import tkinter as tk

import threading

from typing import List, Optional

import pandas as pd

from file\_handler import FileHandler

from comparison import DataComparer

from gui\_handler import GUIHandler

from query\_handler import QueryHandler

from voice\_assistant import VoiceAssistant

class App:

    def \_\_init\_\_(self):

        self.dataframes: List[pd.DataFrame] = []

        self.file\_paths: List[str] = []

        self.comparison\_summary: str = ""

        self.voice\_assistant: Optional[VoiceAssistant] = None

        self.microphone\_available: bool = self.\_check\_microphone\_availability()

        self.stop\_event: threading.Event = threading.Event()

        self.voice\_thread: Optional[threading.Thread] = None

        # Initialize QueryHandler once

        self.query\_handler = QueryHandler()

    def \_check\_microphone\_availability(self) -> bool:

        """Check if a microphone is available and initialize VoiceAssistant."""

        try:

            self.voice\_assistant = VoiceAssistant(language="en-US")

            return True

        except OSError:

            return False

    def load\_files(self, file\_paths: List[str]) -> str:

        """Load selected files and handle multiple file processing if necessary."""

        new\_dataframes = []

        for file\_path in file\_paths:

            try:

                handler = FileHandler(file\_path)

                new\_dataframes.append(handler.df)

            except Exception as e:

                return f"Error processing file {file\_path}: {str(e)}"

        self.dataframes.extend(new\_dataframes)

        self.file\_paths.extend(file\_paths)

        self.\_update\_comparison\_summary()

        return "Files successfully loaded."

    def remove\_files(self, file\_paths: List[str]) -> str:

        """Remove specified files and update dataframes."""

        paths\_to\_remove = set(file\_paths)

        self.dataframes = [df for df, path in zip(self.dataframes, self.file\_paths) if path not in paths\_to\_remove]

        self.file\_paths = [path for path in self.file\_paths if path not in paths\_to\_remove]

        self.\_update\_comparison\_summary()

        return "Files successfully removed."

    def \_update\_comparison\_summary(self) -> None:

        """Update comparison summary based on the current dataframes."""

        if len(self.dataframes) >= 2:

            comparer = DataComparer(self.dataframes)

            self.comparison\_summary = comparer.process\_dataframes()

        else:

            self.comparison\_summary = ""

    def handle\_query(self, question: str) -> str:

        """Process the query based on loaded files and return the response."""

        if self.voice\_assistant:

            standardized\_question = self.voice\_assistant.standardize\_language(question)

        else:

            standardized\_question = question

        if standardized\_question:

            # Use the single instance of QueryHandler

            if len(self.dataframes) == 1:  # Single file processing

                response = self.query\_handler.ask\_question(self.dataframes[0], standardized\_question)

            else:  # Multiple file processing

                response = self.query\_handler.ask\_question(self.comparison\_summary, standardized\_question)

            return response

        else:

            return "Sorry, I couldn't understand your question."

    def respond(self, response: str, voice\_response\_enabled: bool) -> Optional[str]:

        """Play voice response if enabled."""

        if voice\_response\_enabled and self.voice\_assistant:

            self.stop\_event.clear()

            if self.voice\_thread and self.voice\_thread.is\_alive():

                self.stop\_event.set()

                self.voice\_assistant.stop()

            self.voice\_thread = threading.Thread(target=self.\_voice\_response, args=(response,))

            self.voice\_thread.start()

        return None

    def \_voice\_response(self, response: str) -> None:

        """Play voice response in a separate thread."""

        if not self.stop\_event.is\_set():

            try:

                self.voice\_assistant.respond(response)

            except Exception as e:

                print(f"Error during voice response: {str(e)}")

    def main(self) -> None:

        """Initialize the GUI and start the main loop."""

        root = tk.Tk()

        GUIHandler(root, self)

        root.mainloop()

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

    app = App()

    app.main()

import tkinter as tk

from tkinter import filedialog, messagebox, scrolledtext

import tkinter.font as tkfont

import threading

import time

from typing import Any, Callable

import re

from spellchecker import SpellChecker

class Tooltip:

