(row=0, column=0,
columnspan=2, padx=5, pady=5)
```

- **tk.Listbox**: Displays a list of tasks.
- **grid**: Uses a grid layout to place the list box in the frame.

Python Code:

```
    self.entry_task = ttk.Entry(self.frame, width=48,  
font=('Helvetica', 12))  
    self.entry_task.grid(row=1, column=0,  
columnspan=2, padx=5, pady=5)  
• ttk.Entry: An entry widget where users can  
type a new task.
```

Python

Code:

```
    self.button_add = ttk.Button(self.frame,  
text="Add Task", command=self.add_task)  
    self.button_add.grid(row=2, column=0, padx=5,  
pady=5)  
  
    self.button_edit = ttk.Button(self.frame,  
text="Edit Task", command=self.edit_task)  
    self.button_edit.grid(row=2, column=1, padx=5,  
pady=5)  
  
    self.button_delete = ttk.Button(self.frame,  
text="Delete Task", command=self.delete_task)  
    self.button_delete.grid(row=3, column=0,  
padx=5, pady=5)  
  
    self.button_reminder = ttk.Button(self.frame,  
text="Set Reminder", command=self.setReminder)  
    self.button_reminder.grid(row=3, column=1,  
padx=5, pady=5)
```

- **ttk.Button**: Buttons for adding, editing, deleting tasks, and setting reminders. Each button is linked to a method that handles the corresponding action.

Reminder Thread

Python Code:

- ```
self.start_reminder_thread()
```
- **start\_reminder\_thread**: Starts a background thread to continuously check for reminders.

## 3. Task Management Methods

### Adding a Task

**Python Code:**

```
def add_task(self):
 task = self.entry_task.get()
 if task:
 self.tasks.append({"task": task, "reminder": None})
 self.update_task_listbox()
 self.entry_task.delete(0, tk.END)
 else:
 messagebox.showwarning("Warning", "You must enter a task.")
```

- **self.entry\_task.get()**: Retrieves the task entered by the user.
- **self.tasks.append**: Adds the new task to the list.
- **self.update\_task\_listbox()**: Refreshes the displayed list of tasks.
- **messagebox.showwarning**: Shows a warning if no task is entered.

## Editing a Task

**Python**

**Code:**

```
def edit_task(self):
 selected_task_index =
self.task_listbox.curselection()
 if selected_task_index:
 new_task = simpledialog.askstring("Edit Task",
"Edit the selected task:",
initialvalue=self.tasks[selected_task_index[0]]["task"])
 if new_task:
 self.tasks[selected_task_index[0]]["task"] =
new_task
 self.update_task_listbox()
 else:
```

```
 messagebox.showwarning("Warning", "You
must select a task to edit.")
```

- **self.task\_listbox.curselection()**: Gets the index of the selected task.
- **simpledialog.askstring**: Opens a dialog for editing the task.
- **self.tasks[selected\_task\_index[0]]["task"] = new\_task**: Updates the task in the list.
- **messagebox.showwarning**: Shows a warning if no task is selected.

## Deleting a Task

**Python**

**Code:**

```
def delete_task(self):
 selected_task_index =
 self.task_listbox.curselection()
 if selected_task_index:
 del self.tasks[selected_task_index[0]]
 self.update_task_listbox()
 else:
 messagebox.showwarning("Warning", "You
must select a task to delete.")
```

- **del self.tasks[selected\_task\_index[0]]**: Deletes the selected task from the list.

- **self.update\_task\_listbox()**: Refreshes the displayed list of tasks.

## Setting a Reminder

**Python**

**Code:**

```
def setReminder(self):
 selected_task_index =
 self.task_listbox.curselection()
 if selected_task_index:
 reminder_dialog =
 ReminderDialog(self.root)
 self.root.wait_window(reminder_dialog.top)

 reminder_date =
 reminder_dialog.reminder_date
 reminder_time =
 reminder_dialog.reminder_time

 if reminder_date and reminder_time:
 reminder_datetime_str = f'{reminder_date}'
 {reminder_time}'
 try:
 reminder_datetime =
 datetime.datetime.strptime(reminder_datetime_str,
 "%Y-%m-%d %H:%M")
```

```
 self.tasks[selected_task_index[0]]
 ["reminder"] = reminder_datetime
 self.update_task_listbox()
 except ValueError:
 messagebox.showerror("Error", "Invalid
date or time format. Please try again.")
 else:
 messagebox.showwarning("Warning", "You
must select a task to set a reminder.")
```

- **ReminderDialog(self.root)**: Opens a dialog to set a reminder date and time.
- **self.root.wait\_window(reminder\_dialog.top)**: Waits for the dialog to close before proceeding.
- **datetime.datetime.strptime**: Converts the reminder date and time string into a datetime object.
- **self.tasks[selected\_task\_index[0]]**  
**["reminder"] = reminder\_datetime**: Updates the task with the new reminder time.
- **messagebox.showerror**: Shows an error if the date or time format is incorrect.

## Updating the Task Listbox

Python

Code:

```
def update_task_listbox(self):
 self.task_listbox.delete(0, tk.END)
 for task in self.tasks:
 reminder_str = f" (Reminder: {task['reminder']})" if task['reminder'] else ""
 self.task_listbox.insert(tk.END, f"{task['task']}{reminder_str}")
```

- **self.task\_listbox.delete(0, tk.END)**: Clears the existing list.
- **self.task\_listbox.insert(tk.END, f" {task['task']}{reminder\_str}")**: Adds each task, including its reminder if it exists, to the list box.

## Starting Reminder Thread

Python

Code:

```
def start_reminder_thread(self):
 reminder_thread =
 threading.Thread(target=self.check_reminders,
 daemon=True)
 reminder_thread.start()
```

- **threading.Thread**: Creates a new thread to run the check\_reminders method in the background.
- **daemon=True**: Ensures the thread will not block the program from exiting.

## Checking Reminders

**Python**

**Code:**

```
def check_reminders(self):
 while True:
 current_time = datetime.datetime.now()
 for task in self.tasks:
 if task["reminder"] and current_time >=
task["reminder"]:
 messagebox.showinfo("Reminder",
f"Reminder for task: {task['task']}")

 task["reminder"] = None # Reset reminder

after showing
 self.update_task_listbox()

time.sleep(1) # Check reminders every second
```

- **datetime.datetime.now()**: Gets the current time.
- **task["reminder"] and current\_time >= task["reminder"]**: Checks if the reminder time

has arrived.

- **messagebox.showinfo**: Displays a reminder notification.
- **time.sleep(1)**: Pauses for 1 second between checks to avoid constant CPU usage.

## 4. ReminderDialog Class

### Constructor

#### Python

#### Code:

```
class ReminderDialog:
```

```
 def __init__(self, parent):
 self.top = tk.Toplevel(parent)
 self.top.title("Set Reminder")

 self.reminder_date = None
 self.reminder_time = None
```

- **self.top**: Creates a new top-level window for the reminder dialog.

### GUI Layout

#### Python

#### Code:

```
 ttk.Label(self.top, text="Date (YYYY-MM-
DD):").grid(row=0, column=0, padx=5, pady=5)
 self.entry_date = ttk.Entry(self.top)
```

```
 self.entry_date.grid(row=0, column=1, padx=5,
pady=5)

 ttk.Label(self.top, text="Time
(HH:MM):").grid(row=1, column=0, padx=5, pady=5)
 self.entry_time = ttk.Entry(self.top)
 self.entry_time.grid(row=1, column=1, padx=5,
pady=5)

 ttk.Button(self.top, text="Set",
command=self.set_reminder).grid(row=2, column=0,
columnspan=2, padx=5, pady=5)
```

- **ttk.Label**: Labels for date and time input fields.
- **ttk.Entry**: Entry fields for entering the reminder date and time.

## Setting the Reminder

### Python Code:

```
def setReminder(self):
 self.reminder_date = self.entry_date.get()
 self.reminder_time = self.entry_time.get()
 self.top.destroy()
```

- **self.entry\_date.get()**: Retrieves the entered date.

- **self.entry\_time.get()**: Retrieves the entered time.
- **self.top.destroy()**: Closes the reminder dialog.

## Running the Application

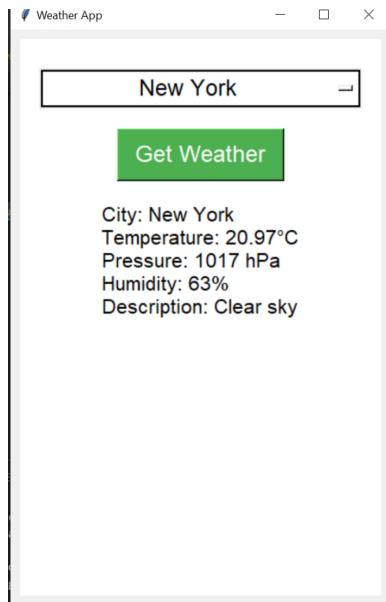
### Python

#### Code:

```
if __name__ == "__main__":
 root = ThemedTk(theme="arc")
 app = ToDoListManager(root)
 root.mainloop()
```

- **ThemedTk(theme="arc")**: Creates the main application window with a specific theme.
- **ToDoListManager(root)**: Initializes the To-Do List Manager with the main window.
- **root.mainloop()**: Starts the application's main event loop.

# Weather Forecasts



```
import requests
import tkinter as tk
from tkinter import messagebox

List of cities for the dropdown menu
cities = [
 "New York", "London", "Paris", "Tokyo", "Sydney", "Berlin",
 "Moscow",
 "Bangkok", "Buriram", "Chiang Mai", "Phuket", "Pattaya", "Krabi",
 "Hat Yai",
 "Nakhon Ratchasima", "Khon Kaen", "Udon Thani", "Nakhon Si
 Thammarat",
 "Surat Thani", "Samut Prakan", "Nonthaburi", "Ubon Ratchathani",
 "Ratchaburi", "Chachoengsao", "Rayong", "Samut Sakhon",
 "Chonburi"
]

def get_weather():
 city = city_var.get()
```

```
if not city:
 messagebox.showerror("Error", "Please select a city.")
 return

 api_key = "a1c2e587a3c663c62eeefac262a1343a" # Replace with
your working API key
 base_url = "http://api.openweathermap.org/data/2.5/weather?"
 complete_url = base_url + "appid=" + api_key + "&q=" + city +
"&units=metric"

try:
 response = requests.get(complete_url)
 response.raise_for_status() # Raises HTTPError for bad responses
 weather_data = response.json()

 main = weather_data["main"]
 temperature = main["temp"]
 pressure = main["pressure"]
 humidity = main["humidity"]
 weather = weather_data["weather"][0]
 description = weather["description"]

 weather_result = f"City: {weather_data['name']}\nTemperature:
{temperature}°C\nPressure: {pressure} hPa\nHumidity:
{humidity}%\nDescription: {description.capitalize()}"
 result_label.config(text=weather_result)
except requests.exceptions.HTTPError as http_err:
 messagebox.showerror("Error", f"HTTP error occurred:
{http_err}")
except requests.exceptions.RequestException as req_err:
 messagebox.showerror("Error", f"Request error occurred:
{req_err}")
```

```
except KeyError:
 messagebox.showerror(
 "Error", "City not found or API response format error.")
except Exception as err:
 messagebox.showerror("Error", f"An error occurred: {err}")
Create Tkinter window
root = tk.Tk()
root.title("Weather App")
root.geometry("400x600+400+-10")
root.configure(bg="#f0f0f0")

Create a frame for better layout
frame = tk.Frame(root, bg="#ffffff", padx=20, pady=20)
frame.pack(expand=True, fill=tk.BOTH, padx=10, pady=10)

Create dropdown menu for city selection
city_var = tk.StringVar(root)
city_var.set(cities[0]) # Set default value
city_menu = tk.OptionMenu(frame, city_var, *cities)
city_menu.config(font=("Helvetica", 18), bg="#ffffff", relief="solid")
city_menu.pack(pady=10, fill=tk.X)

Create button to get weather
get_weather_button = tk.Button(frame, text="Get Weather",
 command=get_weather, font=(
 "Helvetica", 18), bg="#4CAF50", fg="#ffffff", padx=10, pady=5,
 relief="raised")
get_weather_button.pack(pady=10)

Create label to display weather result
result_label = tk.Label(frame, text="", font=(
 "Helvetica", 16), bg="#ffffff", justify=tk.LEFT)
```

```
result_label.pack(pady=10)
```

```
Start the Tkinter event loop
root.mainloop()
```

## Detailed Description of the Weather App Code

The weather app code is a **Python** script that creates a graphical user interface (GUI) application using Tkinter, a built-in **Python** library for creating desktop applications. The app allows users to select a city from a dropdown menu and get the current weather information for that city by querying the OpenWeatherMap API. Below is a detailed breakdown of the code:

### 1. Importing Libraries

**Python**

**Code:**

```
import requests
import tkinter as tk
from tkinter import messagebox
```

- **requests:** A popular library used to make HTTP requests to web APIs. In this case, it is used to fetch weather data from the OpenWeatherMap API.

- **tkinter**: The standard **Python** library for creating graphical user interfaces.
- **messagebox**: A module from tkinter used to display error messages and other dialog boxes.

## 2. List of Cities

**Python  
Code:**

```
cities = [
 "New York", "London", "Paris", "Tokyo", "Sydney",
 "Berlin", "Moscow",
 "Bangkok", "Buriram", "Chiang Mai", "Phuket",
 "Pattaya", "Krabi", "Hat Yai",
 "Nakhon Ratchasima", "Khon Kaen", "Udon Thani",
 "Nakhon Si Thammarat",
 "Surat Thani", "Samut Prakan", "Nonthaburi",
 "Ubon Ratchathani",
 "Ratchaburi", "Chachoengsao", "Rayong", "Samut
 Sakhon", "Chonburi"]
```

This list contains the names of cities for which the weather information can be queried. It includes major cities from around the world and several cities from Thailand.

## 3. get\_weather Function

## Python

### Code:

```
def get_weather():
 city = city_var.get()
 if not city:
 messagebox.showerror("Error", "Please select a
city.")
 return

 api_key = "a1c2e587a3c663c62eeefac262a1343a" #
Replace with your working API key
 base_url =
"http://api.openweathermap.org/data/2.5/weather?"
 complete_url = base_url + "appid=" + api_key +
"&q=" + city + "&units=metric"

 try:
 response = requests.get(complete_url)
 response.raise_for_status() # Raises
HTTPError for bad responses
 weather_data = response.json()

 main = weather_data["main"]
 temperature = main["temp"]
 pressure = main["pressure"]
 humidity = main["humidity"]
 weather = weather_data["weather"][0]
```

```
description = weather["description"]

weather_result = f"City:
{weather_data['name']} \nTemperature: {temperature}
°C \nPressure: {pressure} hPa \nHumidity:
{humidity} % \nDescription: {description.capitalize()}"

result_label.config(text=weather_result)

except requests.exceptions.HTTPError as http_err:
 messagebox.showerror("Error", f"HTTP error occurred: {http_err}")

except requests.exceptions.RequestException as req_err:
 messagebox.showerror("Error", f"Request error occurred: {req_err}")

except KeyError:
 messagebox.showerror("Error", "City not found or API response format error.")

except Exception as err:
 messagebox.showerror("Error", f"An error occurred: {err}")

• city = city_var.get(): Retrieves the currently selected city from the dropdown menu.

• API Key: api_key is used to authenticate with the OpenWeatherMap API. Replace it with a valid API key obtained from OpenWeatherMap.
```

- **complete\_url**: Constructs the URL for the API request using the selected city and API key.
- **response = requests.get(complete\_url)**: Sends a GET request to the OpenWeatherMap API.
- **Exception Handling**: The code handles various exceptions such as HTTP errors, request errors, and key errors to provide appropriate error messages.
- **Weather Data Extraction**: Extracts and formats temperature, pressure, humidity, and weather description from the API response.

## 4. Creating the Tkinter Window

**Python**

**Code:**

```
root = tk.Tk()
root.title("Weather App")
root.geometry("400x600+400+-10")
root.configure(bg="#f0f0f0")
```

- **root = tk.Tk()**: Initializes the Tkinter application window.
- **root.title("Weather App")**: Sets the title of the window.
- **root.geometry("400x600+400+-10")**: Defines the size and position of the window.

- **root.configure(bg="#f0f0f0")**: Sets the background color of the window.

## 5. Creating and Configuring Widgets

### Python

#### Code:

```
frame = tk.Frame(root, bg="#ffffff", padx=20,
pady=20)
frame.pack(expand=True, fill=tk.BOTH, padx=10,
pady=10)

city_var = tk.StringVar(root)
city_var.set(cities[0]) # Set default value
city_menu = tk.OptionMenu(frame, city_var, *cities)
city_menu.config(font=("Helvetica", 18), bg="#ffffff",
relief="solid")
city_menu.pack(pady=10, fill=tk.X)

get_weather_button = tk.Button(frame, text="Get
Weather", command=get_weather, font=(
 "Helvetica", 18), bg="#4CAF50", fg="#ffffff",
padx=10, pady=5, relief="raised")
get_weather_button.pack(pady=10)

result_label = tk.Label(frame, text="", font=(
 "Helvetica", 16), bg="#ffffff", justify=tk.LEFT)
result_label.pack(pady=10)
```

- **frame**: A Frame widget is used to organize other widgets inside the main window.
- **city\_var**: A StringVar is used to hold the selected city from the dropdown menu.
- **city\_menu**: An OptionMenu widget allows users to select a city from the list.
- **get\_weather\_button**: A Button widget that triggers the get\_weather function when clicked.
- **result\_label**: A Label widget to display the weather information.

## 6. Running the Application

**Python**

**Code:**

