


Sviluppo di un Chatbot Grafico con OpenAI API

Benvenuti a questo corso introduttivo sulla creazione di un'interfaccia grafica per chatbot utilizzando Python e l'API di OpenAI. In questo progetto finale, integreremo le conoscenze acquisite durante il corso per sviluppare un'applicazione desktop completa che permetta all'utente di interagire con i modelli linguistici di OpenAI, in particolare con gpt-4.1-nano.

Utilizzeremo la versione 1.87 della libreria OpenAI e vedremo passo dopo passo come costruire un'applicazione funzionale, dalla scelta del framework grafico fino al deployment finale.

 da **Claudio Marfia**



Scelta del Framework GUI

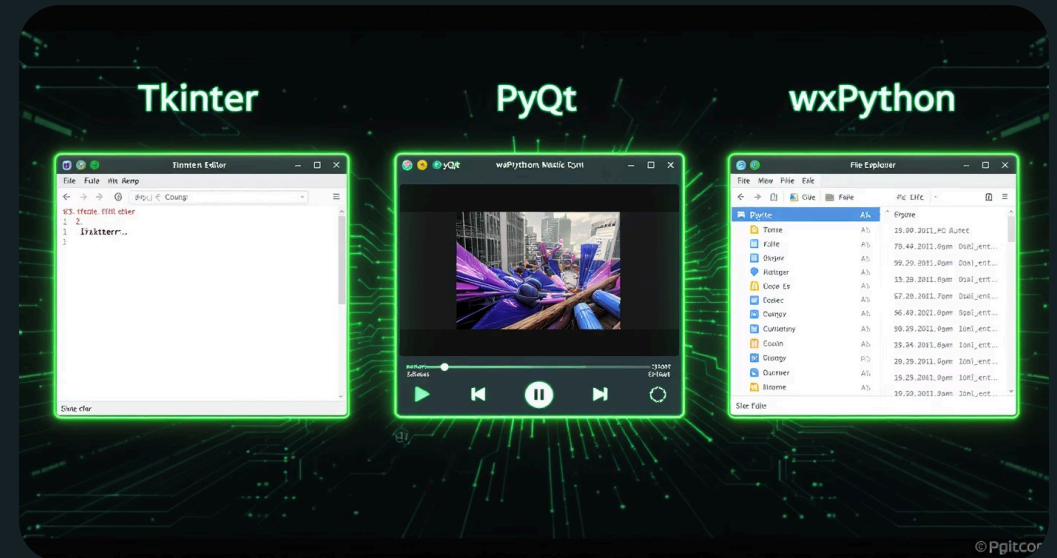
La prima decisione importante riguarda la scelta del framework per l'interfaccia grafica. In Python abbiamo diverse opzioni, ognuna con vantaggi e svantaggi specifici:

- ## Tkinter

Framework standard della libreria Python, semplice ma con meno funzionalità avanzate. Ideale per principianti e progetti semplici.
- ## PyQt/PySide

Più potente e con un aspetto moderno, richiede una licenza commerciale per applicazioni non open source.
- ## wxPython

Buon compromesso tra facilità d'uso e funzionalità avanzate, con un aspetto nativo su tutte le piattaforme.



Per il nostro progetto, utilizzeremo Tkinter per la sua semplicità e perché è incluso nella libreria standard di Python, evitando così l'installazione di dipendenze aggiuntive.

Creazione della Finestra Principale

La finestra principale della nostra applicazione sarà il contenitore di tutti gli elementi dell'interfaccia. È importante strutturarla in modo logico per garantire una buona esperienza utente.