    """Class to create tooltips for widgets"""

    def \_\_init\_\_(self, widget: tk.Widget, text: str):

        self.widget = widget

        self.text = text

        self.tooltip: tk.Toplevel = None

        self.widget.bind('<Enter>', self.show\_tooltip)

        self.widget.bind('<Leave>', self.hide\_tooltip)

    def show\_tooltip(self, event: tk.Event = None) -> None:

        if self.tooltip:

            return

        x, y, \_, \_ = self.widget.bbox("insert")

        x += self.widget.winfo\_rootx() + 25

        y += self.widget.winfo\_rooty() + 25

        self.tooltip = tk.Toplevel(self.widget)

        self.tooltip.wm\_overrideredirect(True)

        self.tooltip.wm\_geometry(f"+{x}+{y}")

        label = tk.Label(self.tooltip, text=self.text, background="lightyellow", relief="solid", borderwidth=1)

        label.pack()

    def hide\_tooltip(self, event: tk.Event = None) -> None:

        if self.tooltip:

            self.tooltip.destroy()

            self.tooltip = None

class GUIHandler:

    def \_\_init\_\_(self, root: tk.Tk, app: Any):

        self.root = root

        self.app = app

        self.voice\_response\_enabled = tk.BooleanVar(value=True)

        self.stop\_event = threading.Event()

        self.query\_in\_progress = False  # Flag to prevent duplicate submissions

        self.configure\_root()

        self.create\_widgets()

    def configure\_root(self) -> None:

        """Configure the root window for the dark theme."""

        self.root.title("AI Assistant")

        self.root.configure(bg="#2B2B2B")

        self.root.geometry("1000x600")

        self.root.resizable(True, True)

    def create\_widgets(self) -> None:

        """Create all GUI widgets and layout with a dark theme."""

        self.\_setup\_fonts()

        self.\_create\_query\_frame()

        self.\_create\_files\_frame()

        self.\_create\_result\_text()

    def \_setup\_fonts(self) -> None:

        """Setup fonts for the GUI."""

        self.font = tkfont.Font(family="Helvetica", size=12)

        self.bold\_font = tkfont.Font(family="Helvetica", size=12, weight="bold")

        self.large\_font = tkfont.Font(family="Helvetica", size=14)

    def \_create\_button(self, parent: tk.Widget, text: str, command: Callable, tooltip\_text: str) -> tk.Button:

        """Create a button with a tooltip."""

        button = tk.Button(parent, text=text, command=command, bg="#4A4A4A", fg="white", font=self.large\_font)

        button.pack(side=tk.LEFT, padx=5)

        Tooltip(button, tooltip\_text)

        return button

    def \_create\_query\_frame(self) -> None:

        """Create the query input frame and its widgets."""

        self.query\_frame = tk.Frame(self.root, bg="#2E2E2E")

        self.query\_frame.pack(fill=tk.X, pady=10, padx=10)

        self.\_create\_button(self.query\_frame, "📎", self.select\_files, "Select files to process")

        self.\_create\_button(self.query\_frame, "❌", self.remove\_files, "Remove selected files")

        self.query\_entry = tk.Text(self.query\_frame, width=50, height=5, font=self.large\_font, bg="#3C3C3C", fg="white", insertbackground="white")

        self.query\_entry.pack(side=tk.LEFT, padx=5, fill=tk.X, expand=True)

        self.query\_entry.insert("1.0", "Enter your query here...")  # Placeholder text

        self.query\_entry.bind("<FocusIn>", self.\_clear\_placeholder)  # Clear placeholder on focus

        self.query\_entry.bind("<FocusOut>", self.\_restore\_placeholder)  # Restore placeholder on focus out

        self.query\_entry.bind("<Shift-Return>", self.\_new\_line)  # Move to new line with Shift+Enter

        self.query\_entry.bind("<Return>", self.submit\_query)  # Submit query with Enter

        self.voice\_response\_button = self.\_create\_button(self.query\_frame, "🔊", self.toggle\_voice\_response, "Toggle voice response on/off")

        self.\_create\_button(self.query\_frame, "🎤", self.use\_microphone, "Use microphone for voice input")

    def \_clear\_placeholder(self, event):

        """Clear the placeholder text when the entry gets focus."""

        if self.query\_entry.get("1.0", tk.END).strip() == "Enter your query here...":

            self.query\_entry.delete("1.0", tk.END)

            self.query\_entry.config(fg="white")

    def \_restore\_placeholder(self, event):