```
root.mainloop()
```

- **root.mainloop()**: Starts the Tkinter event loop, making the application responsive to user interactions.

## How to Use the Program

1. **Run the Application**: Execute the **Python** script to open the weather app window.
2. **Select a City**: Use the dropdown menu to choose a city from the list.

3. **Get Weather:** Click the "Get Weather" button to fetch and display the current weather information for the selected city.
4. **View Results:** The weather details, including temperature, pressure, humidity, and description, will be displayed in the label below the button.



## How to Install and Run

### 1. Install Required Libraries:

- Ensure you have **Python** installed on your system.
- Install the requests library if it's not already installed. You can do this using **pip**:

**bash::**

**Code:**

**pip** install requests

### 2. Obtain an API Key:

- Sign up for an API key from OpenWeatherMap at OpenWeatherMap API.
- Replace the api\_key value in the script with your obtained API key.

### **3. Save the Script:**

- Save the provided **Python** script to a file,  
e.g., weather\_app.py.

### **4. Run the Script:**

- Open a terminal or command prompt.
- Navigate to the directory containing  
weather\_app.py.
- Run the script using **Python**

**bash::**

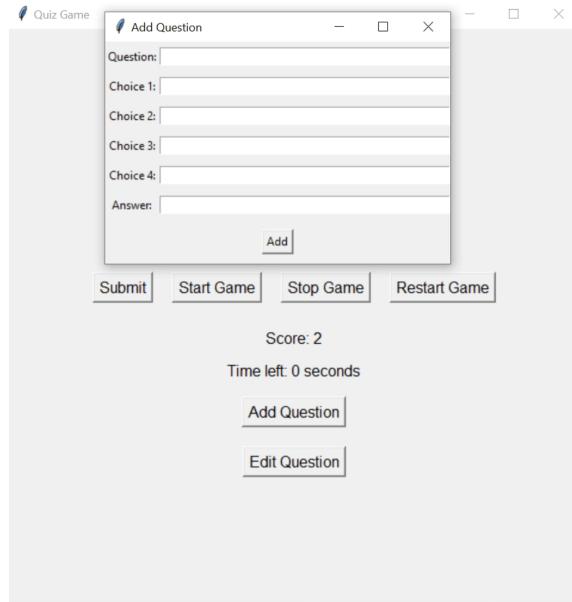
**Code:**

**Python** weather\_app.py

### **5. Use the Application:**

- The Tkinter window will appear.
- Follow the usage instructions to select a  
city and get weather information.

# Quiz Game Multiple Choices



```
import tkinter as tk
from tkinter import messagebox
import sqlite3
import random

def create_database():
 conn = sqlite3.connect('quiz.db')
 cursor = conn.cursor()

 cursor.execute("CREATE TABLE IF NOT EXISTS questions (
 id INTEGER PRIMARY KEY AUTOINCREMENT,
 question TEXT NOT NULL,
 answer TEXT NOT NULL
)")

 cursor.execute("CREATE TABLE IF NOT EXISTS choices (
 id INTEGER PRIMARY KEY AUTOINCREMENT,
 question_id INTEGER,
 choice TEXT NOT NULL
)")
```

```
choice TEXT NOT NULL,
FOREIGN KEY (question_id) REFERENCES
questions (id)
)")

conn.commit()
conn.close()

create_database()

class QuizGame:
 def __init__(self, root):
 self.root = root
 self.root.title("Quiz Game")
 self.root.geometry("600x600")

 self.conn = sqlite3.connect('quiz.db')
 self.cursor = self.conn.cursor()

 self.create_widgets()
 self.reset_game()

 def create_widgets(self):
 self.main_frame = tk.Frame(self.root, padx=10, pady=10)
 self.main_frame.pack(expand=True, fill=tk.BOTH)

 self.question_label = tk.Label(self.main_frame, text="",
wraplength=500, font=(
 "Helvetica", 14), justify=tk.LEFT)
 self.question_label.pack(pady=10)
```

```
self.choice_var = tk.StringVar()
self.radio_buttons_frame = tk.Frame(self.main_frame)
self.radio_buttons_frame.pack(pady=10)

self.radio_buttons = []
for _ in range(4):
 rb = tk.Radiobutton(self.radio_buttons_frame, text="",
 variable=self.choice_var, value="", font=
("Helvetica", 12))
 rb.pack(anchor="w", padx=20, pady=5)
 self.radio_buttons.append(rb)

self.button_frame = tk.Frame(self.main_frame)
self.button_frame.pack(pady=20)

self.submit_button = tk.Button(
 self.button_frame, text="Submit",
 command=self.check_answer, font=("Helvetica", 12))
self.submit_button.grid(row=0, column=0, padx=10)

self.start_game_button = tk.Button(
 self.button_frame, text="Start Game",
 command=self.start_game, font=("Helvetica", 12))
self.start_game_button.grid(row=0, column=1, padx=10)

self.stop_game_button = tk.Button(
 self.button_frame, text="Stop Game",
 command=self.end_game, font=("Helvetica", 12))
self.stop_game_button.grid(row=0, column=2, padx=10)

self.restart_game_button = tk.Button(
```

```
 self.button_frame, text="Restart Game",
command=self.start_game, font=("Helvetica", 12))
 self.restart_game_button.grid(row=0, column=3, padx=10)
 self.restart_game_button.grid_remove() # Hide the restart button
initially

 self.score_label = tk.Label(
 self.main_frame, text=f"Score: 0", font=("Helvetica", 12))
 self.score_label.pack(pady=5)

 self.timer_label = tk.Label(
 self.main_frame, text=f"Time left: 60 seconds", font=
("Helvetica", 12))
 self.timer_label.pack(pady=5)

 self.add_question_button = tk.Button(
 self.main_frame, text="Add Question",
command=self.open_add_question_window, font=("Helvetica", 12))
 self.add_question_button.pack(pady=10)

 self.edit_question_button = tk.Button(
 self.main_frame, text="Edit Question",
command=self.open_edit_question_window, font=("Helvetica", 12))
 self.edit_question_button.pack(pady=10)

def reset_game(self):
 self.score = 0
 self.question_index = 0
 self.time_left = 60
 self.game_over = False
 self.timer_running = False
```

```
 self.score_label.config(text=f"Score: {self.score}")
 self.timer_label.config(text=f"Time left: {self.time_left} seconds")
 self.choice_var.set(None)

def start_game(self):
 self.reset_game()
 self.questions = self.load_questions()
 self.display_question()
 self.restart_game_button.grid_remove()

def load_questions(self):
 self.cursor.execute('SELECT id, question, answer FROM
questions')
 questions = self.cursor.fetchall()

 question_list = []
 for question in questions:
 question_id, question_text, answer = question
 self.cursor.execute(
 'SELECT choice FROM choices WHERE question_id = ?',
 (question_id,))
 choices = [choice[0] for choice in self.cursor.fetchall()]
 question_list.append(
 {"id": question_id, "question": question_text, "choices": choices,
 "answer": answer})

 random.shuffle(question_list)
 return question_list

def display_question(self):
 if self.question_index < len(self.questions):
 question_data = self.questions[self.question_index]
```

```
 self.question_label.config(text=question_data["question"])

 for idx, choice in enumerate(question_data["choices"]):
 self.radio_buttons[idx].config(text=choice, value=choice)

 self.choice_var.set(None)

 self.time_left = 60
 self.update_timer()
else:
 self.end_game()

def update_timer(self):
 if self.time_left > 0 and not self.game_over:
 self.time_left -= 1
 self.timer_label.config(
 text=f"Time left: {self.time_left} seconds")
 self.root.after(1000, self.update_timer)
 else:
 self.check_answer(timeout=True)

def check_answer(self, timeout=False):
 if not timeout:
 selected_answer = self.choice_var.get()
 if selected_answer == self.questions[self.question_index]
 ["answer"]:
 self.score += 1

 self.score_label.config(text=f"Score: {self.score}")
 self.question_index += 1
 if self.question_index < len(self.questions):
```

```
 self.display_question()
 else:
 self.end_game()

def end_game(self):
 if not self.game_over:
 self.game_over = True
 self.restart_game_button.grid() # Show the restart button
 messagebox.showinfo(
 "Game Over", f"Quiz over! Your final score is:
{self.score}")

def open_add_question_window(self):
 self.add_question_window = tk.Toplevel(self.root)
 self.add_question_window.title("Add Question")

 tk.Label(self.add_question_window, text="Question:").grid(
 row=0, column=0, pady=5)
 self.question_entry = tk.Entry(self.add_question_window,
width=50)
 self.question_entry.grid(row=0, column=1, pady=5)

 tk.Label(self.add_question_window, text="Choice 1:").grid(
 row=1, column=0, pady=5)
 self.choice1_entry = tk.Entry(self.add_question_window,
width=50)
 self.choice1_entry.grid(row=1, column=1, pady=5)

 tk.Label(self.add_question_window, text="Choice 2:").grid(
 row=2, column=0, pady=5)
 self.choice2_entry = tk.Entry(self.add_question_window,
width=50)
```

```
 self.choice2_entry.grid(row=2, column=1, pady=5)

 tk.Label(self.add_question_window, text="Choice 3:").grid(
 row=3, column=0, pady=5)
 self.choice3_entry = tk.Entry(self.add_question_window,
width=50)
 self.choice3_entry.grid(row=3, column=1, pady=5)

 tk.Label(self.add_question_window, text="Choice 4:").grid(
 row=4, column=0, pady=5)
 self.choice4_entry = tk.Entry(self.add_question_window,
width=50)
 self.choice4_entry.grid(row=4, column=1, pady=5)

 tk.Label(self.add_question_window, text="Answer:").grid(
 row=5, column=0, pady=5)
 self.answer_entry = tk.Entry(self.add_question_window,
width=50)
 self.answer_entry.grid(row=5, column=1, pady=5)

 tk.Button(self.add_question_window, text="Add",
command=self.add_question_to_db).grid(
 row=6, column=0, columnspan=2, pady=10)

def add_question_to_db(self):
 question = self.question_entry.get()
 choices = [self.choice1_entry.get(), self.choice2_entry.get(),
), self.choice3_entry.get(), self.choice4_entry.get()]
 answer = self.answer_entry.get()

 self.cursor.execute(
```

```
'INSERT INTO questions (question, answer) VALUES (?, ?)',
(question, answer))
 question_id = self.cursor.lastrowid

 for choice in choices:
 self.cursor.execute(
 'INSERT INTO choices (question_id, choice) VALUES (?, ?, ?)',
 (question_id, choice))

 self.conn.commit()

 messagebox.showinfo("Success", "Question added successfully!")
 self.add_question_window.destroy()
 self.questions = self.load_questions()

def open_edit_question_window(self):
 self.edit_question_window = tk.Toplevel(self.root)
 self.edit_question_window.title("Edit Question")

 tk.Label(self.edit_question_window, text="Select Question to
Edit:").grid(
 row=0, column=0, pady=5)

 self.question_listbox = tk.Listbox(self.edit_question_window,
 width=50)
 self.question_listbox.grid(row=1, column=0, columnspan=2,
 pady=5)

 self.load_question_list()
 self.question_listbox.bind(
 '<<ListboxSelect>>', self.on_question_select)
```

```
tk.Button(self.edit_question_window, text="Update",
 command=self.update_question_in_db).grid(row=2,
column=0, pady=10)
tk.Button(self.edit_question_window, text="Delete",
 command=self.delete_question_from_db).grid(row=2,
column=1, pady=10)

def load_question_list(self):
 self.cursor.execute('SELECT id, question FROM questions')
 questions = self.cursor.fetchall()

 self.question_listbox.delete(0, tk.END)
 for question in questions:
 self.question_listbox.insert(
 tk.END, f'{question[0]}: {question[1]}')

def on_question_select(self, event):
 selection = event.widget.curselection()
 if selection:
 index = selection[0]
 self.selected_question_id = int(
 event.widget.get(index).split(':')[0])
 self.load_selected_question()

def load_selected_question(self):
 self.cursor.execute(
 'SELECT question, answer FROM questions WHERE id = ?',
 (self.selected_question_id,))
 question, answer = self.cursor.fetchone()

 self.cursor.execute(
```

```
'SELECT choice FROM choices WHERE question_id = ?',
(self.selected_question_id,))
choices = [choice[0] for choice in self.cursor.fetchall()]

tk.Label(self.edit_question_window, text="Question:").grid(
 row=3, column=0, pady=5)
self.edit_question_entry = tk.Entry(
 self.edit_question_window, width=50)
self.edit_question_entry.grid(row=3, column=1, pady=5)
self.edit_question_entry.insert(0, question)

tk.Label(self.edit_question_window, text="Choice 1:").grid(
 row=4, column=0, pady=5)
self.edit_choice1_entry = tk.Entry(self.edit_question_window,
width=50)
self.edit_choice1_entry.grid(row=4, column=1, pady=5)
self.edit_choice1_entry.insert(0, choices[0])

tk.Label(self.edit_question_window, text="Choice 2:").grid(
 row=5, column=0, pady=5)
self.edit_choice2_entry = tk.Entry(self.edit_question_window,
width=50)
self.edit_choice2_entry.grid(row=5, column=1, pady=5)
self.edit_choice2_entry.insert(0, choices[1])

tk.Label(self.edit_question_window, text="Choice 3:").grid(
 row=6, column=0, pady=5)
self.edit_choice3_entry = tk.Entry(self.edit_question_window,
width=50)
self.edit_choice3_entry.grid(row=6, column=1, pady=5)
self.edit_choice3_entry.insert(0, choices[2])
```

```
tk.Label(self.edit_question_window, text="Choice 4:").grid(
 row=7, column=0, pady=5)
 self.edit_choice4_entry = tk.Entry(self.edit_question_window,
width=50)
 self.edit_choice4_entry.grid(row=7, column=1, pady=5)
 self.edit_choice4_entry.insert(0, choices[3])

tk.Label(self.edit_question_window, text="Answer:").grid(
 row=8, column=0, pady=5)
 self.edit_answer_entry = tk.Entry(self.edit_question_window,
width=50)
 self.edit_answer_entry.grid(row=8, column=1, pady=5)
 self.edit_answer_entry.insert(0, answer)

def update_question_in_db(self):
 question = self.edit_question_entry.get()
 choices = [self.edit_choice1_entry.get(),
self.edit_choice2_entry.get()
, self.edit_choice3_entry.get(), self.edit_choice4_entry.get()]
 answer = self.edit_answer_entry.get()

 self.cursor.execute('UPDATE questions SET question = ?, answer
= ? WHERE id = ?',
 (question, answer, self.selected_question_id))
 self.cursor.execute(
 'DELETE FROM choices WHERE question_id = ?',
(self.selected_question_id,))

for choice in choices:
 self.cursor.execute(
 'INSERT INTO choices (question_id, choice) VALUES (?,
?)', (self.selected_question_id, choice))
```

```
 self.conn.commit()

 messagebox.showinfo("Success", "Question updated
successfully!")
 self.edit_question_window.destroy()
 self.questions = self.load_questions()

def delete_question_from_db(self):
 self.cursor.execute(
 'DELETE FROM questions WHERE id = ?',
 (self.selected_question_id,))
 self.cursor.execute(
 'DELETE FROM choices WHERE question_id = ?',
 (self.selected_question_id,))
 self.conn.commit()

 messagebox.showinfo("Success", "Question deleted
successfully!")
 self.edit_question_window.destroy()
 self.questions = self.load_questions()

if __name__ == "__main__":
 root = tk.Tk()
 app = QuizGame(root)
 root.mainloop()
```

# **Detailed Description of the Quiz Game Program Overview**

The provided program is a **Python**-based quiz game built using the tkinter library for the graphical user interface (GUI) and sqlite3 for database management. This application allows users to create, edit, and delete quiz questions and answers, and also play a quiz game where they can answer questions and see their score.

## **Code Breakdown**

### **Imports and Database Initialization**

#### **1. Imports:**

**Python**

**Code:**

```
import tkinter as tk
from tkinter import messagebox
import sqlite3
import random
```

- tkinter: Used for creating the GUI.
- messagebox: Used to display pop-up messages to the user.
- sqlite3: Used for database operations.
- random: Used to shuffle questions.

#### **2. Database Creation Function:**

## Python

### Code:

```
def create_database():
 conn = sqlite3.connect('quiz.db')
 cursor = conn.cursor()

 cursor.execute("CREATE TABLE IF NOT
EXISTS questions (
 id INTEGER PRIMARY KEY
AUTOINCREMENT,
 question TEXT NOT NULL,
 answer TEXT NOT NULL
)")

 cursor.execute("CREATE TABLE IF NOT
EXISTS choices (
 id INTEGER PRIMARY KEY
AUTOINCREMENT,
 question_id INTEGER,
 choice TEXT NOT NULL,
 FOREIGN KEY (question_id)
REFERENCES questions (id)
)")

 conn.commit()
 conn.close()
```

- This function creates a SQLite database with two tables:
  - **questions**: Stores the question ID, question text, and correct answer.
  - **choices**: Stores possible answers for each question, linked by `question_id`.

## QuizGame Class

The QuizGame class handles the entire functionality of the quiz game, including GUI components, game logic, and database operations.

### 1. Initialization:

#### Python

#### Code:

```
class QuizGame:
 def __init__(self, root):
 self.root = root
 self.root.title("Quiz Game")
 self.root.geometry("600x600")

 self.conn = sqlite3.connect('quiz.db')
 self.cursor = self.conn.cursor()
```

```
 self.create_widgets()
 self.reset_game()
```

- Initializes the main window and database connection.

- Calls methods to create GUI widgets and reset game variables.

## 2. Creating Widgets:

**Python**

**Code:**

```
def create_widgets(self):
 # Main frame
 self.main_frame = tk.Frame(self.root, padx=10,
pady=10)
 self.main_frame.pack(expand=True,
fill=tk.BOTH)

 # Question label
 self.question_label = tk.Label(self.main_frame,
text="",
wraplength=500, font=("Helvetica", 14),
justify=tk.LEFT)
 self.question_label.pack(pady=10)

 # Radio buttons for choices
 self.choice_var = tk.StringVar()
 self.radio_buttons_frame =
tk.Frame(self.main_frame)
 self.radio_buttons_frame.pack(pady=10)

 self.radio_buttons = []
 for _ in range(4):
```

```
 rb = tk.Radiobutton(self.radio_buttons_frame,
text="\"", variable=self.choice_var, value="\"", font=
("Helvetica", 12))
 rb.pack(anchor="w", padx=20, pady=5)
 self.radio_buttons.append(rb)

Buttons for game control
self.button_frame = tk.Frame(self.main_frame)
self.button_frame.pack(pady=20)

self.submit_button =
tk.Button(self.button_frame, text="Submit",
command=self.check_answer, font=("Helvetica",
12))
 self.submit_button.grid(row=0, column=0,
padx=10)

self.start_game_button =
tk.Button(self.button_frame, text="Start Game",
command=self.start_game, font=("Helvetica", 12))
 self.start_game_button.grid(row=0, column=1,
padx=10)

self.stop_game_button =
tk.Button(self.button_frame, text="Stop Game",
command=self.end_game, font=("Helvetica", 12))
 self.stop_game_button.grid(row=0, column=2,
padx=10)
```

```
 self.restart_game_button =
 tk.Button(self.button_frame, text="Restart Game",
 command=self.start_game, font=("Helvetica", 12))
 self.restart_game_button.grid(row=0, column=3,
 padx=10)
 self.restart_game_button.grid_remove() # Hide
 the restart button initially

 # Score and timer labels
 self.score_label = tk.Label(self.main_frame,
 text=f"Score: 0", font=("Helvetica", 12))
 self.score_label.pack(pady=5)

 self.timer_label = tk.Label(self.main_frame,
 text=f"Time left: 60 seconds", font=("Helvetica",
 12))
 self.timer_label.pack(pady=5)