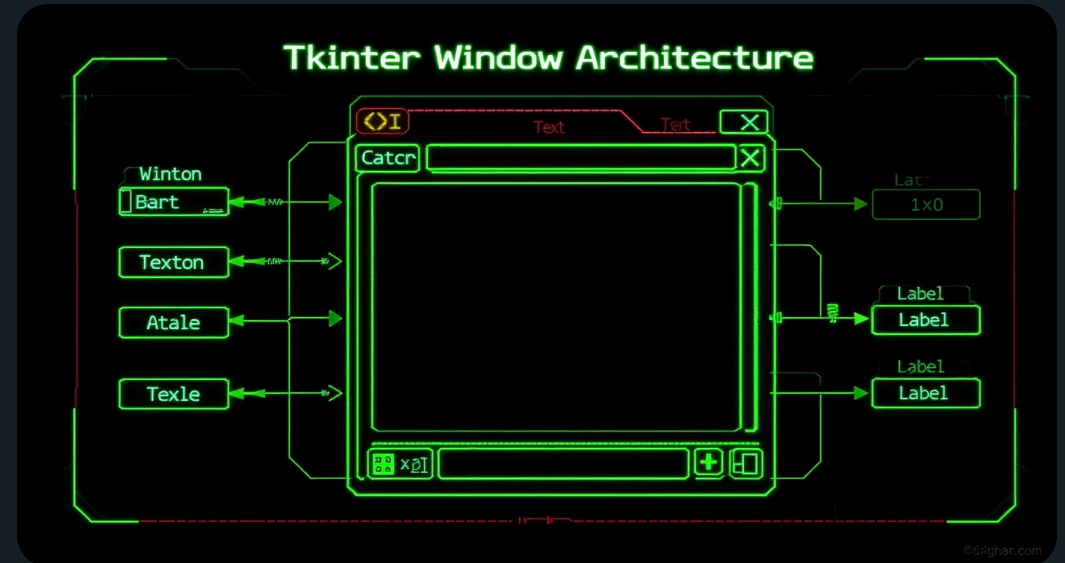
```
import tkinter as tk
from tkinter import ttk

class ChatbotApp:
    def __init__(self, root):
        self.root = root
        self.root.title("OpenAI Chatbot")
        self.root.geometry("800x600")

        # Configurazione della griglia principale
        self.root.columnconfigure(0, weight=1)
        self.root.rowconfigure(0, weight=1)

        # Frame principale
        self.main_frame = ttk.Frame(root)
        self.main_frame.grid(sticky="nsew")

        # Configurazione tema
        self.configure_styles()
```



Nella creazione della finestra principale, definiamo il titolo, le dimensioni e impostiamo la griglia per il posizionamento degli elementi. Utilizziamo un sistema di gestione del layout basato su grid, che offre maggiore flessibilità rispetto ad altri metodi come pack.

Implementazione del Campo Input Utente

Creazione del Frame di Input

Aggiungiamo un frame nella parte inferiore della finestra che conterrà il campo di testo e il pulsante di invio.

```
self.input_frame =  
    ttk.Frame(self.main_frame)  
self.input_frame.grid(row=1,  
    column=0, sticky="ew", padx=10,  
    pady=10)  
self.input_frame.columnconfigure(0,  
    weight=1)
```

Aggiunta del Campo di Testo

Inseriamo un widget Text che permetta l'inserimento di messaggi multi-linea.

```
self.input_field =  
    tk.Text(self.input_frame, height=3,  
    wrap="word")  
self.input_field.grid(row=0, column=0,  
    sticky="ew")  
self.input_field.bind("",  
    self.on_enter_pressed)
```

Implementazione del Pulsante di Invio

Aggiungiamo un pulsante che l'utente può cliccare per inviare il messaggio al modello.

```
self.send_button =  
    ttk.Button(self.input_frame,  
    text="Invia",  
    command=self.send_message)  
self.send_button.grid(row=0,  
    column=1, padx=5)
```

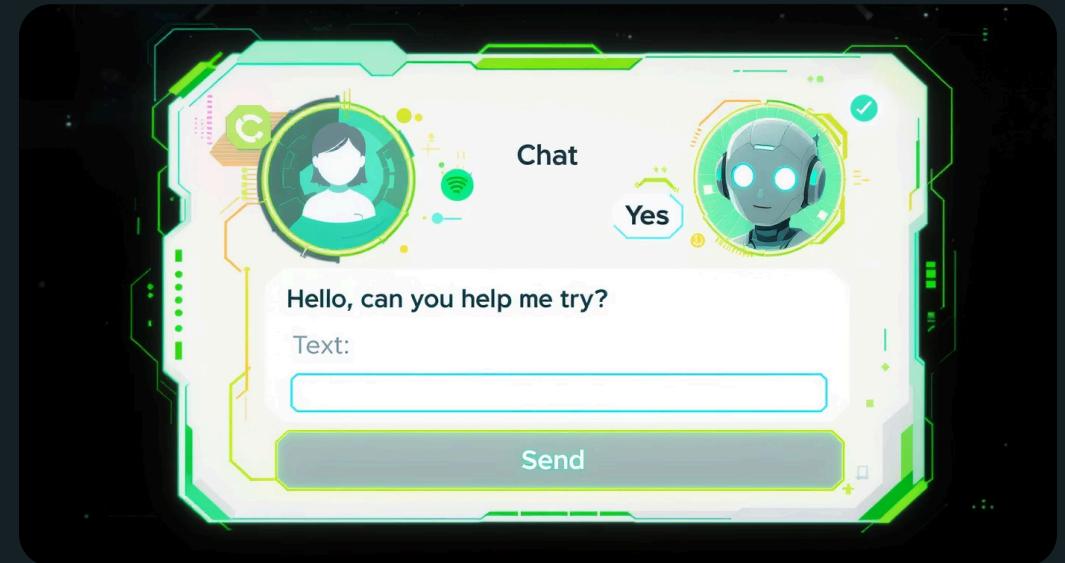
Visualizzazione delle Risposte nella Chat

