

«Talento Tech»

Inteligencia Artificial

Clase 09





Clase N° 9 | Mejoras en la visualización y finalización del chatbot

Temario:

- Mejora de la visualización del historial de chat
- Uso de emojis y formato personalizado
- Preparándonos para el Deploy.
- Implementación de un área de texto para mostrar el chat completo





Conceptos a repasar para la clase:

Bucle for

bucle for

Recordemos que el bucle for es una estructura de control que nos permite iterar sobre una secuencia de un objeto iterable y la variable interna que crea el bucle, también llamada variable de control, toma el valor de cada elemento de la secuencia en cada iteración. Miremos este script y leamos su estructura:

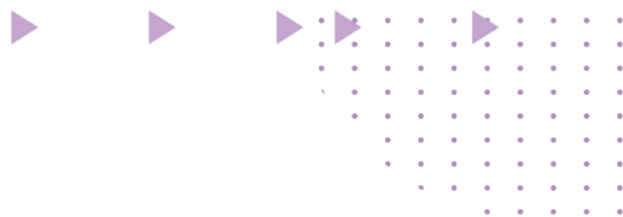
```
suma_pares = 0
for numero in range(1, 101):
    if numero % 2 == 0:
        suma_pares += numero
print(f"La suma de los números pares del 1 al 100 es: {suma_pares}")
```

- **numero** es la variable de control que será todos los números del 1 al 100. Si el condicional es verdadero, entonces lo sumamos a la variable **suma_pares**.

Generador con yield

yield

yield es una palabra clave y especial en Python que nos permite convertir una función normal en un generador. Un generador es un tipo de iterador que genera valores "bajo demanda", osea que en lugar de calcularlos todos de una y almacenarlos en memoria, genera un valor, espera a que sea su turno y cuando le toque, genera otro valor. Cuando una función usa "yield", se pausa en ese punto y devuelve el valor especificado



Probando

Para entender mejor qué es la palabra clave `yield`, lo que haremos será crear una función que genere un número de dos en dos. Esta función en lugar de terminarla con un **`return`** lo vamos a hacer con `yield`. Esto es clave, porque vamos a pausar la función y devolver un valor, **hasta la próxima vez que es llamada y continuará desde donde se quedó.**

```
def pares_generator(limite):  
    # Usamos limite +1 para incluir el número límite si es par  
    for num in range(2, limite + 1, 2):  
        yield num
```

Cuando una función termina con `yield` en lugar del `return` tradicional, **¡convertimos a la función en un generador en lugar de una función!**

```
# Uso del generador  
for par in pares_generator(10):  
    print(par, end=" ")  
# Salida: 2 4 6 8 10
```

cuando lo vamos a usar, creamos un bucle para que itere sobre nuestro generador y en cada iteración, el generador produce el siguiente número par. Esto nos da la ventaja de que no necesitamos crear una lista completa de números pares que ocupen nuestra memoria. Lo que hacemos es generar el número únicamente cuando se necesita, haciendo el programa más eficiente en términos de memoria.

Mostrando la respuesta

La clase pasada terminamos el código con el desafío de desarmar la caja que `streamlit` le arma a la respuesta de `groq`, que hace que no podamos ver correctamente la respuesta a nuestra pregunta. La salida por nuestro sitio es la siguiente:



Lo que estamos viendo es básicamente un elemento de groq, de tipo stream, que lo configuramos y explicamos en la clase 7, que está guardado en una dirección en la memoria de nuestra computadora. Entendemos entonces, que groq nos contesta pero streamlit no lo muestra.

Función generar respuesta

Vamos entonces a generar una función que lea ese “caja” que se va llenando a medidas que groq nos contesta, y vamos sacando palabra por palabra para mostrarla en nuestro navegador. La función la voy a llamar **generar_respuesta** y espero que vaya tomando palabras o frases de la caja y vaya completando una variable de tipo string vacía. Mi función se va a ver de esta forma:

```
def generar_respuesta(chat_completo):
    respuesta_completa = ""
```

En principio mi función va a recibir como parámetro la variable **chat_completo**. Recordá que esta variable era el espacio en memoria donde está la respuesta de groq. También creo internamente una variable llamada **respuesta_completa**, que será un string vacío.

Luego iteramos sobre la variable **chat_completo** y obtenemos frases o fragmentos del código, y tenemos que preguntarnos si hay contenido nuevo en la respuesta. Si hay contenido nuevo, lo agregamos a la variable, y si no hay, pasamos la condición a False.