 # Buttons for managing questions
 self.add_question_button =
 tk.Button(self.main_frame, text="Add Question",
 command=self.open_add_question_window, font=
 ("Helvetica", 12))
 self.add_question_button.pack(pady=10)

 self.edit_question_button =
 tk.Button(self.main_frame, text="Edit Question",
```

```
command=self.open_edit_question_window, font= ("Helvetica", 12))
```

```
self.edit_question_button.pack(pady=10)
```

- Creates and configures the main components of the GUI, including labels, buttons, and radio buttons for answering questions.

### 3. Game Logic:

- **Reset Game:**

**Python**

**Code:**

```
def reset_game(self):
 self.score = 0
 self.question_index = 0
 self.time_left = 60
 self.game_over = False
 self.timer_running = False

 self.score_label.config(text=f"Score:
{self.score}")
 self.timer_label.config(text=f"Time left:
{self.time_left} seconds")
 self.choice_var.set(None)
 • Resets game variables and updates the
 UI.
```

- **Start Game:**

**Python**

**Code:**

```
def start_game(self):
 self.reset_game()
 self.questions = self.load_questions()
 self.display_question()
```

- self.restart\_game\_button.grid\_remove()
  - Initializes and starts the game, loading questions and displaying the first one.

- **Load Questions:**

**Python**

**Code:**

```
def load_questions(self):
 self.cursor.execute('SELECT id, question,
answer FROM questions')
 questions = self.cursor.fetchall()
 question_list = []
 for question in questions:
 question_id, question_text, answer =
question
 self.cursor.execute('SELECT choice
FROM choices WHERE question_id = ?',
(question_id,))
 choices = [choice[0] for choice in
self.cursor.fetchall()]
 question_list.append({'id': question_id,
"question": question_text, "choices": choices,
"answer": answer})
 random.shuffle(question_list)
 return question_list
```

- Retrieves questions and their choices from the database and shuffles them.

- **Display Question:**

**Python**

**Code:**

```
def display_question(self):
 if self.question_index < len(self.questions):
 question_data =
 self.questions[self.question_index]
 self.question_label.config(text=question_da
ta["question"])

 for idx, choice in
enumerate(question_data["choices"]):
 self.radio_buttons[idx].config(text=ch
oice, value=choice)

 self.choice_var.set(None)

 self.time_left = 60
 self.update_timer()
 else:
 self.end_game()
 ▪ Displays the current question and its
 choices. Starts the timer.
```

- **Update Timer:**

### Python

#### Code:

```
def update_timer(self):
 if self.time_left > 0 and not self.game_over:
 self.time_left -= 1
 self.timer_label.config(text=f"Time left:
{self.time_left} seconds")
 self.root.after(1000, self.update_timer)
 else:
 self.check_answer(timeout=True)

- Updates the timer every second and checks the answer when time runs out.

```

- **Check Answer:**

### Python

#### Code:

```
def check_answer(self, timeout=False):
 if not timeout:
 selected_answer = self.choice_var.get()
 if selected_answer ==
self.questions[self.question_index]["answer"]:
 self.score += 1

 self.score_label.config(text=f"Score:
{self.score}")
```

```
self.question_index += 1
if self.question_index < len(self.questions):
 self.display_question()
else:
 self.end_game()
 • Checks if the selected answer is correct
 and updates the score. Moves to the next
 question or ends the game.
```

- **End Game:**

**Python**

**Code:**

```
def end_game(self):
 if not self.game_over:
 self.game_over = True
 self.restart_game_button.grid() # Show
 the restart button
 messagebox.showinfo("Game Over",
f"Quiz over! Your final score is: {self.score}")
 • Ends the game, displays the final score,
 and shows the restart button.
```

## 4. Manage Questions:

- **Add Question:**

**Python**

**Code:**

```
def open_add_question_window(self):
 self.add_question_window =
tk.Toplevel(self.root)
 self.add_question_window.title("Add
Question")
 tk.Label(self.add_question_window,
text="Question:").grid(row=0, column=0,
pady=5)
```

```
 self.question_entry =
 tk.Entry(self.add_question_window, width=50)
 self.question_entry.grid(row=0, column=1,
 pady=5)
```

# Similarly add labels and entries for  
choices and answer

```
tk.Button(self.add_question_window,
text="Add",
command=self.add_question_to_db).grid(row=6, column=0, columnspan=2, pady=10)
```

- Opens a window to add a new question and its choices to the database.

- **Add Question to Database:**

**Python**

**Code:**

```
def add_question_to_db(self):
 question_text = self.question_entry.get()
 # Get choices and answer from entries

 self.cursor.execute('INSERT INTO questions
(question, answer) VALUES (?, ?)',
(question_text, correct_answer))
 question_id = self.cursor.lastrowid

 for choice in choices:
```

```
 self.cursor.execute('INSERT INTO choices
 (question_id, choice) VALUES (?, ?)',
 (question_id, choice))

 self.conn.commit()
 self.add_question_window.destroy()
 ▪ Inserts the new question and choices into
 the database.
```

- **Edit Question:**

### Python

#### Code:

```
def open_edit_question_window(self):
 self.edit_question_window =
 tk.Toplevel(self.root)
 self.edit_question_window.title("Edit
Question")

 # Add widgets to edit existing questions
```

```
 tk.Button(self.edit_question_window,
 text="Save",
 command=self.save_edits).grid(row=6,
 column=0, columnspan=2, pady=10)
```

- Opens a window to edit an existing question and its choices.

- **Save Edits:**

### **Python**

#### **Code:**

```
def save_edits(self):
```

```
 # Retrieve edited question and choices
```

```
 # Update the database with new values
```

```
 self.conn.commit()
```

```
 self.edit_question_window.destroy()
```

- Saves changes made to a question in the database.

## **Conclusion**

The quiz game application offers a user-friendly interface for playing a quiz game and managing quiz questions and answers. The code demonstrates how to integrate tkinter for GUI and sqlite3 for database management, providing a complete solution for a simple quiz game with question management features.

# Pomodoro Timer



```
import tkinter as tk
from tkinter import messagebox
from datetime import timedelta
from playsound import playsound

class PomodoroTimer:
 def __init__(self, root):
 self.root = root
 self.root.title("Advanced Pomodoro Timer")
 self.root.geometry("400x300")
 self.root.config(bg="#f5f5f5")

 self.work_duration = 25 * 60 # 25 minutes in seconds
 self.short_break_duration = 5 * 60 # 5 minutes in seconds
 self.long_break_duration = 15 * 60 # 15 minutes in seconds
 self.cycles = 0

 self.timer_running = False
 self.current_time = self.work_duration

 self.label = tk.Label(root, text="Pomodoro Timer",
 font=("Helvetica", 24), bg="#f5f5f5")
 self.label.pack(pady=20)
```

```
self.time_label = tk.Label(root, text=self.format_time(
 self.current_time), font=("Helvetica", 48), bg="#f5f5f5")
self.time_label.pack(pady=20)

self.start_button = tk.Button(root, text="Start",
command=self.start_timer, font=(
 "Helvetica", 12), bg="#4CAF50", fg="white", width=10)
self.start_button.pack(side="left", padx=20)

self.reset_button = tk.Button(root, text="Reset",
command=self.reset_timer, font=(
 "Helvetica", 12), bg="#f44336", fg="white", width=10)
self.reset_button.pack(side="right", padx=20)

self.update_clock()

def format_time(self, seconds):
 return str(timedelta(seconds=seconds))

def update_clock(self):
 if self.timer_running:
 if self.current_time > 0:
 self.current_time -= 1
 self.time_label.config(
 text=self.format_time(self.current_time))
 else:
 self.timer_running = False
 self.cycles += 1
 self.play_sound()
 if self.cycles % 4 == 0:
 self.current_time = self.long_break_duration
```

```
 self.label.config(text="Long Break", fg="#ff9800")
 elif self.cycles % 2 == 0:
 self.current_time = self.work_duration
 self.label.config(text="Work Time", fg="#4CAF50")
 else:
 self.current_time = self.short_break_duration
 self.label.config(text="Short Break", fg="#03A9F4")
 self.timer_running = True
 self.root.after(1000, self.update_clock)

def start_timer(self):
 if not self.timer_running:
 self.timer_running = True

def reset_timer(self):
 if messagebox.askokcancel("Reset Timer", "Are you sure you want to reset the timer?"):
 self.timer_running = False
 self.current_time = self.work_duration
 self.label.config(text="Pomodoro Timer", fg="black")
 self.time_label.config(text=self.format_time(self.current_time))

def play_sound(self):
 playsound("alarm_sound.mp3") # Replace with your alarm sound file

if __name__ == "__main__":
 root = tk.Tk()
 pomodoro_timer = PomodoroTimer(root)
 root.mainloop()
```

Here's a detailed guide on how to use the Pomodoro Timer, install the necessary components, and an in-depth explanation of the code.

## Advanced Pomodoro Timer

### Overview

The Advanced Pomodoro Timer is a simple and effective time management tool based on the Pomodoro Technique, which encourages users to work in focused intervals (usually 25 minutes), followed by short breaks (5 minutes), and longer breaks after completing several work intervals. This timer is built using [Python](#)'s Tkinter library for the GUI and plays an alarm sound to signal the end of each interval.

## Features

- Work for 25 minutes
- Short break of 5 minutes
- Long break of 15 minutes after every 4 cycles
- Visual countdown and sound notifications
- Start and reset functionality

## How to Use

1. **Start the Timer:** Click the "Start" button to begin the timer. The timer will count down from 25 minutes.
2. **Receive Notifications:** When the timer reaches zero, an alarm sound will play to notify you. Depending on the cycle count, the timer will either start a short break (5 minutes) or a long break (15 minutes).
3. **Reset the Timer:** If you want to stop and reset the timer, click the "Reset" button. A confirmation dialog will appear, asking if you're sure you want to reset.

## How to Install

To run this Pomodoro Timer, you need to have [Python](#) installed on your machine. You can use Visual Studio Code (VSCode) as your IDE. Follow the steps below to install the necessary components:

## 1. Install Python

- Download the latest version of **Python** from the official website: [Python.org](https://www.python.org).
- Make sure to check the box that says "Add **Python** to PATH" during installation.

## 2. Install Visual Studio Code (VSCode)

- Download VSCode from the official website: [code.visualstudio.com](https://code.visualstudio.com).
- Follow the installation instructions for your operating system.

## 3. Set Up a Virtual Environment (Optional)

Creating a virtual environment is recommended for managing project dependencies. Open VSCode and follow these steps:

- Open the terminal in VSCode (`Ctrl + ``).
- Run the following command to create a virtual environment:

**bash::**

**Code:**

**Python** -m venv venv

- Activate the virtual environment:
  - On Windows:

**bash::**

**Code:**

```
venv\Scripts\activate
 ○ On macOS/Linux:
bash::
Code:
source venv/bin/activate
```

## 4. Install Required Libraries

You will need the playsound library for playing sounds. Install it using **pip**:

```
bash::
Code:
pip install playsound
```

## 5. Download an Alarm Sound

Download an MP3 file for the alarm sound and save it in the same directory as your script. Rename the file to alarm\_sound.mp3 or change the filename in the code accordingly.

### Running the Timer

1. Open the script in VSCode.
2. Run the script by clicking the green play button in the upper right corner or by running the command in the terminal:

```
bash::
Code:
```

**Python** your\_script\_name.py

3. The Pomodoro Timer window should appear, and you can start using it.

## Code Explanation

Here's a detailed breakdown of the code:

### 1. Importing Libraries

**Python**

**Code:**

```
import tkinter as tk
from tkinter import messagebox
from datetime import timedelta
from playsound import playsound
```

- tkinter: A standard GUI library in **Python** used for creating windowed applications.
- messagebox: A module within tkinter to show dialog boxes.
- timedelta: A module for manipulating time intervals.
- playsound: A library for playing sound files.

### 2. PomodoroTimer Class

**Python**

**Code:**

```
class PomodoroTimer:
```

```
 def __init__(self, root):
```

This is the main class that manages the Pomodoro Timer. The `__init__` method initializes the timer settings and creates the GUI components.

### 3. Timer Settings

#### Python

##### Code:

```
self.work_duration = 25 * 60 # 25 minutes in seconds
self.short_break_duration = 5 * 60 # 5 minutes in
seconds
self.long_break_duration = 15 * 60 # 15 minutes in
seconds
self.cycles = 0
```

- Here, the timer durations are set in seconds.

### 4. Timer Control Variables

#### Python

##### Code:

```
self.timer_running = False
self.current_time = self.work_duration
```

- `timer_running`: A flag to indicate whether the timer is active.
- `current_time`: Tracks the remaining time for the current session.

### 5. Creating GUI Components

The GUI is built using labels and buttons:

## Python

### Code:

```
self.label = tk.Label(root, text="Pomodoro Timer",
font=("Helvetica", 24), bg="#f5f5f5")
self.time_label = tk.Label(root,
text=self.format_time(self.current_time), font=
("Helvetica", 48), bg="#f5f5f5")
self.start_button = tk.Button(root, text="Start",
command=self.start_timer, font=("Helvetica", 12),
bg="#4CAF50", fg="white", width=10)
self.reset_button = tk.Button(root, text="Reset",
command=self.reset_timer, font=("Helvetica", 12),
bg="#f44336", fg="white", width=10)
```

- Labels show the timer title and the current time.
- Buttons allow starting and resetting the timer.

## 6. Time Formatting

**Python**

**Code:**

```
def format_time(self, seconds):
 return str(timedelta(seconds=seconds))
```

This method converts seconds into a readable format (HH:MM ).

## 7. Timer Update Logic

**Python**

**Code:**

```
def update_clock(self):
```

This method updates the timer every second. It decrements the time and checks for timer expiration to trigger breaks or the next work session.

## 8. Starting the Timer

**Python**

**Code:**

```
def start_timer(self):
 if not self.timer_running:
 self.timer_running = True
```

This method starts the timer when the "Start" button is clicked.

## 9. Resetting the Timer

**Python**

**Code:**

```
def reset_timer(self):
 if messagebox.askokcancel("Reset Timer", "Are you
sure you want to reset the timer?"):
 self.timer_running = False
 self.current_time = self.work_duration
 self.label.config(text="Pomodoro Timer",
fg="black")
 self.time_label.config(text=self.format_time(self
.current_time))
```

This method resets the timer and shows a confirmation dialog when the "Reset" button is clicked.

## 10. Playing Sound

**Python**

**Code:**

```
def play_sound(self):
 playsound("alarm_sound.mp3") # Replace with
your alarm sound file
```

This method plays an alarm sound when the timer reaches zero.

## 11. Main Execution

**Python**

**Code:**

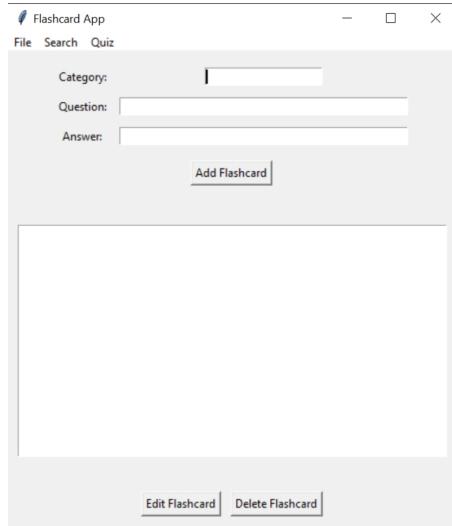
```
if __name__ == "__main__":
 root = tk.Tk()
 pomodoro_timer = PomodoroTimer(root)
 root.mainloop()
```

This block initializes the Tkinter main window and starts the application's event loop.

## Conclusion

The Advanced Pomodoro Timer is an excellent tool for enhancing productivity using the Pomodoro Technique. With easy installation and a user-friendly interface, it allows you to manage your work sessions effectively. Feel free to customize the timer settings or the alarm sound to suit your preferences. Happy timing!

# FlashCard



```
import tkinter as tk
from tkinter import messagebox, simpledialog
import sqlite3

Create database connection
conn = sqlite3.connect('flashcards.db')
c = conn.cursor()

Function to add category column if it doesn't exist
```

```
def update_schema():
 try:
 c.execute("ALTER TABLE flashcards ADD COLUMN category
TEXT")
 conn.commit()
 except sqlite3.OperationalError:
 pass # Column already exists
```

```
Update the schema to include the category column
update_schema()
```

```
Create flashcards table if it doesn't exist
```

```
c.execute("CREATE TABLE IF NOT EXISTS flashcards (
 id INTEGER PRIMARY KEY,
 category TEXT,
 question TEXT,
 answer TEXT)")
conn.commit()
```

```
class FlashcardApp:
 def __init__(self, root):
 self.root = root
 self.root.title("Flashcard App")
 self.create_widgets()
 self.load_flashcards()

 # Initialize quiz attributes
 self.quiz_index = 0
 self.quiz_flashcards = []
 self.correct_answers = 0
 self.total_questions = 0

 def create_widgets(self):
 # Menu
 self.menu = tk.Menu(self.root)
 self.root.config(menu=self.menu)

 self.file_menu = tk.Menu(self.menu, tearoff=0)
 self.menu.add_cascade(label="File", menu=self.file_menu)
 self.file_menu.add_command(
 label="Add Flashcard", command=self.add_flashcard)
 self.file_menu.add_command(
 label="Review Flashcards", command=self.review_flashcards)
 self.file_menu.add_separator()
 self.file_menu.add_command(label="Exit",
 command=self.root.quit)

 self.search_menu = tk.Menu(self.menu, tearoff=0)
```

```
self.menu.add_cascade(label="Search", menu=self.search_menu)
self.search_menu.add_command(
 label="Search Flashcards", command=self.search_flashcards)

self.quiz_menu = tk.Menu(self.menu, tearoff=0)
self.menu.add_cascade(label="Quiz", menu=self.quiz_menu)
self.quiz_menu.add_command(label="Start Quiz",
command=self.start_quiz)

Frames
self.top_frame = tk.Frame(self.root)
self.top_frame.pack(pady=10)

self.middle_frame = tk.Frame(self.root)
self.middle_frame.pack(pady=10)

self.bottom_frame = tk.Frame(self.root)
self.bottom_frame.pack(pady=10)

Top Frame Widgets
self.category_label = tk.Label(self.top_frame, text="Category:")
self.category_label.grid(row=0, column=0, padx=5, pady=5)
self.category_entry = tk.Entry(self.top_frame, width=20)
self.category_entry.grid(row=0, column=1, padx=5, pady=5)

self.question_label = tk.Label(self.top_frame, text="Question:")
self.question_label.grid(row=1, column=0, padx=5, pady=5)
self.question_entry = tk.Entry(self.top_frame, width=50)
self.question_entry.grid(row=1, column=1, padx=5, pady=5)

self.answer_label = tk.Label(self.top_frame, text="Answer:")
self.answer_label.grid(row=2, column=0, padx=5, pady=5)
self.answer_entry = tk.Entry(self.top_frame, width=50)
self.answer_entry.grid(row=2, column=1, padx=5, pady=5)

self.add_button = tk.Button(
```

```
 self.top_frame, text="Add Flashcard",
command=self.add_flashcard)
 self.add_button.grid(row=3, column=0, columnspan=2, pady=10)