        """Restore the placeholder text if the entry is empty."""

        if self.query\_entry.get("1.0", tk.END).strip() == "":

            self.query\_entry.insert("1.0", "Enter your query here...")

            self.query\_entry.config(fg="grey")

    def \_new\_line(self, event):

        """Insert a new line without submitting the query."""

        self.query\_entry.insert(tk.INSERT, "\n")

        return "break"

    def \_create\_files\_frame(self) -> None:

        """Create the frame for displaying attached files."""

        self.files\_frame = tk.Frame(self.root, bg="#2B2B2B")

        self.files\_frame.pack(fill=tk.X, pady=(0, 10), padx=10)

        self.attached\_files\_label = tk.Label(self.files\_frame, text="Attached Files:", font=self.bold\_font, bg="#2B2B2B", fg="white")

        self.attached\_files\_label.pack(anchor=tk.W)

        self.files\_listbox = tk.Listbox(self.files\_frame, font=self.font, bg="#3C3C3C", fg="white", selectbackground="#4A4A4A", height=4)

        self.files\_listbox.pack(fill=tk.X, pady=5)

    def \_create\_result\_text(self) -> None:

        """Create the text area for displaying results."""

        self.result\_text = scrolledtext.ScrolledText(self.root, wrap=tk.WORD, height=15, font=self.font, bg="#1E1E1E", fg="white", insertbackground='white', padx=10, pady=10)

        self.result\_text.pack(pady=10, padx=10, fill=tk.BOTH, expand=True)

    def select\_files(self) -> None:

        """Open a file dialog and load selected files."""

        file\_paths = filedialog.askopenfilenames(

            title="Select files to process",

            filetypes=[("All files", "\*.\*")]

        )

        if file\_paths:

            result = self.app.load\_files(file\_paths)

            self.update\_files\_listbox()

            messagebox.showinfo("Info", result)

    def remove\_files(self) -> None:

        """Remove selected files."""

        selected\_files = [self.files\_listbox.get(i) for i in self.files\_listbox.curselection()]

        if not selected\_files:

            messagebox.showwarning("Warning", "No files selected to remove.")

            return

        result = self.app.remove\_files(selected\_files)

        self.update\_files\_listbox()

        messagebox.showinfo("Info", result)

    def update\_files\_listbox(self) -> None:

        """Update the Listbox to display the attached files."""

        self.files\_listbox.delete(0, tk.END)

        for file\_path in self.app.file\_paths:

            self.files\_listbox.insert(tk.END, file\_path)

    def submit\_query(self, event: tk.Event = None) -> None:

        """Handle the query submission and respond."""

        if self.query\_in\_progress:

            return  # Prevent duplicate queries

        self.query\_in\_progress = True

        question = self.query\_entry.get("1.0", tk.END).strip()

        if not question or question == "Enter your query here...":

            messagebox.showwarning("Warning", "Please enter a question.")

            self.query\_in\_progress = False

            return

        # Check spelling and spacing

        question = self.\_check\_spelling\_and\_spacing(question)

        self.stop\_event.clear()

        response\_thread = threading.Thread(target=self.process\_query, args=(question,))

        response\_thread.start()

    def \_check\_spelling\_and\_spacing(self, text: str) -> str:

        """Check and correct spelling and spacing in the given text."""

        spell = SpellChecker()

        words = text.split()

        corrected\_words = [spell.correction(word) if spell.correction(word) else word for word in words]

        corrected\_text = " ".join(corrected\_words)

        corrected\_text = re.sub(r'\s+', ' ', corrected\_text).strip()

        return corrected\_text

    def process\_query(self, question: str) -> None:

        """Process the query and handle the response."""

        response = self.app.handle\_query(question)

        if not self.stop\_event.is\_set():

            self.stream\_text(response)

            if not self.stop\_event.is\_set():

                time.sleep(2)

                error = self.app.respond(response, self.voice\_response\_enabled.get())

                if error:

                    messagebox.showwarning("Warning", error)

        self.query\_in\_progress = False

    def stream\_text(self, response: str) -> None:

        """Display the response text incrementally."""

        existing\_text = self.result\_text.get("1.0", tk.END).strip()