Per visualizzare lo scambio di messaggi tra l'utente e il chatbot, creeremo un'area di chat scorrevole. Utilizzeremo un widget Text con funzionalità di scrolling per mostrare la conversazione.

```
# Creazione del frame per la chat
self.chat_frame = ttk.Frame(self.main_frame)
self.chat_frame.grid(row=0, column=0, sticky="nsew", padx=10, pady=10)
self.chat_frame.columnconfigure(0, weight=1)
self.chat_frame.rowconfigure(0, weight=1)

# Area di visualizzazione della chat con scrollbar
self.chat_display = tk.Text(self.chat_frame, wrap="word", state="disabled")
self.chat_display.grid(row=0, column=0, sticky="nsew")

self.scrollbar = ttk.Scrollbar(self.chat_frame, orient="vertical",
                              command=self.chat_display.yview)
self.scrollbar.grid(row=0, column=1, sticky="ns")
self.chat_display["yscrollcommand"] = self.scrollbar.set
```



L'area di chat è configurata come "disabled" per impedire all'utente di modificare direttamente il contenuto. Per aggiungere messaggi, dovremo temporaneamente cambiare lo stato a "normal", inserire il testo e poi riportarlo a "disabled".

Per differenziare visivamente i messaggi dell'utente da quelli del chatbot, utilizzeremo tag personalizzati con colori e stili diversi.

Collegamento con l'API di OpenAI



Configurazione dell'API

Prima di utilizzare l'API di OpenAI, dobbiamo configurare la libreria con la nostra chiave API. Utilizzeremo la versione 1.87 della libreria OpenAI come richiesto.

```
import openai

# Configurazione della chiave API
openai.api_key = "YOUR_API_KEY_HERE"
```



Creazione della Funzione di Chiamata

Implementiamo una funzione che invierà il testo dell'utente all'API e recupererà la risposta del modello gpt-4.1-nano.

```
def get_ai_response(self, user_message):
    try:
        response = openai.ChatCompletion.create(
            model="gpt-4.1-nano",
            messages=[
                {"role": "system", "content":
self.system_prompt},
                {"role": "user", "content": user_message}
            ],
            temperature=self.temperature
        )
        return response.choices[0].message.content
    except Exception as e:
        return f"Errore nella comunicazione con OpenAI: {str(e)}"
```



Integrazione nell'Interfaccia

Collegiamo questa funzione al pulsante di invio e al campo di input, in modo che venga chiamata quando l'utente invia un messaggio.

```
def send_message(self):
    user_input = self.input_field.get("1.0", "end-1c").strip()
    if not user_input:
        return

    # Mostra il messaggio dell'utente
    self.display_message(f"Tu: {user_input}",
"user_message")

    # Pulisci il campo di input
    self.input_field.delete("1.0", tk.END)

    # Ottieni e mostra la risposta dell'AI
    ai_response = self.get_ai_response(user_input)
    self.display_message(f"AI: {ai_response}",
"ai_message")
```