 # Middle Frame Widgets
 self.flashcard_listbox = tk.Listbox(
 self.middle_frame, width=75, height=15)
 self.flashcard_listbox.grid(row=0, column=0, padx=10, pady=10)
 self.flashcard_listbox.bind(
 "<<ListboxSelect>>", self.load_selected_flashcard)

 # Bottom Frame Widgets
 self.edit_button = tk.Button(
 self.bottom_frame, text="Edit Flashcard",
command=self.edit_flashcard)
 self.edit_button.grid(row=0, column=0, padx=5, pady=5)

 self.delete_button = tk.Button(
 self.bottom_frame, text="Delete Flashcard",
command=self.delete_flashcard)
 self.delete_button.grid(row=0, column=1, padx=5, pady=5)

def add_flashcard(self):
 category = self.category_entry.get()
 question = self.question_entry.get()
 answer = self.answer_entry.get()

 if category and question and answer:
 c.execute("INSERT INTO flashcards (category, question,
answer) VALUES (?, ?, ?)",
 (category, question, answer))
 conn.commit()
 self.clear_entries()
 self.load_flashcards()
 else:
 messagebox.showwarning(
```

```
"Input Error", "Please provide a category, question, and
answer.")
```

```
def load_flashcards(self):
 self.flashcard_listbox.delete(0, tk.END)
 c.execute("SELECT id, category, question FROM flashcards")
 flashcards = c.fetchall()
 for flashcard in flashcards:
 self.flashcard_listbox.insert(
 tk.END, f"{flashcard[0]}. {flashcard[1]}: {flashcard[2]}")
```

```
def load_selected_flashcard(self, event):
 selected = self.flashcard_listbox.curselection()
 if selected:
 flashcard_id = self.flashcard_listbox.get(selected).split(".")[0]
 c.execute(
 "SELECT category, question, answer FROM flashcards
WHERE id = ?", (flashcard_id,))
 flashcard = c.fetchone()
 if flashcard:
 self.category_entry.delete(0, tk.END)
 self.category_entry.insert(tk.END, flashcard[0])
 self.question_entry.delete(0, tk.END)
 self.question_entry.insert(tk.END, flashcard[1])
 self.answer_entry.delete(0, tk.END)
 self.answer_entry.insert(tk.END, flashcard[2])
```

```
def edit_flashcard(self):
 selected = self.flashcard_listbox.curselection()
 if selected:
 flashcard_id = self.flashcard_listbox.get(selected).split(".")[0]
 category = self.category_entry.get()
 question = self.question_entry.get()
 answer = self.answer_entry.get()

 if category and question and answer:
```

```
 c.execute("UPDATE flashcards SET category = ?, question = ?, answer = ? WHERE id = ?",
 (category, question, answer, flashcard_id))
 conn.commit()
 self.clear_entries()
 self.load_flashcards()
 else:
 messagebox.showwarning(
 "Input Error", "Please provide a category, question, and answer.")
 else:
 messagebox.showwarning(
 "Selection Error", "Please select a flashcard to edit.")

def delete_flashcard(self):
 selected = self.flashcard_listbox.curselection()
 if selected:
 flashcard_id = self.flashcard_listbox.get(selected).split(".")[0]
 c.execute("DELETE FROM flashcards WHERE id = ?",
 (flashcard_id,))
 conn.commit()
 self.clear_entries()
 self.load_flashcards()
 else:
 messagebox.showwarning(
 "Selection Error", "Please select a flashcard to delete.")

def clear_entries(self):
 self.category_entry.delete(0, tk.END)
 self.question_entry.delete(0, tk.END)
 self.answer_entry.delete(0, tk.END)

def review_flashcards(self):
 c.execute("SELECT category, question, answer FROM
 flashcards")
 flashcards = c.fetchall()
 if flashcards:
```

```
for flashcard in flashcards:
 category = flashcard[0]
 question = flashcard[1]
 answer = flashcard[2]
 messagebox.showinfo(
 "Review", f"Category: {category}\nQuestion:
{question}\nAnswer: {answer}")
else:
 messagebox.showinfo(
 "No Flashcards", "No flashcards available to review.")

def search_flashcards(self):
 search_term = simpledialog.askstring("Search", "Enter search
term:")
 if search_term:
 self.flashcard_listbox.delete(0, tk.END)
 c.execute("SELECT id, category, question FROM flashcards
WHERE category LIKE ? OR question LIKE ? OR answer LIKE ?",
 ('%' + search_term + '%', '%' + search_term + '%', '%' +
search_term + '%'))
 flashcards = c.fetchall()
 for flashcard in flashcards:
 self.flashcard_listbox.insert(
 tk.END, f"{flashcard[0]}. {flashcard[1]}:
{flashcard[2]}")
 else:
 self.load_flashcards()

def start_quiz(self):
 c.execute("SELECT id, question, answer FROM flashcards
ORDER BY RANDOM()")
 self.quiz_flashcards = c.fetchall()
 if self.quiz_flashcards:
 self.quiz_index = 0
 self.correct_answers = 0
 self.total_questions = len(self.quiz_flashcards)
 self.ask_question()
```

```

else:
 messagebox.showinfo(
 "Quiz", "No flashcards available for the quiz.")

def ask_question(self):
 if self.quiz_index < self.total_questions:
 flashcard = self.quiz_flashcards[self.quiz_index]
 question = flashcard[1]
 answer = flashcard[2]
 user_answer = simpledialog.askstring("Quiz", question)
 if user_answer:
 if user_answer.lower() == answer.lower():
 messagebox.showinfo("Quiz", "Correct!")
 self.correct_answers += 1
 else:
 messagebox.showinfo(
 "Quiz", f"Wrong! The correct answer is: {answer}")
 self.quiz_index += 1
 self.ask_question()
 else:
 messagebox.showinfo(
 "Quiz Result", f"Your score:
{self.correct_answers}/{self.total_questions}")

```

```

if __name__ == "__main__":
 root = tk.Tk()
 app = FlashcardApp(root)
 root.mainloop()

```

## Flashcard App Guide

### Overview

The Flashcard App is a simple yet effective tool for learning and memorizing information. It allows users

to create, edit, delete, and review flashcards categorized by topics. Additionally, it features a quiz mode to test users on their knowledge of the flashcards.

## **How to Use the Flashcard App**

### **1. Launching the App:**

- Run the script in your **Python** environment to launch the application. The GUI window will open with options for adding, reviewing, searching, and quizzing on flashcards.

### **2. Adding Flashcards:**

- Enter a category, question, and answer in the provided fields.
- Click the "Add Flashcard" button to save the flashcard to the database. If any field is left blank, a warning message will prompt you to fill in all fields.

### **3. Reviewing Flashcards:**

- Click on "Review Flashcards" in the menu to view each flashcard one by one in a message box.

### **4. Editing Flashcards:**

- Select a flashcard from the list, modify the details in the entry fields, and click the "Edit Flashcard" button to save the changes.

## 5. Deleting Flashcards:

- Select a flashcard and click the "Delete Flashcard" button to remove it from the database.

## 6. Searching Flashcards:

- Click on "Search Flashcards" to enter a search term. The app will display flashcards that match the category, question, or answer.

## 7. Starting a Quiz:

- Click "Start Quiz" to randomly select flashcards and prompt the user to answer questions. The app will track correct answers and display the final score at the end of the quiz.

## Installation Guide

To run the Flashcard App, follow these steps:

### 1. Prerequisites

- **Python** Ensure that **Python** is installed on your system. You can download it from [Python.org](https://www.python.org).

- **Visual Studio Code (VSCode):** It is recommended to use VSCode as your code editor. You can download it from [code.visualstudio.com](https://code.visualstudio.com).

## 2. Setting Up the Environment

### 1. Install Python

- Download and install **Python**, ensuring you check the box to add **Python** to your PATH during installation.

### 2. Install SQLite3 (if not already included):

- SQLite3 comes pre-installed with **Python**. You can check by running the command **Python -c "import sqlite3"** in your terminal. If there's no error, it's installed.

### 3. Install Required Libraries:

- The Flashcard App uses the following libraries, which are included in the standard **Python** library:
  - **tkinter**: For creating the GUI.
  - **sqlite3**: For database operations.
- No additional installations are needed.

## 3. Running the App

1. Open VSCode and create a new file (e.g., flashcard\_app.py).
2. Copy and paste the provided code into the file.
3. Save the file and run it using the terminal by navigating to the directory containing the file and executing:

**bash::**

**Code:**

**Python** flashcard\_app.py

## Code Explanation

Here's a detailed breakdown of the code in the Flashcard App:

### 1. Database Setup

**Python**

**Code:**

```
conn = sqlite3.connect('flashcards.db')
```

```
c = conn.cursor()
```

- A connection to an SQLite database named flashcards.db is established. The cursor object c is used to execute SQL commands.

### 2. Schema Update

**Python**

**Code:**

```
def update_schema():
```

```
try:
 c.execute("ALTER TABLE flashcards ADD
COLUMN category TEXT")
 conn.commit()
except sqlite3.OperationalError:
 pass # Column already exists
 • This function adds a new column category to
 the flashcards table if it doesn't already exist. It
 handles the OperationalError in case the
 column is already present.
```

### 3. Flashcards Table Creation

#### Python Code:

```
c.execute("CREATE TABLE IF NOT EXISTS
flashcards (
 id INTEGER PRIMARY KEY,
 category TEXT,
 question TEXT,
 answer TEXT)")
conn.commit()
 • This SQL command creates a flashcards table
 with columns for id, category, question, and
 answer. The id serves as the primary key.
```

### 4. GUI Application Class

#### Python

## Code:

```
class FlashcardApp:
 def __init__(self, root):
 self.root = root
 self.root.title("Flashcard App")
 self.create_widgets()
 self.load_flashcards()
```

...

- The FlashcardApp class initializes the main application window and creates the user interface elements.

## 5. Creating Widgets

**Python**

**Code:**

```
def create_widgets(self):
```

...

- This method sets up the GUI layout, including menus for file operations, search, and quiz functionality. It also creates frames and entry fields for inputting flashcard data.

## 6. Adding Flashcards

**Python**

**Code:**

```
def add_flashcard(self):
```

...

- This method retrieves the data from the entry fields and inserts it into the flashcards table. It also updates the display list of flashcards.

## 7. Loading Flashcards

**Python**

**Code:**

```
def load_flashcards(self):
```

...

- This method fetches all flashcards from the database and displays them in the listbox for

user interaction.

## 8. Editing and Deleting Flashcards

- The methods `edit_flashcard` and `delete_flashcard` handle updating and removing flashcards from the database based on user selection in the listbox.

## 9. Reviewing and Searching Flashcards

- The methods `review_flashcards` and `search_flashcards` allow users to review flashcards in a message box and search for specific flashcards, respectively.

## 10. Quiz Functionality

### Python

#### Code:

```
def start_quiz(self):
```

...

- This method randomly selects flashcards for a quiz. The user is prompted with questions, and their answers are evaluated against the correct answers stored in the database.

```
def ask_question(self):
```

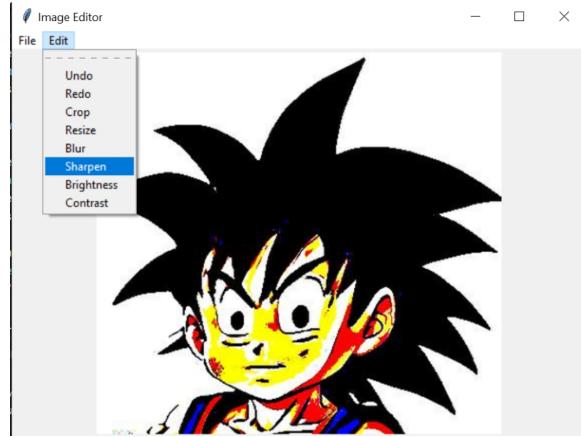
- The `ask_question` method controls the flow of the quiz by sequentially asking questions, evaluating user responses, and providing feedback. It ensures that all questions are asked and the final score is presented once the quiz is complete.

## Conclusion

The Flashcard App is a practical tool for anyone looking to improve their learning through flashcards. It

leverages the simplicity of SQLite for data storage and Tkinter for a user-friendly interface, making it accessible and easy to use. With this guide, you should be able to install, run, and utilize the app effectively.

# Image Editor



```
import tkinter as tk
from tkinter import filedialog, messagebox, Canvas, Toplevel, Scale,
HORIZONTAL
from PIL import Image, ImageTk, ImageFilter, ImageEnhance
```

```
class ImageEditor:
 def __init__(self, root):
 self.root = root
 self.root.title("Image Editor")

 self.canvas = Canvas(root, width=600, height=400)
 self.canvas.pack()

 self.image = None
 self.original_image = None
 self.tk_image = None
 self.undo_stack = []
 self.redo_stack = []
 self.crop_rect = None

 self.create_menu()
 self.crop_box = None

 # Bind keyboard shortcuts
 self.root.bind("<Control-z>", self.undo)
```

```
self.root.bind("<Control-y>", self.redo)

def create_menu(self):
 menu = tk.Menu(self.root)
 self.root.config(menu=menu)

 file_menu = tk.Menu(menu)
 menu.add_cascade(label="File", menu=file_menu)
 file_menu.add_command(label="Open",
command=self.open_image)
 file_menu.add_command(label="Save",
command=self.save_image)
 edit_menu = tk.Menu(menu)
 menu.add_cascade(label="Edit", menu=edit_menu)
 edit_menu.add_command(label="Undo", command=self.undo)
 edit_menu.add_command(label="Redo", command=self.redo)
 edit_menu.add_command(label="Crop",
command=self.start_crop)
 edit_menu.add_command(label="Resize",
command=self.resize_image)
 edit_menu.add_command(label="Blur",
command=self.apply_blur)
 edit_menu.add_command(label="Sharpen",
command=self.apply_sharpen)
 edit_menu.add_command(label="Brightness",
command=self.adjust_brightness)
 edit_menu.add_command(label="Contrast",
command=self.adjust_contrast)

def open_image(self):
 file_path = filedialog.askopenfilename(
 filetypes=[("Image files", "*.*")]
)
 if file_path:
 self.image = Image.open(file_path)
 self.original_image = self.image.copy()
 self.undo_stack.append(self.image.copy())
 self.display_image(self.image)
```

```
 self.clear_crop_rectangle()

def save_image(self):
 if self.image:
 file_path =
filedialog.asksaveasfilename(defaultextension=".png", filetypes=[(
 "PNG files", "*.*"), ("JPEG files", "*.*"), ("BMP files", "*.*")])
 if file_path:
 if file_path.lower().endswith('.jpg', '.jpeg'):
 if self.image.mode == 'RGBA':
 self.image = self.image.convert('RGB')
 self.image.save(file_path)
 else:
 messagebox.showwarning("Save Image", "Save operation
cancelled.")
 else:
 messagebox.showwarning("Save Image", "No image to save.")
```

```
def display_image(self, image):
 img_ratio = image.width / image.height
 canvas_ratio = self.canvas.winfo_width() /
self.canvas.winfo_height()

 if canvas_ratio > img_ratio:
 height = self.canvas.winfo_height()
 width = int(height * img_ratio)
 else:
 width = self.canvas.winfo_width()
 height = int(width / img_ratio)

 resized_image = image.resize((width, height), Image.LANCZOS)
 self.tk_image = ImageTk.PhotoImage(resized_image)

 self.canvas.create_image(self.canvas.winfo_width(
```

```
) // 2, self.canvas.winfo_height() // 2, image=self.tk_image,
anchor=tk.CENTER)

def clear_crop_rectangle(self):
 if self.crop_rect:
 self.canvas.delete(self.crop_rect)
 self.crop_rect = None

def start_crop(self):
 if self.image:
 self.canvas.bind("<ButtonPress-1>", self.on_button_press)
 self.canvas.bind("<B1-Motion>", self.on_move_press)
 self.canvas.bind("<ButtonRelease-1>", self.on_button_release)
 else:
 messagebox.showwarning("Crop Image", "No image to crop.")

def on_button_press(self, event):
 self.crop_box = [event.x, event.y, event.x, event.y]
 if self.crop_rect:
 self.canvas.delete(self.crop_rect)
 self.crop_rect = self.canvas.create_rectangle(
 *self.crop_box, outline='red')

def on_move_press(self, event):
 self.crop_box[2], self.crop_box[3] = event.x, event.y
 self.canvas.coords(self.crop_rect, *self.crop_box)

def on_button_release(self, event):
 if self.crop_box:
 self.crop_image()
 self.canvas.unbind("<ButtonPress-1>")
 self.canvas.unbind("<B1-Motion>")
 self.canvas.unbind("<ButtonRelease-1>")

def crop_image(self):
 x1, y1, x2, y2 = self.crop_box
 ratio_w = self.image.width / self.canvas.winfo_width()
```

```
ratio_h = self.image.height / self.canvas.winfo_height()

crop_area = (int(x1 * ratio_w), int(y1 * ratio_h),
 int(x2 * ratio_w), int(y2 * ratio_h))
self.image = self.image.crop(crop_area)
self.add_undo(self.image.copy())
self.original_image = self.image.copy()
self.display_image(self.image)
self.clear_crop_rectangle()

def resize_image(self):
 if self.image:
 width, height = 200, 200 # Example new size
 self.image = self.image.resize((width, height))
 self.add_undo(self.image.copy())
 self.original_image = self.image.copy()
 self.display_image(self.image)
 else:
 messagebox.showwarning("Resize Image", "No image to
resize.")

def apply_blur(self):
 if self.image:
 self.show_blur_settings()
 else:
 messagebox.showwarning("Apply Blur", "No image to apply
blur.")

def apply_sharpen(self):
 if self.image:
 self.show_sharpen_settings()
 else:
 messagebox.showwarning(
 "Apply Sharpen", "No image to apply sharpen.")

def adjust_brightness(self):
 if self.image:
```

```
 self.show_brightness_settings()
 else:
 messagebox.showwarning("Adjust Brightness",
 "No image to adjust brightness.")

def adjust_contrast(self):
 if self.image:
 self.show_contrast_settings()
 else:
 messagebox.showwarning(
 "Adjust Contrast", "No image to adjust contrast.")