        # Insert a single separation line before adding the new response

        if existing\_text:

            self.result\_text.insert(tk.END, "\n" + "-"\*40 + "\n\n")  # Single line separator

        # Adding new response

        chunk\_size = 100

        for i in range(0, len(response), chunk\_size):

            if self.stop\_event.is\_set():

                break

            self.result\_text.insert(tk.END, response[i:i+chunk\_size])

            self.result\_text.yview(tk.END)

            time.sleep(0.1)

    def use\_microphone(self) -> None:

        """Handle microphone input."""

        question = self.app.voice\_assistant.get\_query() if self.app.voice\_assistant else ""

        if question:

            self.query\_entry.delete("1.0", tk.END)

            self.query\_entry.insert("1.0", question)

            self.submit\_query()

        else:

            messagebox.showwarning("Warning", "No Microphone detected or input not recognized.")

    def toggle\_voice\_response(self) -> None:

        """Toggle the voice response feature on or off."""

        self.voice\_response\_enabled.set(not self.voice\_response\_enabled.get())

        # Update button text and color based on the state

        if self.voice\_response\_enabled.get():

            new\_text = "🔊"  # Speaker with sound for "On"

            new\_bg = "#4CAF50"  # Green for "On"

        else:

            new\_text = "🔇"  # Muted speaker for "Off"

            new\_bg = "#F44336"  # Red for "Off"

        self.voice\_response\_button.config(text=new\_text, bg=new\_bg)

import pandas as pd

from docx import Document as DocxDocument

import pdfplumber

from sqlalchemy import create\_engine, text

import sqlalchemy

from sqlalchemy.exc import SQLAlchemyError

from langchain\_openai import ChatOpenAI

from typing import Optional, Union

import os

class FileHandler:

    def \_\_init\_\_(self, file\_path: Optional[str] = None, connection\_string: Optional[str] = None):

        self.file\_path: Optional[str] = file\_path

        self.connection\_string: Optional[str] = connection\_string

        self.engine = self.\_create\_engine()

        self.df: pd.DataFrame = pd.DataFrame()

        self.llm: Optional[ChatOpenAI] = self.\_initialize\_llm()

        if file\_path:

            self.df = self.load\_file()

    def \_create\_engine(self) -> Optional[Union[sqlalchemy.engine.base.Engine, sqlalchemy.engine.base.Connection]]:

        """Create an appropriate database engine based on the input."""

        if not self.connection\_string and self.file\_path and self.file\_path.endswith('.sql'):

            return create\_engine('sqlite:///:memory:')

        elif self.connection\_string:

            return create\_engine(self.connection\_string)

        return None

    def \_initialize\_llm(self) -> Optional[ChatOpenAI]:

        """Initialize the language model if an engine is available."""

        return ChatOpenAI(model="gpt-4o", temperature=0.1) if self.engine else None

    def load\_file(self) -> pd.DataFrame:

        """Load file based on its extension and return a DataFrame."""

        if not self.file\_path:

            raise ValueError("No file path provided.")

        file\_extension = os.path.splitext(self.file\_path)[1].lower()

        loaders = {

            '.xlsx': self.\_load\_excel,

            '.xls': self.\_load\_excel,

            '.csv': self.\_load\_csv,

            '.pdf': self.\_load\_pdf,

            '.docx': self.\_load\_docx,

            '.sql': self.\_load\_sql\_file

        }

        loader = loaders.get(file\_extension)

        if loader:

            return loader()

        elif self.file\_path.startswith('sql://'):

            return self.\_load\_sql\_query()

        else:

            raise ValueError(f"Unsupported file type: {self.file\_path}")

    def \_load\_excel(self) -> pd.DataFrame:

        """Read Excel file and return a DataFrame."""

        try:

            return pd.read\_excel(self.file\_path)

        except Exception as e:

            print(f"Error reading Excel file {self.file\_path}: {str(e)}")

            return pd.DataFrame()

    def \_load\_csv(self) -> pd.DataFrame:

        """Read CSV file and return a DataFrame."""

        try:

            return pd.read\_csv(self.file\_path)

        except Exception as e:

            print(f"Error reading CSV file {self.file\_path}: {str(e)}")

            return pd.DataFrame()

    def \_load\_pdf(self) -> pd.DataFrame:

        """Read PDF file and return a DataFrame."""