Gestione Asincrona delle Chiamate API

Le chiamate API possono richiedere tempo, specialmente quando la rete è lenta o il server è occupato. Per evitare che l'interfaccia utente si blocchi durante l'attesa della risposta, implementeremo una gestione asincrona delle chiamate.

```
import threading
import queue

class ChatbotApp:
    # ... codice precedente ...

    def __init__(self, root):
        # ... inizializzazione precedente ...

        # Coda per la comunicazione tra thread
        self.response_queue = queue.Queue()

        # Avvia il thread di controllo della coda
        threading.Thread(target=self.check_response_queue,
                        daemon=True).start()

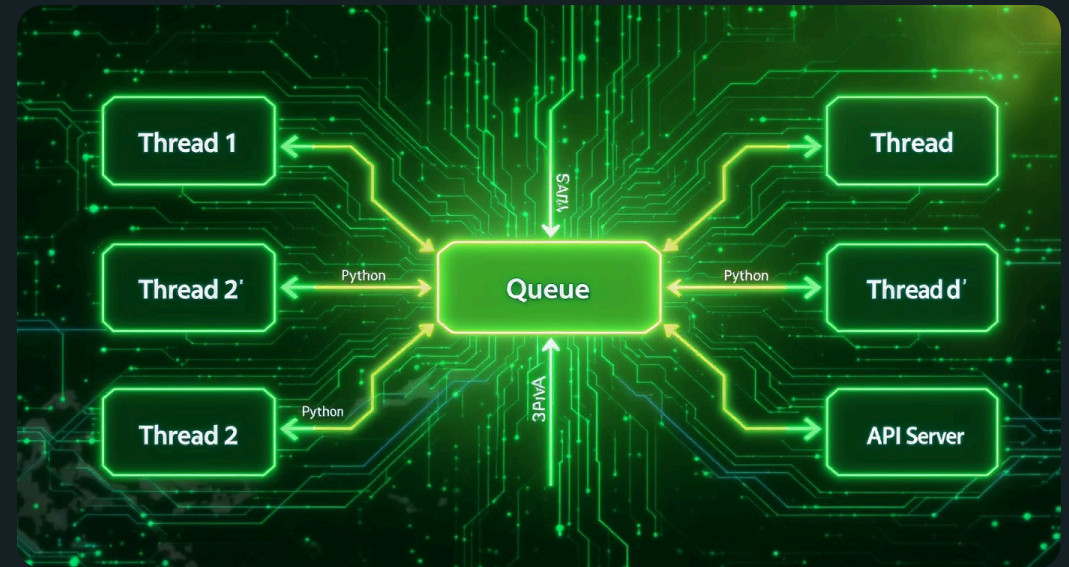
    def send_message_async(self):
        user_input = self.input_field.get("1.0", "end-1c").strip()
        if not user_input:
            return

        # Mostra il messaggio dell'utente
        self.display_message(f"Tu: {user_input}", "user_message")

        # Pulisci il campo di input
        self.input_field.delete("1.0", tk.END)

        # Mostra indicatore di caricamento
        self.display_message("AI: Sto pensando...", "loading")

        # Avvia thread per la chiamata API
        threading.Thread(
            target=self.process_in_background,
            args=(user_input,),
            daemon=True
        ).start()
```



L'implementazione asincrona utilizza thread separati per le chiamate API, mantenendo l'interfaccia reattiva. Utilizziamo una coda per trasferire in modo sicuro i dati tra i thread, e un thread daemon che controlla continuamente la coda per nuove risposte.

```
def process_in_background(self, user_input):
    try:
        response = self.get_ai_response(user_input)
        self.response_queue.put(response)
    except Exception as e:
        self.response_queue.put(f"Errore: {str(e)}")

def check_response_queue(self):
    while True:
        try:
            # Controlla la coda ogni 100ms
            self.root.after(100, self._check_queue)
        except Exception:
            pass

def _check_queue(self):
    try:
        if not self.response_queue.empty():
            response = self.response_queue.get()

            # Rimuovi l'indicatore di caricamento
            self.remove_loading_indicator()

            # Mostra la risposta
            self.display_message(f"AI: {response}", "ai_message")
    except Exception:
        pass
```


Gestione di Errori e Timeout

Tipi di Errori da Gestire

Durante l'interazione con l'API di OpenAI possono verificarsi diversi tipi di errori: problemi di rete, timeout, errori di autenticazione, limiti di rate superati, e problemi con il modello stesso. È fondamentale gestire questi errori in modo elegante per evitare che l'applicazione si blocchi.