def show_blur_settings(self):
 settings_win = Toplevel(self.root)
 settings_win.title("Blur Settings")

 tk.Label(settings_win, text="Blur Radius:").pack()
 blur_radius = tk.DoubleVar(value=2.0)
 Scale(settings_win, from_=0, to=10, orient=HORIZONTAL,
 variable=blur_radius,
 command=lambda x:
 self.update_blur(blur_radius.get())).pack()

 def apply_settings():
 self.add_undo(self.image.copy())
 self.original_image = self.image.copy()
 settings_win.destroy()

 tk.Button(settings_win, text="Apply",
 command=apply_settings).pack()

def update_blur(self, radius):
 self.image = self.original_image.filter(
 ImageFilter.GaussianBlur(radius))
 self.display_image(self.image)

def show_sharpen_settings(self):
```

```
settings_win = Toplevel(self.root)
settings_win.title("Sharpen Settings")

tk.Label(settings_win, text="Sharpen Factor:").pack()
sharpen_factor = tk.DoubleVar(value=2.0)
Scale(settings_win, from_=0, to=10, orient=HORIZONTAL,
variable=sharpen_factor,
 command=lambda x:
self.update_sharpen(sharpen_factor.get())).pack()

def apply_settings():
 self.add_undo(self.image.copy())
 self.original_image = self.image.copy()
 settings_win.destroy()

tk.Button(settings_win, text="Apply",
command=apply_settings).pack()

def update_sharpen(self, factor):
 self.image = self.original_image.filter(
 ImageFilter.UnsharpMask(factor))
 self.display_image(self.image)

def show_brightness_settings(self):
 settings_win = Toplevel(self.root)
 settings_win.title("Brightness Settings")

 tk.Label(settings_win, text="Brightness Level:").pack()
 brightness_level = tk.DoubleVar(value=1.0)
 Scale(settings_win, from_=0.5, to=2.0, resolution=0.1,
orient=HORIZONTAL, variable=brightness_level,
 command=lambda x:
self.update_brightness(brightness_level.get())).pack()

 def apply_settings():
 self.add_undo(self.image.copy())
 self.original_image = self.image.copy()
```

```
settings_win.destroy()

tk.Button(settings_win, text="Apply",
command=apply_settings).pack()

def update_brightness(self, level):
 enhancer = ImageEnhance.Brightness(self.original_image)
 self.image = enhancer.enhance(level)
 self.display_image(self.image)

def show_contrast_settings(self):
 settings_win = Toplevel(self.root)
 settings_win.title("Contrast Settings")

 tk.Label(settings_win, text="Contrast Level:").pack()
 contrast_level = tk.DoubleVar(value=1.0)
 Scale(settings_win, from_=0.5, to=2.0, resolution=0.1,
orient=HORIZONTAL,
 variable=contrast_level, command=lambda x:
self.update_contrast(contrast_level.get())).pack()

 def apply_settings():
 self.add_undo(self.image.copy())
 self.original_image = self.image.copy()
 settings_win.destroy()

 tk.Button(settings_win, text="Apply",
command=apply_settings).pack()

def update_contrast(self, level):
 enhancer = ImageEnhance.Contrast(self.original_image)
 self.image = enhancer.enhance(level)
 self.display_image(self.image)

def add_undo(self, image):
 self.undo_stack.append(image)
 self.redo_stack.clear()
```

```
def undo(self, event=None):
 if len(self.undo_stack) > 1:
 self.redo_stack.append(self.undo_stack.pop())
 self.image = self.undo_stack[-1]
 self.original_image = self.image.copy()
 self.display_image(self.image)
 else:
 messagebox.showwarning("Undo", "No more actions to
undo.")

def redo(self, event=None):
 if self.redo_stack:
 self.image = self.redo_stack.pop()
 self.undo_stack.append(self.image)
 self.original_image = self.image.copy()
 self.display_image(self.image)
 else:
 messagebox.showwarning("Redo", "No more actions to redo.")

if __name__ == "__main__":
 root = tk.Tk()
 app = ImageEditor(root)
 root.mainloop()
```

## How to Install and Run the Image Editor in VSCode

### Step 1: Install Python

Ensure you have **Python** installed on your computer. You can download it from the official website: [Python Downloads](#). During installation, make sure to check the box that says "Add **Python** to PATH."

### Step 2: Install Visual Studio Code

If you haven't already, download and install Visual Studio Code from the official website: [VSCode Downloads](#).

### Step 3: Install Required Libraries

You will need to install the following **Python** libraries:

1. **Pillow**: For image processing.
2. **tkinter**: This is included with **Python** installations by default, so you typically don't need to install it separately.

To install Pillow, open a terminal (you can do this in VSCode) and run:

**bash::**

**Code:**

**pip** install Pillow

### Step 4: Create a New **Python** File

1. Open VSCode.
2. Create a new folder for your project.
3. Open that folder in VSCode.
4. Create a new file named `image_editor.py` and copy the provided code into this file.

### Step 5: Run the Program

1. Open a terminal in VSCode (`View > Terminal`).

2. Navigate to the directory where your image\_editor.py file is located.
3. Run the program by executing:

**bash::**

**Code:**

**Python** image\_editor.py

## Step 6: Use the Image Editor

Once the program is running, a window will appear with the image editor interface. You can open images, apply edits, and save the modified images using the available options in the menu.

## Description of Each Function

### Class: ImageEditor

The ImageEditor class manages the overall functionality of the image editor. It initializes the GUI, handles image loading, processing, and user interactions.

#### **\_\_init\_\_(self, root)**

- **Parameters:** root (the main window of the application).
- **Description:** Initializes the main application window, sets up the canvas for image display, and creates the menu bar. It also prepares the undo and redo stacks for image editing actions.

#### **create\_menu(self)**

- **Description:** Creates the menu bar with options for file operations (open, save) and editing

functions (undo, redo, crop, resize, blur, sharpen, brightness, contrast).

### **open\_image(self)**

- **Description:** Opens a file dialog to select an image file. Loads the selected image, saves a copy to the undo stack, and displays the image on the canvas. It also resets the crop rectangle.

### **save\_image(self)**

- **Description:** Opens a file dialog to save the edited image. It checks the image format and handles the conversion if needed (e.g., converting RGBA to RGB). It then saves the image to the specified location.

## **display\_image(self, image)**

- **Parameters:** image (the PIL Image to display).
- **Description:** Resizes the image to fit the canvas while maintaining the aspect ratio. It then converts the image to a format suitable for display in the Tkinter canvas and renders it.

## **clear\_crop\_rectangle(self)**

- **Description:** Clears the crop rectangle if it exists. This is called after a cropping operation is completed or canceled.

## **start\_crop(self)**

- **Description:** Binds mouse events to start the cropping process. If no image is loaded, it displays a warning.

## **on\_button\_press(self, event)**

- **Parameters:** event (mouse event).
- **Description:** Initializes the crop box when the left mouse button is pressed. It creates a rectangle on the canvas to indicate the cropping area.

## **on\_move\_press(self, event)**

- **Parameters:** event (mouse event).
- **Description:** Updates the dimensions of the crop rectangle as the mouse moves while the

button is pressed.

### **on\_button\_release(self, event)**

- **Parameters:** event (mouse event).
- **Description:** Finalizes the crop operation and unbinds the mouse events.

### **crop\_image(self)**

- **Description:** Crops the image based on the defined crop rectangle and updates the image display. It also saves the cropped image to the undo stack.

### **resize\_image(self)**

- **Description:** Resizes the image to a fixed size (200x200 pixels in this case). Updates the undo stack and displays the resized image.

### **apply\_blur(self)**

- **Description:** Opens a settings window for the user to set the blur radius. Displays a warning if no image is loaded.

### **apply\_sharpen(self)**

- **Description:** Similar to apply\_blur, this opens a settings window for sharpening the image. Displays a warning if no image is loaded.

### **adjust\_brightness(self)**

- **Description:** Opens a settings window for adjusting the brightness of the image. Displays a warning if no image is loaded.

### **adjust\_contrast(self)**

- **Description:** Opens a settings window for adjusting the contrast of the image. Displays a warning if no image is loaded.

### **show\_blur\_settings(self)**

- **Description:** Creates a new window to set the blur radius. Contains a slider for the user to adjust the radius and an apply button to confirm changes.

### **update\_blur(self, radius)**

- **Parameters:** radius (the radius for the Gaussian blur).
- **Description:** Applies Gaussian blur to the image based on the selected radius and updates the display.

### **show\_sharpen\_settings(self)**

- **Description:** Similar to show\_blur\_settings, but for sharpening the image. Contains a slider to adjust the sharpen factor.

### **update\_sharpen(self, factor)**

- **Parameters:** factor (the factor for sharpening).

- **Description:** Applies sharpening to the image based on the selected factor and updates the display.

### **show\_brightness\_settings(self)**

- **Description:** Creates a new window to set the brightness level. Contains a slider for the user to adjust the brightness level.

### **update\_brightness(self, level)**

- **Parameters:** level (the brightness level).
- **Description:** Adjusts the brightness of the image and updates the display.

### **show\_contrast\_settings(self)**

- **Description:** Creates a new window to set the contrast level. Contains a slider for the user to adjust the contrast level.

### **update\_contrast(self, level)**

- **Parameters:** level (the contrast level).
- **Description:** Adjusts the contrast of the image and updates the display.

### **add\_undo(self, image)**

- **Parameters:** image (the current state of the image).
- **Description:** Adds the current image to the undo stack and clears the redo stack.

## **undo(self, event=None)**

- **Parameters:** event (optional; used for keyboard shortcut).
- **Description:** Reverts the last action by popping the most recent image from the undo stack and displaying it. Displays a warning if no actions are left to undo.

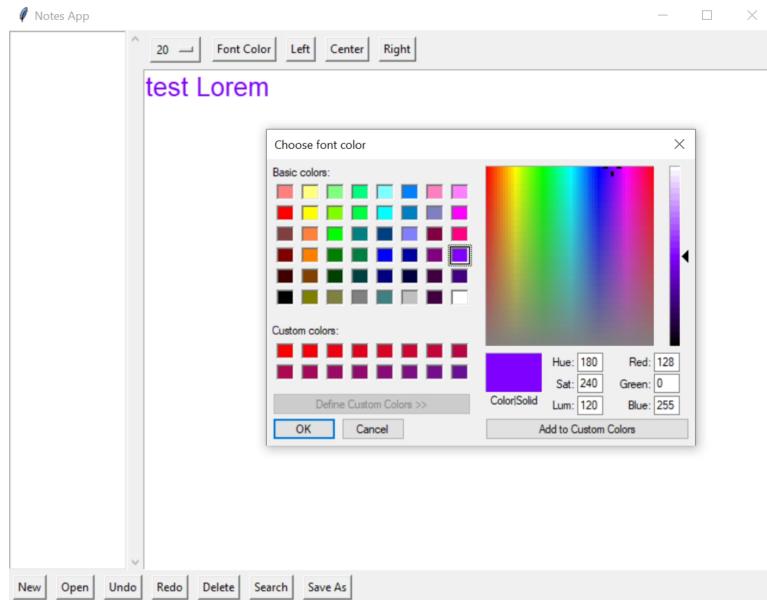
## **redo(self, event=None)**

- **Parameters:** event (optional; used for keyboard shortcut).
- **Description:** Re-applies the most recent undone action by popping from the redo stack and displaying it. Displays a warning if no actions are left to redo.

## **Running the Code**

Once you have followed all the steps, you can run the image editor and start using it to edit your images interactively. The GUI allows you to perform various edits, such as cropping, resizing, applying filters, and saving your edited images.

# Notes



```
import tkinter as tk
from tkinter import messagebox, filedialog, colorchooser, simpledialog
import os
from tkinter import font as tkFont
import re
from docx import Document
from docx.shared import Pt, RGBColor
from PIL import ImageColor
```

```
class NotesApp:
 def __init__(self, root):
 self.root = root
 self.root.title("Notes App")
 self.root.geometry("800x600")

 self.notes = []
 self.current_note = None
 self.color_code = 'black' # Default color code
 self.setup_ui()

 def setup_ui(self):
```

```
Setting up the frames
self.left_frame = tk.Frame(self.root, bg="lightgray")
self.left_frame.grid(row=0, column=0, rowspan=2, sticky="ns")

self.top_frame = tk.Frame(self.root)
self.top_frame.grid(row=0, column=1, sticky="ew")

self.right_frame = tk.Frame(self.root)
self.right_frame.grid(row=1, column=1, sticky="nsew")

self.bottom_frame = tk.Frame(self.root)
self.bottom_frame.grid(row=2, column=0, columnspan=2,
sticky="ew")

self.root.grid_rowconfigure(1, weight=1)
self.root.grid_columnconfigure(1, weight=1)

Listbox to display note titles
self.notes_listbox = tk.Listbox(self.left_frame, bg="white")
self.notes_listbox.pack(side="left", fill="y", expand=True)
self.notes_listbox.bind('<<ListboxSelect>>',
self.load_selected_note)

Scrollbar for the listbox
self.scrollbar = tk.Scrollbar(self.left_frame)
self.scrollbar.pack(side="right", fill="y")
self.notes_listbox.config(yscrollcommand=self.scrollbar.set)
self.scrollbar.config(command=self.notes_listbox.yview)

Font options frame
self.font_frame = tk.Frame(self.top_frame)
self.font_frame.pack(side="top", fill="x")

Font size option
self.font_size_var = tk.StringVar(value="12")
self.font_size_menu = tk.OptionMenu(self.font_frame,
self.font_size_var, "8",
```

```
 "10", "12", "14", "16", "18", "20",
"24", command=self.change_font_size)
self.font_size_menu.pack(side="left", padx=5, pady=5)

Font color option
self.font_color_button = tk.Button(
 self.font_frame, text="Font Color",
command=self.change_font_color)
self.font_color_button.pack(side="left", padx=5, pady=5)

Justify options
self.justify_left_button = tk.Button(
 self.font_frame, text="Left", command=self.justify_left)
self.justify_left_button.pack(side="left", padx=5, pady=5)

self.justify_center_button = tk.Button(
 self.font_frame, text="Center", command=self.justify_center)
self.justify_center_button.pack(side="left", padx=5, pady=5)

self.justify_right_button = tk.Button(
 self.font_frame, text="Right", command=self.justify_right)
self.justify_right_button.pack(side="left", padx=5, pady=5)

Text area to write notes
self.text_area = tk.Text(
 self.right_frame, wrap='word', undo=True, font=("Arial", 12))
self.text_area.pack(expand=1, fill='both')

Buttons for note actions
self.new_button = tk.Button(
 self.bottom_frame, text="New", command=self.new_note)
self.new_button.pack(side='left', padx=5, pady=5)

self.open_button = tk.Button(
 self.bottom_frame, text="Open", command=self.open_notes)
self.open_button.pack(side='left', padx=5, pady=5)
```

```
self.undo_button = tk.Button(
 self.bottom_frame, text="Undo", command=self.undo)
self.undo_button.pack(side='left', padx=5, pady=5)

self.redo_button = tk.Button(
 self.bottom_frame, text="Redo", command=self.redo)
self.redo_button.pack(side='left', padx=5, pady=5)

self.delete_button = tk.Button(
 self.bottom_frame, text="Delete", command=self.delete_note)
self.delete_button.pack(side='left', padx=5, pady=5)

self.search_button = tk.Button(
 self.bottom_frame, text="Search",
 command=self.search_notes)
self.search_button.pack(side='left', padx=5, pady=5)

self.save_button = tk.Button(
 self.bottom_frame, text="Save As",
 command=self.save_note_as)
self.save_button.pack(side='left', padx=5, pady=5)

Load notes into the listbox
self.load_notes_into_listbox()

def new_note(self):
 self.text_area.delete(1.0, tk.END)
 self.current_note = None
 print("Debug: Created new note. Current note is None.") # Debug
statement

def open_notes(self):
 file_paths = filedialog.askopenfilenames(defaultextension=".txt",
filetypes=[(
 "Text files", "*.*"), ("Word files", "*.*.docx"), ("All files",
"*.*")])
 if file_paths:
```

```
self.notes = {} # Clear current notes
self.notes_listbox.delete(0, tk.END) # Clear the listbox

for path in file_paths:
 filename = os.path.basename(path)
 file_content = self.load_file_content(path)
 # Store content with filename
 self.notes[filename] = file_content

 self.load_notes_into_listbox()
 self.current_note = None # Clear current note after opening
new notes
 messagebox.showinfo(
 "Info", f"Opened files: {', '.join(file_paths)}")

def load_file_content(self, file_path):
 try:
 if file_path.endswith(".txt"):
 with open(file_path, 'r', encoding='utf-8') as file:
 content = file.read()
 elif file_path.endswith(".docx"):
 doc = Document(file_path)
 content = '\n'.join([p.text for p in doc.paragraphs])
 else:
 content = "Unsupported file type"
 return {
 'content': content,
 'font_size': '12',
 'font_color': self.color_code, # Use default color code
 'justification': 'left'
 }
except Exception as e:
 messagebox.showerror("Error", f"Failed to load file: {e}")
return {
 'content': "Error loading file",
 'font_size': '12',
 'font_color': self.color_code, # Use default color code
```

```
 'justification': 'left'
 }

def load_notes_into_listbox(self):
 self.notes_listbox.delete(0, tk.END)
 for note_title in self.notes.keys():
 self.notes_listbox.insert(tk.END, note_title)

def load_selected_note(self, event):
 selection = self.notes_listbox.curselection()
 if selection:
 index = selection[0]
 if index < len(self.notes_listbox.get(0, tk.END)):
 note_title = self.notes_listbox.get(index)
 if note_title in self.notes:
 note_data = self.notes[note_title]
 self.text_area.delete(1.0, tk.END)
 self.text_area.insert(tk.END, note_data['content'])
 self.text_area.configure(
 font=("Arial", note_data['font_size']))
 self.text_area.tag_configure(
 "color", foreground=note_data['font_color'])
 self.text_area.tag_add("color", 1.0, tk.END)
 self.apply_justification(note_data['justification'])
 self.current_note = note_title
 # Debug statement
 print(f"Debug: Loaded note '{self.current_note}'")
 else:
 messagebox.showwarning("Warning", "Note title not found.")
 self.current_note = None
 else:
 messagebox.showwarning(
 "Warning", "Selected index is out of range.")
 self.current_note = None
 else:
 messagebox.showwarning("Warning", "No note selected.")
```

```
 self.current_note = None

def delete_note(self):
 selection = self.notes_listbox.curselection()
 if selection:
 index = selection[0]
 if index < len(self.notes_listbox.get(0, tk.END)):
 note_title = self.notes_listbox.get(index)
 del self.notes[note_title]
 self.load_notes_into_listbox()
 self.text_area.delete(1.0, tk.END)
 self.current_note = None
 else:
 messagebox.showwarning(
 "Warning", "Selected index is out of range.")
else:
 messagebox.showwarning("Warning", "No note selected.")