        try:

            with pdfplumber.open(self.file\_path) as pdf:

                text\_data = [page.extract\_text() for page in pdf.pages]

            return pd.DataFrame({'Content': ["\n".join(text\_data)]})

        except Exception as e:

            print(f"Error reading PDF file {self.file\_path}: {str(e)}")

            return pd.DataFrame()

    def \_load\_docx(self) -> pd.DataFrame:

        """Read DOCX file and return a DataFrame."""

        try:

            doc = DocxDocument(self.file\_path)

            paragraphs = [para.text for para in doc.paragraphs]

            return pd.DataFrame({'Content': ["\n".join(paragraphs)]})

        except Exception as e:

            print(f"Error reading DOCX file {self.file\_path}: {str(e)}")

            return pd.DataFrame()

    def \_load\_sql\_file(self) -> pd.DataFrame:

        """Read SQL file, execute the queries, and return a DataFrame."""

        if not self.engine:

            raise ValueError("No engine available for executing SQL file.")

        try:

            with open(self.file\_path, 'r') as file:

                sql\_commands = file.read().split(';')

            with self.engine.connect() as conn:

                for command in sql\_commands:

                    command = command.strip()

                    if command:

                        conn.execute(text(command))

                tables = conn.execute(text("SELECT name FROM sqlite\_master WHERE type='table';")).fetchall()

                if tables:

                    table\_name = tables[0][0]

                    return pd.read\_sql(f"SELECT \* FROM {table\_name}", conn)

                else:

                    print(f"No tables found after executing SQL file {self.file\_path}.")

                    return pd.DataFrame()

        except (SQLAlchemyError, IOError) as e:

            print(f"Error reading SQL file {self.file\_path}: {str(e)}")

            return pd.DataFrame()

    def \_load\_sql\_query(self) -> pd.DataFrame:

        """Execute a SQL query and return the result as a DataFrame."""

        connection\_string, query = self.\_extract\_sql\_params(self.file\_path)

        try:

            engine = create\_engine(connection\_string)

            return pd.read\_sql(query, engine)

        except Exception as e:

            print(f"Error executing SQL query: {str(e)}")

            return pd.DataFrame()

    def \_extract\_sql\_params(self, file\_path: str) -> tuple[str, str]:

        """Extract SQL connection string and query from the file path."""

        # This is a placeholder implementation. You should customize this based on your needs.

        return 'sqlite:///example.db', 'SELECT \* FROM table\_name'

    def generate\_sql(self, prompt: str) -> str:

        """Generate SQL code using LangChain based on a natural language prompt."""

        if not self.llm:

            raise ValueError("No LLM available for SQL generation.")

        response = self.llm(f"Generate SQL for the following request: {prompt}")

        return response.content.strip()

    def manage\_database(self, management\_prompt: str) -> str:

        """Generate and execute SQL for database management tasks."""

        if not self.engine:

            raise ValueError("No database connection available for management.")

        sql\_code = self.generate\_sql(management\_prompt)

        try:

            with self.engine.connect() as conn:

                conn.execute(text(sql\_code))

            return f"Executed SQL: {sql\_code}"

        except Exception as e:

            return f"Error executing SQL: {str(e)}"

    def execute\_sql\_query(self, query: str) -> pd.DataFrame:

        """Execute a raw SQL query and return the result as a DataFrame."""

        if not self.engine:

            raise ValueError("No database connection available.")

        try:

            return pd.read\_sql(query, self.engine)

        except Exception as e:

            print(f"Error executing SQL query: {str(e)}")

            return pd.DataFrame()

from dotenv import load\_dotenv

import pandas as pd

from langchain\_openai import ChatOpenAI

from typing import Union, List

# Load environment variables (API KEY)

load\_dotenv()

class QueryHandler:

    def \_\_init\_\_(self):

        self.llm: ChatOpenAI = ChatOpenAI(

            model="gpt-4o",

            temperature=0.5,

        )

        self.history: List[str] = []  # List to store the history of questions and answers

    def ask\_question(self, content: Union[pd.DataFrame, str], question: str) -> str:

        """

        Ask a question using LangChain, with context from previous questions and answers.

        Args:

            content (Union[pd.DataFrame, str]): The content to base the answer on.