- Errori di rete: quando la connessione è assente o instabile
- Timeout: quando il server non risponde entro un tempo ragionevole
- Errori di autenticazione: chiave API non valida o scaduta
- Rate limiting: troppe richieste in un breve periodo

Implementazione del Timeout

Per evitare che l'applicazione rimanga in attesa indefinitamente, implementiamo un timeout per le chiamate API. Se la risposta non arriva entro un determinato tempo, informiamo l'utente e permettiamo di riprovare.

```
def get_ai_response_with_timeout(self, user_message, timeout=30):
    try:
        # Imposta un timeout per la richiesta
        response = openai.ChatCompletion.create(
            model="gpt-4.1-nano",
            messages=[
                {"role": "system", "content": self.system_prompt},
                {"role": "user", "content": user_message}
            ],
            temperature=self.temperature,
            request_timeout=timeout
        )
        return response.choices[0].message.content
    except openai.error.Timeout:
        return "La richiesta è scaduta. La rete potrebbe essere lenta o il servizio sovraccarico. Riprova più tardi."
    except openai.error.APIConnectionError:
        return "Impossibile connettersi ai server OpenAI. Verifica la tua connessione internet."
    except openai.error.AuthenticationError:
        return "Errore di autenticazione. Verifica la tua chiave API."
    except openai.error.RateLimitError:
        return "Hai superato il limite di richieste. Attendi qualche minuto prima di riprovare."
    except Exception as e:
        return f"Si è verificato un errore: {str(e)}"
```


Logging delle Conversazioni su File

È utile registrare le conversazioni per riferimento futuro, analisi o debug. Implementeremo un sistema di logging che salva ogni scambio di messaggi in un file di testo con timestamp.

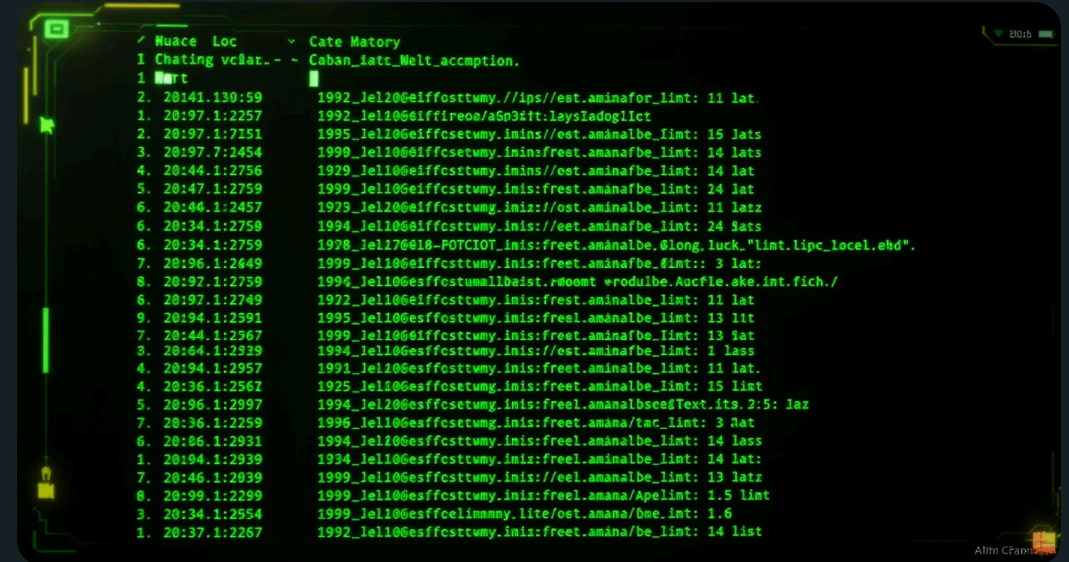
```
import logging
import datetime

def setup_logging(self):
    # Configura il logger
    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s - %(message)s',
        handlers=[
            logging.FileHandler("chatbot_logs.txt"),
            logging.StreamHandler()
        ]
    )

    self.logger = logging.getLogger("ChatbotLogger")

def log_conversation(self, user_message, ai_response):
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    self.logger.info(f"\n--- Nuova conversazione: {timestamp} ---")
    self.logger.info(f"Utente: {user_message}")
    self.logger.info(f"AI: {ai_response}")

    # Aggiorna anche il file di log personalizzato
    with open("conversazioni.txt", "a", encoding="utf-8") as f:
        f.write(f"\n--- {timestamp} ---")
        f.write(f"Utente: {user_message}\n")
        f.write(f"AI: {ai_response}\n\n")
```



Il sistema di logging utilizza sia il modulo logging standard di Python che un file personalizzato. Il primo è utile per il debug tecnico, mentre il secondo è formattato in modo più leggibile per la revisione delle conversazioni.

Oltre a salvare le conversazioni in formato testo, potremmo anche implementare l'esportazione in formati come JSON o CSV per facilitare l'analisi dei dati o l'integrazione con altri strumenti.