def search_notes(self):
 query = simpledialog.askstring("Search", "Enter search query:")
 if query:
 results = [title for title in self.notes if re.search(
 query, title, re.IGNORECASE)]
 if results:
 self.notes_listbox.delete(0, tk.END)
 for result in results:
 self.notes_listbox.insert(tk.END, result)
 else:
 messagebox.showinfo("Info", "No notes found.")

def change_font_size(self, size):
 try:
 size = int(size)
 self.text_area.configure(font=("Arial", size))
 except ValueError:
 messagebox.showerror("Error", "Invalid font size.")
```

```
def change_font_color(self):
 color_code = colorchooser.askcolor(title="Choose font color")[1]
 if color_code:
 self.color_code = color_code
 if self.current_note:
 self.notes[self.current_note]['font_color'] = color_code
 self.text_area.tag_configure("color", foreground=color_code)
 self.text_area.tag_add("color", 1.0, tk.END)

def justify_left(self):
 self.apply_justification('left')

def justify_center(self):
 self.apply_justification('center')

def justify_right(self):
 self.apply_justification('right')

def apply_justification(self, justification):
 if justification == 'left':
 self.text_area.tag_configure("justify", justify=tk.LEFT)
 elif justification == 'center':
 self.text_area.tag_configure("justify", justify=tk.CENTER)
 elif justification == 'right':
 self.text_area.tag_configure("justify", justify=tk.RIGHT)
 self.text_area.tag_add("justify", 1.0, tk.END)
 if self.current_note:
 self.notes[self.current_note]['justification'] = justification

def save_note_as(self):
 file_path = filedialog.asksaveasfilename(defaultextension=".txt",
 filetypes=[(
 "Text files", "*.txt"), ("Word files", "*.docx"), ("All files",
 "*.*")])
 if file_path:
 try:
 if file_path.endswith(".txt"):
```

```

 self.save_as_text_file(file_path)
 elif file_path.endswith(".docx"):
 self.save_as_docx_file(file_path)
 messagebox.showinfo("Info", f"Note saved to: {file_path}")
 except Exception as e:
 messagebox.showerror("Error", f"Failed to save file: {e}")

def save_as_text_file(self, file_path):
 with open(file_path, 'w', encoding='utf-8') as file:
 file.write(self.text_area.get(1.0, tk.END))

def save_as_docx_file(self, file_path):
 try:
 doc = Document()
 para = doc.add_paragraph()
 run = para.add_run(self.text_area.get(1.0, tk.END))
 font = run.font

 # Debugging font size extraction
 try:
 font_size = int(self.text_area.cget('font').split()[1])
 font.size = Pt(font_size)
 except (ValueError, IndexError) as e:
 print(f"Debug: Error extracting font size: {e}")
 font.size = Pt(12) # Default size if extraction fails

 # Debugging color code handling
 try:
 color_rgb = ImageColor.getrgb(self.color_code)
 font.color.rgb = RGBColor(*color_rgb)
 except (ValueError, TypeError) as e:
 print(f"Debug: Error handling font color: {e}")
 # Default color if handling fails
 font.color.rgb = RGBColor(0, 0, 0)

 doc.save(file_path)
 print(f"Debug: File saved successfully to {file_path}")

```

```

except Exception as e:
 print(f"Debug: Error saving docx file: {e}")
 messagebox.showerror("Error", f"Failed to save file: {e}")

def undo(self):
 self.text_area.edit_undo()

def redo(self):
 self.text_area.edit_redo()

def save_note_content(self):
 if self.current_note:
 file_path = filedialog.asksaveasfilename(
 defaultextension=".docx", filetypes=[("Word files",
 "*.docx")])
 if file_path:
 try:
 self.save_as_docx_file(file_path)
 messagebox.showinfo(
 "Info", f"Note content saved to: {file_path}")
 except Exception as e:
 messagebox.showerror(
 "Error", f"Failed to save note content: {e}")

 else:
 messagebox.showwarning("Warning", "No note selected to
save.")

if __name__ == "__main__":
 root = tk.Tk()
 app = NotesApp(root)
 root.mainloop()

```

## Detailed Guide to Using the Notes App

This guide will provide a comprehensive overview of the NotesApp code, including how to set up the development environment, install required packages, run the application, and dive deep into the details of the code.

## 1. Installation and Setup

### 1.1 Requirements

To run the NotesApp, you'll need the following:

- **Python 3.6 or newer:** Ensure that you have a compatible **Python** version installed.
- **Python packages:** tkinter, docx, Pillow.

### 1.2 Installing Python and Packages

#### 1. Install Python

- Download and install **Python** from [Python.org](https://www.python.org).
- Ensure **Python** is added to your system's PATH during installation.

#### 2. Install Required Python Packages:

Open a terminal or command prompt and execute the following commands to install the required packages:

**bash:**

## **Code:**

**pip** install **Python-docx** Pillow

- **Python-docx**: For handling .docx files.
- Pillow: For color handling (used indirectly through ImageColor).

## **1.3 Running the Application**

1. **Save the Code**: Save the provided code in a file named notes\_app.py.
2. **Run the Application**: Open a terminal or command prompt, navigate to the directory where notes\_app.py is saved, and execute:

### **bash:**

## **Code:**

**Python** notes\_app.py

This will launch the NotesApp GUI application.

## **2. Setting Up in VS Code**

### **2.1 Installing VS Code**

1. Download and install [Visual Studio Code \(VS Code\)](#).
2. **Install Python Extension**:
  - Open VS Code.
  - Go to Extensions (Ctrl+Shift+X).

- Search for "**Python**" and install the extension provided by Microsoft.

## 2.2 Open and Run the Code

### 1. Open the Project:

- Open VS Code and go to File > Open File....
- Select the notes\_app.py file.

### 2. Run the Code:

- Open a new terminal in VS Code (Terminal > New Terminal).
- Run the script by executing:

**bash:**

**Code:**

**Python** notes\_app.py



## 3. Deep Dive into the Code

### 3.1 Overview

The NotesApp is a simple text editor application with functionalities to create, open, edit, save, and delete notes. It supports both plain text (.txt) and Word document (.docx) formats. The application is built using the tkinter library for the GUI.

### 3.2 Detailed Breakdown

#### 3.2.1 `__init__` Method

##### Python

##### Code:

```
def __init__(self, root):
 self.root = root
 self.root.title("Notes App")
 self.root.geometry("800x600")
```

- Initializes the application window with a title and size.
- Sets up initial properties, such as notes (a dictionary to store note contents) and current\_note (to keep track of the currently selected note).

#### 3.2.2 `setup_ui` Method

This method sets up the entire user interface of the application.

## **1. Frames:**

- `left_frame`: Contains the listbox for note titles and its scrollbar.
- `top_frame`: Contains font-related options (size, color, alignment).
- `right_frame`: Contains the main text area for editing notes.
- `bottom_frame`: Contains buttons for note actions.

## **2. Listbox:**

- Displays the list of note titles.
- Allows note selection to load the content into the text area.

## **3. Font Options:**

- Font size selection (OptionMenu).
- Font color selection (Button with color chooser).
- Text alignment options (Button for left, center, right alignment).

## **4. Text Area:**

- Main editing area for notes.

## **5. Action Buttons:**

- New, Open, Undo, Redo, Delete, Search, Save As.

### 3.2.3 new\_note Method

**Python**

**Code:**

```
def new_note(self):
 self.text_area.delete(1.0, tk.END)
 self.current_note = None
```

- Clears the text area and resets the current\_note.

### 3.2.4 open\_notes Method

**Python**

**Code:**

```
def open_notes(self):
 file_paths = filedialog.askopenfilenames(...)
 if file_paths:
 self.notes = {}
 self.notes_listbox.delete(0, tk.END)
 for path in file_paths:
 filename = os.path.basename(path)
 file_content = self.load_file_content(path)
 self.notes[filename] = file_content
 self.load_notes_into_listbox()
 self.current_note = None
```

- Opens multiple files and loads their content.
- Updates the listbox and notes dictionary.

### 3.2.5 load\_file\_content Method

Python

Code:

```
def load_file_content(self, file_path):
 try:
 if file_path.endswith(".txt"):
 with open(file_path, 'r', encoding='utf-8') as
file:
 content = file.read()
 elif file_path.endswith(".docx"):
 doc = Document(file_path)
 content = '\n'.join([p.text for p in
doc.paragraphs])
 else:
 content = "Unsupported file type"
 return {
 'content': content,
 'font_size': '12',
 'font_color': self.color_code,
 'justification': 'left'
 }
except Exception as e:
 messagebox.showerror("Error", f"Failed to load
file: {e}")
 return {
```

```
'content': "Error loading file",
'font_size': '12',
'font_color': self.color_code,
'justification': 'left'
}
```

- Handles loading content from .txt and .docx files.
- Provides default settings if loading fails.

### **3.2.6 load\_notes\_into\_listbox Method**

**Python**

**Code:**

```
def load_notes_into_listbox(self):
 self.notes_listbox.delete(0, tk.END)
 for note_title in self.notes.keys():
 self.notes_listbox.insert(tk.END, note_title)
 • Populates the listbox with note titles from the
 notes dictionary.
```

### **3.2.7 load\_selected\_note Method**

**Python**

**Code:**

```
def load_selected_note(self, event):
 ...
 if note_title in self.notes:
 note_data = self.notes[note_title]
 self.text_area.delete(1.0, tk.END)
 self.text_area.insert(tk.END, note_data['content'])
 self.text_area.configure(font=("Arial",
note_data['font_size']))
 self.text_area.tag_configure("color",
foreground=note_data['font_color'])
 self.text_area.tag_add("color", 1.0, tk.END)
 self.apply_justification(note_data['justification'])
 self.current_note = note_title
```

- Loads the selected note into the text area, applying stored formatting.

### **3.2.8 delete\_note Method**

**Python**

**Code:**

```
def delete_note(self):
 selection = self.notes_listbox.curselection()
 if selection:
 index = selection[0]
 if index < len(self.notes_listbox.get(0, tk.END)):
 note_title = self.notes_listbox.get(index)
 del self.notes[note_title]
 self.load_notes_into_listbox()
 self.text_area.delete(1.0, tk.END)
 self.current_note = None
 • Deletes the selected note from both the notes
 dictionary and the listbox.
```

### **3.2.9 search\_notes Method**

**Python**

**Code:**

```
def search_notes(self):
 query = simpledialog.askstring("Search", "Enter
search query:")
 if query:
 results = [title for title in self.notes if
re.search(query, title, re.IGNORECASE)]
 if results:
```

```
self.notes_listbox.delete(0, tk.END)
for result in results:
 self.notes_listbox.insert(tk.END, result)
else:
 messagebox.showinfo("Info", "No notes
found.")
```

- Searches for notes by title and displays matching results in the listbox.

### **3.2.10 change\_font\_size Method**

**Python**

**Code:**

```
def change_font_size(self, size):
 try:
 size = int(size)
 self.text_area.configure(font=("Arial", size))
 except ValueError:
 messagebox.showerror("Error", "Invalid font
size.")
```

- Changes the font size of the text in the text area.

### **3.2.11 change\_font\_color Method**

**Python**

**Code:**

```
def change_font_color(self):
 color_code = colorchooser.askcolor(title="Choose
font color")[1]
 if color_code:
 self.color_code = color_code
 if self.current_note:
 self.notes[self.current_note]['font_color'] =
color_code
 self.text_area.tag_configure("color",
foreground=color_code)
```

- ```
self.text_area.tag_add("color", 1.0, tk.END)
```
- Allows users to choose a font color and updates the text area and current note's color.

3.2.12 **justify_left**, **justify_center**, **justify_right** Methods

Python

Code:

```
def justify_left(self):
    self.apply_justification('left')

def justify_center(self):
    self.apply_justification('center')

def justify_right(self):
    self.apply_justification('right')

def apply_justification(self, justification):
    self.text_area.tag_configure("justify",
justify=justification)
    self.text_area.tag_add("justify", 1.0, tk.END)
```

- Methods for aligning text to the left, center, or right.

3.2.13 **save_note** Method

Python

Code:

```
def save_note(self):
```

```
if self.current_note:  
    self.notes[self.current_note]['content'] =  
        self.text_area.get(1.0, tk.END)  
    if self.current_note.endswith(".txt"):  
        with open(self.current_note, 'w', encoding='utf-  
8') as file:  
            file.write(self.notes[self.current_note]  
['content'])  
    elif self.current_note.endswith(".docx"):  
        doc = Document()  
        doc.add_paragraph(self.notes[self.current_note]  
['content'])  
        doc.save(self.current_note)  
        messagebox.showinfo("Info", "Note saved  
successfully.")  
    else:  
        self.save_as()  
    • Saves the current note to its file if it exists;  
    otherwise, it prompts for a save-as action.
```

3.2.14 save_as Method

Python

Code:

```
def save_as(self):  
    file_path = filedialog.asksaveasfilename(  
        defaultextension=".txt",  
        filetypes=[("Text Files", "*.txt"), ("Word  
Documents", "*.docx")])
```

```
)
```

```
if file_path:
```

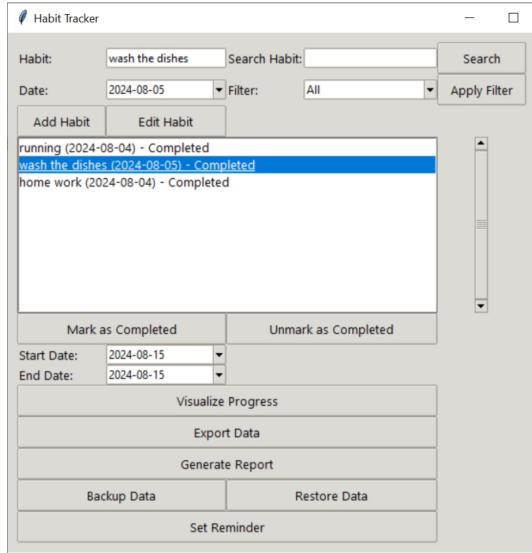
```
    self.current_note = file_path  
    self.save_note()
```

- Prompts the user to select a location and file type to save the current note.

Conclusion

The NotesApp is a versatile text editor application built with tkinter for managing notes. It supports both text and Word document formats, providing essential functionalities like creating, opening, editing, and saving notes. By following this guide, you should be able to set up, run, and understand the application in detail

Habit Tracker



```
import tkinter as tk
from tkinter import ttk
import sqlite3
import datetime
import matplotlib.pyplot as plt
from tkcalendar import DateEntry
from tkinter import messagebox
from threading import Timer

class HabitTrackerApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Habit Tracker")
        self.conn = sqlite3.connect("habits.db")
        self.create_table()

        self.habit_name_var = tk.StringVar()
        self.date_var = tk.StringVar(
            value=datetime.date.today().strftime("%Y-%m-%d"))
        self.start_date_var = tk.StringVar(
            value=datetime.date.today().strftime("%Y-%m-%d"))
        self.end_date_var = tk.StringVar()
```

```

        value=datetime.date.today().strftime("%Y-%m-%d"))
self.search_var = tk.StringVar()
self.filter_var = tk.StringVar(value="All")
self.selected_habit_id = None # To store the ID of the selected
habit for editing

self.setup_ui()

def create_table(self):
    with self.conn:
        self.conn.execute("""
            CREATE TABLE IF NOT EXISTS habits (
                id INTEGER PRIMARY KEY,
                name TEXT NOT NULL,
                date TEXT NOT NULL,
                completed BOOLEAN NOT NULL
            )
        """)

def setup_ui(self):
    style = ttk.Style(self.root)
    style.theme_use('clam') # Use a modern theme
    style.configure("TButton", font=("Segoe UI", 10))
    style.configure("TLabel", font=("Segoe UI", 10))
    style.configure("TEntry", font=("Segoe UI", 10))

    frame = ttk.Frame(self.root, padding="10")
    frame.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))

    ttk.Label(frame, text="Habit:").grid(row=0, column=0,
                                         sticky=tk.W)
    ttk.Entry(frame, textvariable=self.habit_name_var).grid(
        row=0, column=1, sticky=(tk.W, tk.E))

    ttk.Label(frame, text="Date:").grid(row=1, column=0,
                                         sticky=tk.W)
    DateEntry(frame, textvariable=self.date_var,

```

```
        date_pattern='yyyy-mm-dd').grid(row=1, column=1,
sticky=(tk.W, tk.E))

    ttk.Button(frame, text="Add Habit",
command=self.add_habit).grid(
    row=2, column=0, sticky=(tk.W, tk.E))
    ttk.Button(frame, text="Edit Habit",
command=self.edit_habit).grid(
    row=2, column=1, sticky=(tk.W, tk.E))

    ttk.Label(frame, text="Search Habit:").grid(
        row=0, column=2, sticky=tk.W)
    ttk.Entry(frame, textvariable=self.search_var).grid(
        row=0, column=3, sticky=(tk.W, tk.E))
    ttk.Button(frame, text="Search",
command=self.search_habit).grid(
        row=0, column=4, sticky=(tk.W, tk.E))

    ttk.Label(frame, text="Filter:").grid(row=1, column=2,
sticky=tk.W)
    ttk.Combobox(frame, textvariable=self.filter_var, values=[
        "All", "Completed", "Not Completed"]).grid(row=1,
column=3, sticky=(tk.W, tk.E))
    ttk.Button(frame, text="Apply Filter",
command=self.apply_filter).grid(
        row=1, column=4, sticky=(tk.W, tk.E))

self.habits_listbox = tk.Listbox(
    frame, height=10, width=50, font=("Segoe UI", 10))
self.habits_listbox.grid(
    row=3, column=0, columnspan=4, sticky=(tk.W, tk.E, tk.N,
tk.S))
self.habits_listbox.bind('<<ListboxSelect>>', self.on_habit_select)

self.scrollbar = ttk.Scrollbar(
    frame, orient="vertical", command=self.habits_listbox.yview)
self.habits_listbox.configure(yscrollcommand=self.scrollbar.set)
```

```
self.scrollbar.grid(row=3, column=4, sticky=(tk.N, tk.S))

    ttk.Button(frame, text="Mark as Completed",
command=self.mark_as_completed).grid(
        row=4, column=0, columnspan=2, sticky=(tk.W, tk.E))
    ttk.Button(frame, text="Unmark as Completed",
command=self.unmark_as_completed).grid(
        row=4, column=2, columnspan=2, sticky=(tk.W, tk.E))

    ttk.Label(frame, text="Start Date:").grid(row=5, column=0,
sticky=tk.W)
    DateEntry(frame, textvariable=self.start_date_var,
              date_pattern='yyyy-mm-dd').grid(row=5, column=1,
sticky=(tk.W, tk.E))

    ttk.Label(frame, text="End Date:").grid(row=6, column=0,
sticky=tk.W)
    DateEntry(frame, textvariable=self.end_date_var,
              date_pattern='yyyy-mm-dd').grid(row=6, column=1,
sticky=(tk.W, tk.E))

    ttk.Button(frame, text="Visualize Progress",
command=self.visualize_progress).grid(
        row=7, column=0, columnspan=4, sticky=(tk.W, tk.E))

    ttk.Button(frame, text="Export Data",
command=self.export_data).grid(
        row=8, column=0, columnspan=4, sticky=(tk.W, tk.E))

    ttk.Button(frame, text="Generate Report",
command=self.generate_report).grid(
        row=9, column=0, columnspan=4, sticky=(tk.W, tk.E))

    ttk.Button(frame, text="Backup Data",
command=self.backup_data).grid(
        row=10, column=0, columnspan=2, sticky=(tk.W, tk.E))
```