                Can be either a DataFrame or a string (comparison summary).

            question (str): The question to be answered.

        Returns:

            str: The generated answer to the question.

        """

        prompt = self.\_create\_prompt(content, question)

        answer = self.\_generate\_response(prompt)

        self.\_update\_history(question, answer)

        return self.display\_full\_conversation()

    def \_create\_prompt(self, content: Union[pd.DataFrame, str], question: str) -> str:

        """

        Create a prompt based on the content type, question, and conversation history.

        Args:

            content (Union[pd.DataFrame, str]): The content to base the answer on.

            question (str): The question to be answered.

        Returns:

            str: The generated prompt.

        """

        if isinstance(content, pd.DataFrame):

            content\_str = content.head(250).to\_string(index=False)

            content\_str = f"DataFrame content:\n{content\_str}\n\n"

        else:

            content\_str = f"Comparison Summary:\n{content}\n\n"

        # Combine history into prompt, only including the latest context

        history\_str = "\n\n".join(self.history)

        return f"{history\_str}\n{content\_str}Question: {question}\nAnswer:"

    def \_generate\_response(self, prompt: str) -> str:

        """

        Generate a response using the LLM based on the given prompt.

        Args:

            prompt (str): The prompt to generate a response for.

        Returns:

            str: The generated response.

        """

        response = self.llm.stream([prompt])

        answer = "".join(chunk.content for chunk in response)

        return answer.strip()

    def \_update\_history(self, question: str, answer: str) -> None:

        """

        Update the conversation history with the latest question and answer.

        Args:

            question (str): The question asked.

            answer (str): The answer received.

        """

        self.history.append(f"Question: {question}\n\nAnswer: {answer}\n")

        # Optional: Limit history length to prevent excessive data accumulation

        if len(self.history) > 10:  # Adjust as needed

            self.history.pop(0)  # Remove the oldest entry

    def display\_full\_conversation(self) -> str:

        """

        Retrieve the latest part of the conversation history formatted for display.

        Returns:

            str: The latest part of the conversation history formatted for display.

        """

        if not self.history:

            return ""

        # Find the last occurrence of a question to start from

        latest\_question\_index = len(self.history) - 1

        while latest\_question\_index >= 0 and not self.history[latest\_question\_index].startswith("Question:"):

            latest\_question\_index -= 1

        # Display from the latest question forward

        if latest\_question\_index >= 0:

            return "\n\n".join(self.history[latest\_question\_index:])

        return ""

import speech\_recognition as sr

import pyttsx3

from langchain\_openai import ChatOpenAI

import threading

from typing import Optional

class VoiceAssistant:

    def \_\_init\_\_(self, language: str = "en-US"):

        self.language: str = language

        self.llm: ChatOpenAI = ChatOpenAI(model="gpt-4o", temperature=0.7)

        self.tts\_engine: pyttsx3.Engine = pyttsx3.init()

        self.stop\_event: threading.Event = threading.Event()

        self.microphone: Optional[sr.Microphone] = None

        self.recognizer: Optional[sr.Recognizer] = None

        self.\_initialize\_microphone()

    def \_initialize\_microphone(self) -> None:

        """Initialize microphone if available."""

        try:

            self.microphone = sr.Microphone()

            self.recognizer = sr.Recognizer()

            print("Microphone successfully initialized.")

        except OSError:

            print("No default input device available. Microphone will not be used.")

    def listen(self) -> Optional[str]:

        """Capture voice input from the user and convert it to text."""

        if not self.microphone or not self.recognizer:

            print("Microphone is not available.")

            return None

        with self.microphone as source:

            print("How can I help you?")

            self.speak("How can I help you?")

            audio = self.recognizer.listen(source)

        try:

            return self.recognizer.recognize\_google(audio, language=self.language)

        except sr.UnknownValueError:

            self.speak("Sorry, I could not understand the audio.")

        except sr.RequestError as e:

            self.speak(f"Sorry, there was an error with the speech recognition service: {str(e)}")

        return None

    def standardize\_language(self, text: Optional[str]) -> Optional[str]:

        """Standardize the language of the given text using an LLM."""

        if not text:

            return None

        prompt = f"Please rephrase the following text in a more standard and formal language: '{text}'"

        response = self.llm(prompt)

        return response.content.strip()

    def speak(self, text: Optional[str]) -> None:

        """Convert the text to speech and play it back to the user."""

        if not text:

            return

        def tts() -> None:

            """Thread function for text-to-speech."""

            self.tts\_engine.say(text)

            self.tts\_engine.runAndWait()

        self.stop\_event.clear()

        self.tts\_thread = threading.Thread(target=tts)

        self.tts\_thread.start()

    def get\_query(self) -> Optional[str]:

        """Capture user's voice input, standardize it, and return the standardized text."""

        user\_text = self.listen()

        return self.standardize\_language(user\_text) if user\_text else None

    def respond(self, response\_text: str) -> None:

        """Respond to the user by speaking the text out loud."""

        if response\_text:

            self.speak(response\_text)

import pandas as pd

from typing import List, Union

class DataComparer:

    def \_\_init\_\_(self, dataframes: List[pd.DataFrame]):

        """Initialize the DataComparer with a list of DataFrames."""

        self.dataframes: List[pd.DataFrame] = dataframes

    def process\_dataframes(self) -> Union[pd.DataFrame, str]:

        """Process the DataFrames and return a summary of their differences."""

        if len(self.dataframes) == 1:

            return self.dataframes[0]

        elif len(self.dataframes) < 2:

            raise ValueError("At least two DataFrames are required for comparison.")

        return self.\_compare\_multiple\_dataframes()

    def \_compare\_multiple\_dataframes(self) -> str:

        """Compare multiple DataFrames and return a summary of their differences."""

        comparison\_results: List[str] = []

        num\_dfs = len(self.dataframes)

        for i in range(num\_dfs):

            for j in range(i + 1, num\_dfs):

                df1 = self.dataframes[i]

                df2 = self.dataframes[j]

                comparison\_results.append(f"Comparing DataFrame {i + 1} with DataFrame {j + 1}:")

                comparison\_results.append(self.\_compare\_two\_dataframes(df1, df2))

                comparison\_results.append("\n")  # Add a newline between comparisons

        return "\n".join(comparison\_results)

    def process\_single\_dataframe(self, df: pd.DataFrame) -> str:

        """Generate a summary for a single DataFrame."""

        if 'Content' in df.columns:

            return f"Single file content:\n{df['Content'].iloc[0]}"

        else:

            summary = [

                f"Single DataFrame shape: {df.shape}",

                f"Column names: {list(df.columns)}",

                f"Data preview:\n{df.head().to\_string(index=False)}"

            ]

            return "\n".join(summary)

    def \_compare\_two\_dataframes(self, df1: pd.DataFrame, df2: pd.DataFrame) -> str:

        """Compare two DataFrames and return a summary of their differences."""

        comparison\_results: List[str] = []

        if 'Content' in df1.columns and 'Content' in df2.columns:

            comparison\_results.extend(self.\_compare\_text\_content(df1, df2))

        else:

            comparison\_results.extend(self.\_compare\_tabular\_content(df1, df2))

        return "\n".join(comparison\_results)

    def \_compare\_text\_content(self, df1: pd.DataFrame, df2: pd.DataFrame) -> List[str]:

        """Compare text content of two DataFrames."""

        text1 = df1['Content'].iloc[0]

        text2 = df2['Content'].iloc[0]

        if text1 == text2:

            return ["No differences in text content."]

        else:

            return [

                "Differences found in text content.",

                f"File 1 Content:\n{text1}",

                f"File 2 Content:\n{text2}"

            ]

    def \_compare\_tabular\_content(self, df1: pd.DataFrame, df2: pd.DataFrame) -> List[str]:

        """Compare tabular content of two DataFrames."""

        df1\_text = self.\_convert\_dataframe\_to\_text(df1)

        df2\_text = self.\_convert\_dataframe\_to\_text(df2)

        if df1\_text == df2\_text:

            return ["No differences in tabular content."]

        else:

            return [

                "Differences found in tabular content.",

                f"File 1 Tabular Content:\n{df1\_text}",

                f"File 2 Tabular Content:\n{df2\_text}"

            ]

    @staticmethod

    def \_convert\_dataframe\_to\_text(df: pd.DataFrame) -> str:

        """Convert a DataFrame to a text representation."""

        return df.to\_csv(index=False) if not df.empty else ""