```
        ttk.Button(frame, text="Restore Data",
command=self.restore_data).grid(
    row=10, column=2, columnspan=2, sticky=(tk.W, tk.E))

        ttk.Button(frame, text="Set Reminder",
command=self.setReminder).grid(
    row=11, column=0, columnspan=4, sticky=(tk.W, tk.E))

for i in range(12):
    frame.rowconfigure(i, weight=1)
for i in range(5):
    frame.columnconfigure(i, weight=1)

self.load_habits()

def add_habit(self):
    name = self.habit_name_var.get()
    date = self.date_var.get()
    with self.conn:
        self.conn.execute(
            "INSERT INTO habits (name, date, completed) VALUES
(?, ?, ?)", (name, date, False))
    self.load_habits()
    self.clear_form()

def load_habits(self):
    self.habits_listbox.delete(0, tk.END)
    cursor = self.conn.execute(
        "SELECT id, name, date, completed FROM habits")
    for row in cursor:
        self.habits_listbox.insert(
            tk.END, f"{row[1]} ({row[2]}) - {'Completed' if row[3]
else 'Not Completed'}")

def on_habit_select(self, event):
    selected = self.habits_listbox.curselection()
    if selected:
```

```

habit = self.habits_listbox.get(selected[0])
habit_name = habit.split(" (")[-1]
habit_date = habit.split("(")[-1].split(")")[0]
self.habit_name_var.set(habit_name)
self.date_var.set(habit_date)
cursor = self.conn.execute(
    "SELECT id FROM habits WHERE name = ? AND date = ?",
    (habit_name, habit_date))
self.selected_habit_id = cursor.fetchone()[0]

def mark_as_completed(self):
    selected = self.habits_listbox.curselection()
    if selected:
        habit = self.habits_listbox.get(selected[0])
        habit_name = habit.split(" (")[-1]
        habit_date = habit.split("(")[-1].split(")")[0]
        with self.conn:
            self.conn.execute(
                "UPDATE habits SET completed = ? WHERE name = ?"
                "AND date = ?", (True, habit_name, habit_date))
        self.load_habits()

def unmark_as_completed(self):
    selected = self.habits_listbox.curselection()
    if selected:
        habit = self.habits_listbox.get(selected[0])
        habit_name = habit.split(" (")[-1]
        habit_date = habit.split("(")[-1].split(")")[0]
        with self.conn:
            self.conn.execute(
                "UPDATE habits SET completed = ? WHERE name = ?"
                "AND date = ?", (False, habit_name, habit_date))
        self.load_habits()

def edit_habit(self):
    if self.selected_habit_id:
        name = self.habit_name_var.get()

```

```

date = self.date_var.get()
with self.conn:
    self.conn.execute(
        "UPDATE habits SET name = ?, date = ? WHERE id = ?",
        (name, date, self.selected_habit_id))
    self.load_habits()
    self.clear_form()

def clear_form(self):
    self.habit_name_var.set("")
    self.date_var.set(datetime.date.today().strftime("%Y-%m-%d"))
    self.selected_habit_id = None

def visualize_progress(self):
    start_date = self.start_date_var.get()
    end_date = self.end_date_var.get()
    cursor = self.conn.execute(
        "SELECT name, date, completed FROM habits WHERE date BETWEEN ? AND ?",
        (start_date, end_date))
    data = {}
    for row in cursor:
        name, date, completed = row
        if name not in data:
            data[name] = {"dates": [], "completed": []}
        data[name]["dates"].append(date)
        data[name]["completed"].append(completed)

    plt.figure(figsize=(10, 6))
    for habit, values in data.items():
        plt.plot(values["dates"], values["completed"], label=habit)
    plt.xlabel("Date")
    plt.ylabel("Completed")
    plt.title("Habit Progress")
    plt.legend()
    plt.grid(True)
    plt.show()

```

```

def export_data(self):
    cursor = self.conn.execute("SELECT name, date, completed
FROM habits")
    with open("habits_data.csv", "w") as f:
        f.write("Name,Date,Completed\n")
        for row in cursor:
            f.write(f"{row[0]},{row[1]},{row[2]}\n")
    messagebox.showinfo("Export Completed",
                        "Habits data has been exported to
habits_data.csv")

def generate_report(self):
    start_date = self.start_date_var.get()
    end_date = self.end_date_var.get()
    cursor = self.conn.execute(
        "SELECT name, date, completed FROM habits WHERE date
BETWEEN ? AND ?", (start_date, end_date))
    report_data = {}
    for row in cursor:
        name, date, completed = row
        if name not in report_data:
            report_data[name] = {"completed": 0, "not_completed": 0}
        if completed:
            report_data[name]["completed"] += 1
        else:
            report_data[name]["not_completed"] += 1

    report = tk.Toplevel(self.root)
    report.title("Habit Report")
    text = tk.Text(report)
    for habit, data in report_data.items():
        text.insert(
            tk.END, f"Habit: {habit}\nCompleted:
{data['completed']}\\nNot Completed: {data['not_completed']}\\n\\n")
    text.pack()

def backup_data(self):

```

```
with open("habits_backup.sql", "w") as f:
    for line in self.conn.iterdump():
        f.write(f'{line}\n')
messagebox.showinfo("Backup Completed",
                    "Habits data has been backed up successfully!")

def restore_data(self):
    with open("habits_backup.sql", "r") as f:
        sql_script = f.read()
    with self.conn:
        self.conn.executescript(sql_script)
    self.load_habits()
    messagebox.showinfo("Restore Completed",
                        "Habits data has been restored successfully!")

def search_habit(self):
    search_term = self.search_var.get()
    self.habits_listbox.delete(0, tk.END)
    cursor = self.conn.execute(
        "SELECT id, name, date, completed FROM habits WHERE
name LIKE ?",
        ('%' + search_term + '%',))
    for row in cursor:
        self.habits_listbox.insert(
            tk.END, f"{row[1]} ({row[2]}) - {'Completed' if row[3]
else 'Not Completed'}")

def apply_filter(self):
    filter_option = self.filter_var.get()
    self.habits_listbox.delete(0, tk.END)
    if filter_option == "All":
        cursor = self.conn.execute(
            "SELECT id, name, date, completed FROM habits")
    elif filter_option == "Completed":
        cursor = self.conn.execute(
            "SELECT id, name, date, completed FROM habits WHERE
completed = 1")
    else:
```

```
cursor = self.conn.execute(
    "SELECT id, name, date, completed FROM habits WHERE
completed = 0")
for row in cursor:
    self.habits_listbox.insert(
        tk.END, f"{row[1]} ({row[2]}) - {'Completed' if row[3]
else 'Not Completed'}")

def setReminder(self):
    habit = self.habit_name_var.get()
    reminder_time = self.date_var.get()
    reminder_datetime = datetime.datetime.strptime(
        reminder_time, "%Y-%m-%d")
    delay = (reminder_datetime -
datetime.datetime.now()).total_seconds()
    Timer(delay, self.showReminder, args=[habit]).start()

def showReminder(self, habit):
    messagebox.showinfo("Reminder", f"Time to work on your habit:
{habit}")

def on_closing(self):
    self.conn.close()
    self.root.destroy()

if __name__ == "__main__":
    root = tk.Tk()
    root.geometry("600x550+450+50")
    app = HabitTrackerApp(root)
    root.protocol("WM_DELETE_WINDOW", app.on_closing)
    root.mainloop()
```

How to Install Requirements and Run the Habit Tracker App with VSCode

1. Setting Up Your Environment

Before you start running the Habit Tracker App, you'll need to install **Python** and some necessary libraries. Here's a step-by-step guide to get everything up and running:

1.1 Install Python

- **Download Python** Go to the [official Python website](#) and download the latest version of **Python**. Make sure to check the box that says "Add **Python** to PATH" during installation.
- **Verify Installation:** Open a terminal or command prompt and type **Python --version** to verify that **Python** is installed correctly.

1.2 Install Required Libraries

The Habit Tracker App uses the following libraries:

- tkinter: For creating the GUI.
- sqlite3: For database management (comes with **Python**).
- matplotlib: For plotting graphs.
- tkcalendar: For calendar widgets.
- pandas (optional, for advanced data handling).

You can install these libraries using [pip](#). Open a terminal or command prompt and run:

bash:

Code:

[pip](#) install matplotlib tkcalendar pandas

1.3 Set Up the Project in VSCode

1. **Open VSCode:** Launch Visual Studio Code (VSCode).
2. **Create a New Project Folder:** Create a folder for your project. You can name it HabitTrackerApp or something similar.
3. **Open the Project Folder in VSCode:**
 - Click on File > Open Folder... and select the folder you created.
4. **Create a New Python File:**
 - Click on File > New File, name it habit_tracker.py, and paste the code below into this file.

1.4 Install the Python Extension for VSCode

If you haven't already, install the [Python](#) extension for VSCode:

- Open the Extensions view by clicking on the Extensions icon in the Activity Bar on the side of the window.

- Search for "**Python**" and install the extension provided by Microsoft.

1.5 Run the Code

1. Open the Integrated Terminal:

- You can open the terminal by selecting Terminal > New Terminal from the menu.

2. Run the **Python** File:

- In the terminal, ensure you're in the project folder (where habit_tracker.py is located).
- Type **Python** habit_tracker.py and press Enter to run the script.

Deep Dive into the Code

Here's a detailed explanation of the Habit Tracker App code:

2.1 Imports and Initialization

Python

Code:

```
import tkinter as tk
from tkinter import ttk
import sqlite3
import datetime
import matplotlib.pyplot as plt
from tkcalendar import DateEntry
from tkinter import messagebox
from threading import Timer
```

- **tkinter**: Standard **Python** library for creating graphical user interfaces.
- **ttk**: Provides themed widgets.
- **sqlite3**: For SQLite database operations.
- **datetime**: For handling date and time.
- **matplotlib.pyplot**: For creating plots and visualizations.
- **tkcalendar**: Adds calendar widgets to tkinter.
- **messagebox**: For displaying dialog boxes.
- **Timer**: For scheduling reminders.

2.2 Class Definition: HabitTrackerApp

This is the main class for the application.

Python

Code:

```
class HabitTrackerApp:
```

```
    def __init__(self, root):
```

```
        self.root = root
```

```
        self.root.title("Habit Tracker")
```

```
        self.conn = sqlite3.connect("habits.db")
```

```
        self.create_table()
```

```
        self.setup_ui()
```

- **__init__ Method:** Initializes the application, sets up the database connection, creates the database table, and sets up the user interface (UI).

2.3 Database Setup

Python

Code:

```
def create_table(self):
```

```
    with self.conn:
```

```
        self.conn.execute("""
```

```
            CREATE TABLE IF NOT EXISTS habits (
```

```
                id INTEGER PRIMARY KEY,
```

```
                name TEXT NOT NULL,
```

```
                date TEXT NOT NULL,
```

```
    completed BOOLEAN NOT NULL  
)  
""")
```

- **create_table Method:** Creates a table named habits in the SQLite database if it doesn't exist. The table stores habit data with fields for ID, name, date, and completion status.

2.4 UI Setup

Python

Code:

```
def setup_ui(self):
    style = ttk.Style(self.root)
    style.theme_use('clam') # Use a modern theme
    style.configure("TButton", font=("Segoe UI", 10))
    style.configure(" TLabel", font=("Segoe UI", 10))
    style.configure("TEntry", font=("Segoe UI", 10))
```

- **setup_ui Method:** Sets up the GUI using tkinter and ttk widgets, applying styles and creating layout with labels, entry fields, buttons, listboxes, and date pickers.

2.5 Add, Edit, and Load Habits

Python

Code:

```
def add_habit(self):
    name = self.habit_name_var.get()
    date = self.date_var.get()
    with self.conn:
        self.conn.execute(
            "INSERT INTO habits (name, date, completed)
            VALUES (?, ?, ?)", (name, date, False))
    self.load_habits()
```

```
self.clear_form()
```

- **add_habit Method:** Adds a new habit to the database and reloads the habit list.

Python

Code:

```
def load_habits(self):  
    self.habits_listbox.delete(0, tk.END)  
    cursor = self.conn.execute(  
        "SELECT id, name, date, completed FROM  
        habits")  
    for row in cursor:  
        self.habits_listbox.insert(  
            tk.END, f"{row[1]} ({row[2]}) - {'Completed'  
if row[3] else 'Not Completed'}")  
    • load_habits Method: Loads habits from the  
    database and displays them in the listbox.
```

2.6 Handling Habit Selection

Python

Code:

```
def on_habit_select(self, event):  
    selected = self.habits_listbox.curselection()  
    if selected:  
        habit = self.habits_listbox.get(selected[0])  
        habit_name = habit.split(" (")[0]
```

```
habit_date = habit.split("(")[1].split(")")[0]
self.habit_name_var.set(habit_name)
self.date_var.set(habit_date)
cursor = self.conn.execute(
    "SELECT id FROM habits WHERE name = ?  

AND date = ?", (habit_name, habit_date))
self.selected_habit_id = cursor.fetchone()[0]
```

- **on_habit_select Method:** Updates the form with the selected habit's details when a habit is selected from the listbox.

2.7 Marking Habits as Completed

Python

Code:

```
def mark_as_completed(self):
    selected = self.habits_listbox.curselection()
    if selected:
        habit = self.habits_listbox.get(selected[0])
        habit_name = habit.split(" ")[0]
        habit_date = habit.split("(")[1].split(")")[0]
        with self.conn:
            self.conn.execute(
                "UPDATE habits SET completed = ?  

WHERE name = ? AND date = ?", (True, habit_name,
habit_date))
```

- **mark_as_completed Method:** Marks the selected habit as completed in the database and updates the listbox.

2.8 Data Visualization

Python

Code:

```
def visualize_progress(self):  
    start_date = self.start_date_var.get()  
    end_date = self.end_date_var.get()  
    cursor = self.conn.execute(  
        "SELECT name, date, completed FROM habits  
        WHERE date BETWEEN ? AND ?", (start_date,  
        end_date))  
    data = {}  
    for row in cursor:  
        name, date, completed = row  
        if name not in data:  
            data[name] = {"dates": [], "completed": []}  
            data[name]["dates"].append(date)  
            data[name]["completed"].append(completed)  
    plt.figure(figsize=(10, 6))  
    for habit, values in data.items():  
        plt.plot(values["dates"], values["completed"],  
        label=habit)
```

```
plt.xlabel("Date")
plt.ylabel("Completed")
plt.title("Habit Progress")
plt.legend()
plt.grid(True)
plt.show()
```

- **visualize_progress Method:** Plots the habit progress over a specified date range using matplotlib.

2.9 Exporting and Backing Up Data

Python

Code:

```
def export_data(self):
    cursor = self.conn.execute("SELECT name, date,
completed FROM habits")
    with open("habits_data.csv", "w") as f:
        f.write("Name,Date,Completed\n")
        for row in cursor:
            f.write(f"{row[0]},{row[1]},{row[2]}\n")
    messagebox.showinfo("Export Completed",
                      "Habits data has been exported to
habits_data.csv")
```

- **export_data Method:** Exports the habit data to a CSV file.

Python

Code:

```
def backup_data(self):
    with open("habits_backup.sql", "w") as f:
        for line in self.conn.iterdump():
            f.write(f'{line}\n')
    messagebox.showinfo("Backup Completed",
```

"Habits data has been backed up successfully!")

- **backup_data Method:** Backs up the database to an SQL file.

2.10 Restoring Data

Python

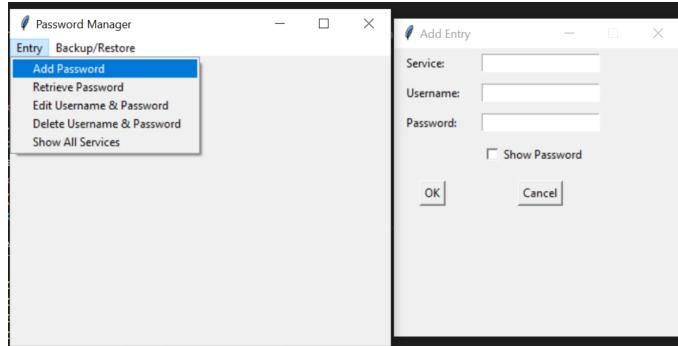
Code:

```
def restore_data(self):
    with open("habits_backup.sql", "r") as f:
        sql = f.read()
    self.conn.executescript(sql)
    messagebox.showinfo("Restore Completed",
                        "Habits data has been restored from the
                        backup!")
    self.load_habits()
    • restore_data Method: Restores the database
      from a backup SQL file.
```

Conclusion

With these steps and explanations, you should be able to install the required libraries, set up your project in VSCode, and run your Habit Tracker App. Feel free to tweak the code and add more features as needed. If you have any specific questions or need further assistance, let me know!

Password Manager



```
import tkinter as tk
from tkinter import simpledialog, messagebox, Toplevel, Scrollbar,
Listbox
import sqlite3
import pyperclip
from cryptography.fernet import Fernet
import os
import re
import shutil

DB_FILE = "password_manager.db" # Define your database file name
or path here

# Load or generate the encryption key
try:
    with open('secret.key', 'rb') as key_file:
        KEY = key_file.read()
except FileNotFoundError:
    KEY = Fernet.generate_key()
    with open('secret.key', 'wb') as key_file:
        key_file.write(KEY)
cipher_suite = Fernet(KEY)

# Database setup

def setup_database():
    conn = sqlite3.connect(DB_FILE)
```

```
c = conn.cursor()
c.execute("""
    CREATE TABLE IF NOT EXISTS passwords (
        id INTEGER PRIMARY KEY,
        service TEXT NOT NULL,
        username TEXT NOT NULL,
        password BLOB NOT NULL,
        UNIQUE(service, username)
    )
""")
conn.commit()
conn.close()
```

```
# Encrypt password
```

```
def encrypt_password(password):
    return cipher_suite.encrypt(password.encode())
```

```
# Decrypt password
```

```
def decrypt_password(encrypted_password):
    try:
        return cipher_suite.decrypt(encrypted_password).decode()
    except Exception as e:
        messagebox.showerror(
            "Decryption Error", f"Failed to decrypt the password.\nError: {str(e)}")
        return ""
```

```
# Add a password entry
```

```
def add_password(service, username, password):
    if not is_strong_password(password):
        messagebox.showwarning(
            "Weak Password", "Password must be at least 8 characters long
            and include letters, numbers, and special characters.")
```

```
    return
conn = sqlite3.connect(DB_FILE)
c = conn.cursor()
encrypted_password = encrypt_password(password)
try:
    c.execute('INSERT INTO passwords (service, username,
password) VALUES (?, ?, ?)',
              (service, username, encrypted_password))
    conn.commit()
    messagebox.showinfo("Success", "Password added successfully")
except sqlite3.IntegrityError:
    messagebox.showerror(
        "Duplicate Entry", "This service and username combination
already exists.")
    conn.close()

# Retrieve a password entry
```

```
def get_password(service, username):
    conn = sqlite3.connect(DB_FILE)
    c = conn.cursor()
    c.execute('SELECT password FROM passwords WHERE service = ?
AND username = ?',
              (service, username))
    result = c.fetchone()
    conn.close()
    if result:
        encrypted_password = result[0]
        decrypted_password = decrypt_password(encrypted_password)
        if decrypted_password:
            show_password_popup(service, username,
decrypted_password)
        else:
            messagebox.showwarning(
                "Not Found", "No entry found for this service and username")
```

```
# Show a popup window with the retrieved password and a copy button
```

```
def show_password_popup(service, username, password):
    popup = tk.Toplevel(root)
    popup.title("Retrieved Password")

    tk.Label(popup, text=f"Service: {service}",
             font=("Arial", 14)).pack(pady=5)
    tk.Label(popup, text=f"Username: {username}",
             font=("Arial", 14)).pack(pady=5)
    tk.Label(popup, text=f>Password: {password}",
             font=("Arial", 14)).pack(pady=5)

def copy_to_clipboard():
    root.clipboard_clear()
    root.clipboard_append(password)
    messagebox.showinfo("Copied", "Password copied to clipboard")

    tk.Button(popup, text="Copy Password",
              command=copy_to_clipboard).pack(pady=10)
    tk.Button(popup, text="Close",
              command=popup.destroy).pack(pady=5)

# Search for passwords

def search_password(service):
    conn = sqlite3.connect(DB_FILE)
    c = conn.cursor()
    c.execute('SELECT service, username FROM passwords WHERE
service LIKE ?',
              ('%' + service + '%'))
    results = c.fetchall()
    conn.close()
    if results:
        search_results = "\n".join(
            [f"Service: {result[0]}, Username: {result[1]}" for result in
            results])
```

```
    messagebox.showinfo("Search Results", search_results)
else:
    messagebox.showwarning("Not Found", "No entries found for this
search")

# Edit an existing password entry

def edit_password(service, username):
    conn = sqlite3.connect(DB_FILE)
    c = conn.cursor()
    c.execute('SELECT id, password FROM passwords WHERE service
= ? AND username = ?',
              (service, username))
    result = c.fetchone()
    if result:
        id, encrypted_password = result
        new_username = simpledialog.askstring(
            "Username", "Enter the new username:",
            initialvalue=username, parent=root)
        new_password = simpledialog.askstring(
            "Password", "Enter the new password:", parent=root)
        if new_username and new_password:
            if not is_strong_password(new_password):
                messagebox.showwarning(
                    "Weak Password", "Password must be at least 8
characters long and include letters, numbers, and special characters.")
                return
            new_encrypted_password = encrypt_password(new_password)
            c.execute('UPDATE passwords SET username = ?, password =
? WHERE id = ?',
                      (new_username, new_encrypted_password, id))
            conn.commit()
            messagebox.showinfo("Success", "Password updated
successfully")
        else:
            messagebox.showwarning("Input Error", "All fields are
required")
```

```
else:
    messagebox.showwarning(
        "Not Found", "No entry found for this service and username")
    conn.close()

# Edit an existing service name

def edit_service(old_service_name, username):
    conn = sqlite3.connect(DB_FILE)
    c = conn.cursor()
    new_service_name = simpledialog.askstring(
        "Edit Service", "Enter the new service name:",
        initialvalue=old_service_name, parent=root)
    if new_service_name:
        c.execute('UPDATE passwords SET service = ? WHERE service
        = ? AND username = ?',
                  (new_service_name, old_service_name, username))
        conn.commit()
        messagebox.showinfo("Success", "Service name updated
        successfully")
    else:
        messagebox.showwarning("Input Error", "Service name is
        required")
    conn.close()

# Delete a password entry

def delete_password(service, username):
    conn = sqlite3.connect(DB_FILE)
    c = conn.cursor()
    c.execute('DELETE FROM passwords WHERE service = ? AND
    username = ?',
              (service, username))
    conn.commit()
    conn.close()
    messagebox.showinfo("Success", "Password deleted successfully")
```

```
# Backup the database
```

```
def backup_database():
    try:
        shutil.copy(DB_FILE, 'password_manager_backup.db')
        messagebox.showinfo("Backup", "Database backup created
successfully")
    except Exception as e:
        messagebox.showerror(
            "Backup Error", f"Failed to create backup.\nError: {str(e)}")
```

```
# Restore the database from backup
```

```
def restore_database():
    if os.path.exists('password_manager_backup.db'):
        try:
            shutil.copy('password_manager_backup.db', DB_FILE)
            messagebox.showinfo("Restore", "Database restored from
backup")
        except Exception as e:
            messagebox.showerror(
                "Restore Error", f"Failed to restore backup.\nError:
{str(e)}")
        else:
            messagebox.showwarning("Restore Error", "Backup file not
found")
```

```
# Validate password strength
```

```
def is_strong_password(password):
    return (len(password) >= 8 and
            re.search(r'[A-Za-z]', password) and
            re.search(r'[0-9]', password) and
            re.search(r'[@#$%^&*]', password))
```

```
# Show all services

def show_all_services():
    def on_select(event):
        selected = listbox.get(listbox.curselection())
        service, username = selected.split(" | ")
        get_password(service, username)

    top = Toplevel()
    top.title("All Services")
    top.geometry("400x300")

    frame = tk.Frame(top)
    frame.pack(pady=10, padx=10, fill=tk.BOTH, expand=True)

    scrollbar = Scrollbar(frame)
    scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

    listbox = Listbox(frame, yscrollcommand=scrollbar.set, width=50)
    listbox.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
    listbox.bind("<Double-1>", on_select)

    scrollbar.config(command=listbox.yview)

    conn = sqlite3.connect(DB_FILE)
    cursor = conn.cursor()
    cursor.execute("SELECT service, username FROM passwords")
    rows = cursor.fetchall()
    conn.close()

    for row in rows:
        listbox.insert(tk.END, f"{row[0]} | {row[1]}")

# Custom Dialog for User Input

class CustomDialog(tk.Toplevel):
```

```
def __init__(self, parent, title, labels):
    super().__init__(parent)
    self.parent = parent
    self.title(title)
    self.geometry("300x300")
    self.resizable(False, False)

    # Handle the window close button
    self.protocol("WM_DELETE_WINDOW", self.cancel)

    # Position the dialog window below the main window
    x = self.parent.winfo_x() + (self.parent.winfo_width() // 2) - 150
    y = self.parent.winfo_y() + (self.parent.winfo_height() // 2) - 150
    self.geometry(f"300x300+{x}+{y}")

    self.labels = labels
    self.entries = {}

    # Create label and entry fields
    for i, (text, default_value) in enumerate(labels):
        tk.Label(self, text=text).grid(
            row=i, column=0, sticky=tk.W, padx=10, pady=5)
        show_password = None
        if 'Password' in text:
            show_password = '*'
        entry = tk.Entry(self, show=show_password)
        entry.insert(0, default_value)
        entry.grid(row=i, column=1, padx=10, pady=5)
        self.entries[text] = entry

    # Add Show Password checkbox if there's a password field
    if any('Password' in text for text, _ in labels):
        self.show_password_var = tk.BooleanVar()
        self.show_password_check = tk.Checkbutton(
            self, text="Show Password",
            variable=self.show_password_var,
            command=self.toggle_password_visibility)
```

```

        self.show_password_check.grid(
            row=len(labels), column=1, sticky=tk.W, padx=10, pady=5)

    # Create OK and Cancel buttons
    tk.Button(self, text="OK", command=self.ok).grid(
        row=len(labels) + 1, column=0, padx=10, pady=10)
    tk.Button(self, text="Cancel", command=self.cancel).grid(
        row=len(labels) + 1, column=1, padx=10, pady=10)

def toggle_password_visibility(self):
    show = self.show_password_var.get()
    print(f"Show Password: {show}") # Debugging line
    for label, entry in self.entries.items():
        if 'Password' in label:
            entry.config(show=" if show else '*' )

def ok(self):
    self.result = {text: entry.get()
                  for text, entry in self.entries.items()}
    self.destroy()

def cancel(self):
    self.result = None
    self.destroy()

def add_entry():
    labels = [("Service:", ""), ("Username:", ""), ("Password:", "")]
    dialog = CustomDialog(root, "Add Entry", labels)
    root.wait_window(dialog)
    result = dialog.result
    if result:
        service = result["Service:"]
        username = result["Username:"]
        password = result["Password:"]
        add_password(service, username, password)

```

```
def retrieve_entry():
    labels = [("Service:", ""), ("Username:", "")]
    dialog = CustomDialog(root, "Retrieve Entry", labels)
    root.wait_window(dialog)
    result = dialog.result
    if result:
        service = result["Service:"]
        username = result["Username:"]
        get_password(service, username)
```

```
def edit_entry():
    labels = [("Service:", ""), ("Username:", "")]
    dialog = CustomDialog(root, "Edit Entry", labels)
    root.wait_window(dialog)
    result = dialog.result
    if result:
        service = result["Service:"]
        username = result["Username:"]
        edit_password(service, username)
```

```
def delete_entry():
    labels = [("Service:", ""), ("Username:", "")]
    dialog = CustomDialog(root, "Delete Entry", labels)
    root.wait_window(dialog)
    result = dialog.result
    if result:
        service = result["Service:"]
        username = result["Username:"]
        delete_password(service, username)
```

```
def show_service_list():
    show_all_services()
```

```
def backup():
    backup_database()
```

```
def restore():
    restore_database()

root = tk.Tk()
root.title("Password Manager")
root.geometry("400x300")

# Create menu
menu = tk.Menu(root)
root.config(menu=menu)
entry_menu = tk.Menu(menu, tearoff=0)
menu.add_cascade(label="Entry", menu=entry_menu)
entry_menu.add_command(label="Add Password",
command=add_entry)
entry_menu.add_command(label="Retrieve Password",
command=retrieve_entry)
entry_menu.add_command(label="Edit Username & Password",
command=edit_entry)
entry_menu.add_command(label="Delete Username & Password",
command=delete_entry)
entry_menu.add_command(label="Show All Services",
command=show_service_list)

backup_menu = tk.Menu(menu, tearoff=0)
menu.add_cascade(label="Backup/Restore", menu=backup_menu)
backup_menu.add_command(label="Backup Database",
command=backup)
backup_menu.add_command(label="Restore Database",
command=restore)

setup_database()
root.mainloop()
```

Password Manager Application Guide

1. Installation and Setup

1.1 Prerequisites

Before you start, make sure you have the following installed:

- **Python 3.x:** The latest version of **Python** can be downloaded from [Python.org](https://www.python.org).
- **VS Code:** You can download it from code.visualstudio.com.

1.2 Install Required Packages

1. **Open VS Code** and open a new terminal (you can use the integrated terminal in VS Code by navigating to View > Terminal).
2. **Create a virtual environment** (optional but recommended):

bash:

Code:

Python -m venv venv

Activate the virtual environment:

- On Windows:

bash:

Code:

.\venv\Scripts\activate

- On macOS/Linux:

bash:

Code:

source venv/bin/activate

3. **Install the necessary packages** using pip:

bash:

Code:

pip install tkinter pyperclip cryptography

4. **Create the database file** and the encryption key:

- Run the following **Python** script to create password_manager.db and secret.key:

Python

Code:

```
import sqlite3
from cryptography.fernet import Fernet

DB_FILE = "password_manager.db"

# Load or generate the encryption key
try:
    with open('secret.key', 'rb') as key_file:
        KEY = key_file.read()
except FileNotFoundError:
    KEY = Fernet.generate_key()
    with open('secret.key', 'wb') as key_file:
        key_file.write(KEY)

# Setup database
conn = sqlite3.connect(DB_FILE)
c = conn.cursor()
c.execute("""
    CREATE TABLE IF NOT EXISTS
    passwords (
        id INTEGER PRIMARY KEY,
        service TEXT NOT NULL,

```

```
username TEXT NOT NULL,  
password BLOB NOT NULL,  
UNIQUE(service, username)  
)  
"")  
conn.commit()  
conn.close()
```

2. Code Deep Dive

The code provided is a simple password manager with encryption, backup, and restore functionalities. Let's break down the main components:

2.1 Imports and Key Setup

Python

Code:

```
import tkinter as tk
from tkinter import simpledialog, messagebox,
Toplevel, Scrollbar, Listbox
import sqlite3
import pyperclip
from cryptography.fernet import Fernet
import os
import re
import shutil

DB_FILE = "password_manager.db" # Define your
database file name or path here

# Load or generate the encryption key
try:
    with open('secret.key', 'rb') as key_file:
        KEY = key_file.read()
except FileNotFoundError:
    KEY = Fernet.generate_key()
```

```
with open('secret.key', 'wb') as key_file:  
    key_file.write(KEY)  
cipher_suite = Fernet(KEY)
```

- **tkinter** is used for GUI development.
- **sqlite3** is used for database interactions.
- **pyperclip** handles clipboard operations.
- **cryptography** handles encryption and decryption of passwords.
- **os, re, and shutil** are used for file and string operations.

2.2 Database Setup

Python
Code:

```
def setup_database():  
    conn = sqlite3.connect(DB_FILE)  
    c = conn.cursor()  
    c.execute("""  
        CREATE TABLE IF NOT EXISTS passwords (  
            id INTEGER PRIMARY KEY,  
            service TEXT NOT NULL,  
            username TEXT NOT NULL,  
            password BLOB NOT NULL,  
            UNIQUE(service, username)  
    )  
    """)
```

```
conn.commit()  
conn.close()
```

- This function creates a SQLite database with a table to store encrypted passwords.

2.3 Encryption and Decryption

Python

Code:

```
def encrypt_password(password):  
    return cipher_suite.encrypt(password.encode())  
  
def decrypt_password(encrypted_password):  
    try:  
        return  
        cipher_suite.decrypt(encrypted_password).decode()  
    except Exception as e:  
        messagebox.showerror("Decryption Error",  
f"Failed to decrypt the password.\nError: {str(e)}")  
    return ""
```

- **encrypt_password**: Encrypts passwords using the Fernet encryption scheme.
- **decrypt_password**: Decrypts passwords and handles decryption errors.

2.4 Adding, Retrieving, Editing, and Deleting Passwords

Python

Code:

```
def add_password(service, username, password):
    if not is_strong_password(password):
        messagebox.showwarning("Weak Password",
        "Password must be at least 8 characters long and
        include letters, numbers, and special characters.")
        return
    conn = sqlite3.connect(DB_FILE)
    c = conn.cursor()
    encrypted_password =
    encrypt_password(password)
    try:
        c.execute('INSERT INTO passwords (service,
        username, password) VALUES (?, ?, ?)',
        (service, username,
        encrypted_password))
        conn.commit()
        messagebox.showinfo("Success", "Password
        added successfully")
    except sqlite3.IntegrityError:
        messagebox.showerror("Duplicate Entry", "This
        service and username combination already exists.")
    conn.close()
    • add_password: Adds a new password entry
      after checking for password strength.
```

- **get_password**: Retrieves and decrypts a password entry.
- **edit_password**: Allows modification of an existing password entry.
- **delete_password**: Deletes an entry from the database.

2.5 Backup and Restore

Python

Code:

```
def backup_database():
    try:
        shutil.copy(DB_FILE,
'password_manager_backup.db')
        messagebox.showinfo("Backup", "Database
backup created successfully")
    except Exception as e:
        messagebox.showerror("Backup Error", f"Failed
to create backup.\nError: {str(e)}")

def restore_database():
    if os.path.exists('password_manager_backup.db'):
        try:
            shutil.copy('password_manager_backup.db',
DB_FILE)
            messagebox.showinfo("Restore", "Database
restored from backup")
```

```
except Exception as e:  
    messagebox.showerror("Restore Error",  
    f"Failed to restore backup.\nError: {str(e)}")  
else:  
    messagebox.showwarning("Restore Error",  
    "Backup file not found")
```

- **backup_database:** Creates a backup of the current database file.
- **restore_database:** Restores the database from a backup file.

2.6 Password Strength Validation

Python

Code:

```
def is_strong_password(password):  
    return (len(password) >= 8 and  
            re.search(r'[A-Za-z]', password) and  
            re.search(r'[0-9]', password) and  
            re.search(r'[@#$%^&*]', password))
```

- **is_strong_password:** Checks if the password meets the criteria of being at least 8 characters long and includes letters, numbers, and special characters.

2.7 GUI Components

- **CustomDialog:** A custom dialog class for user input, allowing users to add, retrieve, edit, or

delete password entries.

- **show_password_popup:** Displays the retrieved password in a popup window with an option to copy it to the clipboard.
- **show_all_services:** Shows a list of all services and usernames in a separate window.

2.8 Main Application

Python

Code:

```
root = tk.Tk()
root.title("Password Manager")
root.geometry("400x300")

# Create menu
menu = tk.Menu(root)
root.config(menu=menu)
entry_menu = tk.Menu(menu, tearoff=0)
menu.add_cascade(label="Entry", menu=entry_menu)
entry_menu.add_command(label="Add Password",
command=add_entry)
entry_menu.add_command(label="Retrieve
Password", command=retrieve_entry)
entry_menu.add_command(label="Edit Username &
Password", command=edit_entry)
entry_menu.add_command(label="Delete Username &
Password", command=delete_entry)
entry_menu.add_command(label="Show All
Services", command=show_service_list)

backup_menu = tk.Menu(menu, tearoff=0)
menu.add_cascade(label="Backup/Restore",
menu=backup_menu)
```

```
backup_menu.add_command(label="Backup Database", command=backup)
```

```
backup_menu.add_command(label="Restore Database", command=restore)
```

```
setup_database()
```

```
root.mainloop()
```

- **root**: The main application window.
- **menu**: Contains menu options for managing passwords and database backup/restore.

With this guide, you should be able to set up the password manager application, understand the key components of the code, and run it efficiently