El código completo nos quedará de esta forma;

```
def generar_respuesta(chat_completo):
    respuesta_completa = ""
    for frase in chat_completo:
```

```

if frase.choices[0].delta.content:
    respuesta_completa += frase.choices[0].delta.content
    yield frase.choices[0].delta.content
return respuesta_completa
    
```

Veamos que es cada parte del código:

1. Recorro el elemento **chat_completo** y obtengo una frase, o un fragmento.
2. Verificamos si la frase actual tiene algún contenido nuevo. En este punto es importante entender que:
 - a. La *frase* suele contener multiples opciones de respuesta, nosotros solamente queremos la primera [0], por lo tanto, seleccionamos la primer opción.
 - b. Dentro de esa frase, tenemos a **delta**, que representa un cambio entre la frase actual y la anterior. Va a contener siempre información nueva.
 - c. **content** en cambio, es el atributo que contiene el texto real. La línea **delta.content** entonces, es un nuevo fragmento de texto.

La comprobación **if frase.choices[0].delta.content** se utiliza para verificar si hay algún contenido nuevo en este fragmento. Esto es importante porque:

- No todos los fragmentos necesariamente contienen nuevo contenido de texto
 - Algunos fragmentos pueden guardar solo metadatos o información de control sobre el input.
 - Ayuda a evitar procesar fragmentos vacíos o irrelevantes.
3. Una vez que tenemos una frase nueva, la agregamos al string vacío creado.
 4. **yield frase.choices[0].delta.content:** Yield al convertir a la función en un generador, permite que el contenido se procese de forma incremental, frase por frase. Dando una mejor sensación de respuesta en tiempo real.

Mi función principal

Ahora que tenemos todas las funciones listas para correr nuestro chatBot, te propongo que organicemos nuestro código en una función principal. Anteriormente nuestro chatBot corría con estas líneas de código:

```

modelo = configurar_pagina()
clienteUsuario = crear_usuario_groq()
inicializar_estado()

mensaje = st.chat_input("Escribí tu mensaje:")
# print(mensaje)
# Verifica que el mensaje no esté vacío antes de configurar el modelo
area_chat()
if mensaje:
    actualizar_historial("user", mensaje, "👤")

    chat_completo = configurar_modelo(clienteUsuario, modelo, mensaje)

    actualizar_historial("assistant", chat_completo, "🤖")

    st.rerun()
    
```

Para organizar nuestro código, crearemos una función principal a la que llamaremos **main()**. A esta función le agregaremos el código visto anteriormente y agregaremos la función **generar_respuesta**.

Vamos a hacerlo siguiendo estos pasos:

1. Creamos **la función main()** y encapsulo toda la lógica del chatbot. Recordá que es muy importante que el código esté bien indentado, para determinar que está dentro de la función y que no.

```

def main():
    modelo = configurar_pagina()
    clienteUsuario = crear_usuario_groq()
    inicializar_estado()

    mensaje = st.chat_input("Escribí tu mensaje:")
    # print(mensaje)
    # Verifica que el mensaje no esté vacío antes de configurar el modelo
    area_chat()
    if mensaje:
        actualizar_historial("user", mensaje, "👤")
    
```

```

        chat_completo = configurar_modelo(clienteUsuario, modelo,
mensaje)

        actualizar_historial("assistant", chat_completo, "🤖")

    st.rerun()
    
```

En la mayoría de las aplicaciones, crear una función principal para que corra todo nuestro código nos va a permitir:

- a) Organizar y visualizar mejor el flujo de la aplicación.
 - b) Centralizar la lógica principal del chatbot
 - c) Manejar el estado de la conversación de manera más eficiente.
 - d) Inicializar variables necesarias en un solo lugar.
 - e) Mejorar y facilitar la integración de nuevas funcionalidades en el futuro, como también su mantenimiento.
2. Incorporamos la función **generar_respuesta**:
- Para incorporar esta función, tengo que tener en cuenta cuál es su funcionalidad. Esta es descomprimir un espacio en memoria que nosotros guardamos en la variable **chat_completo**. Entonces lo primero que tengo que hacer es saber si la variable está "llena", lo que si es cierto, nos indica que la configuración del modelo fue exitosa. Eso lo hago con un condicional de la siguiente manera:

```
if chat_completo:
```

3. Dentro de este condicional, vamos a agregar las siguientes líneas de código

```

with st.chat_message("assistant"):
    respuesta_completa = st.write_stream(generar_respuesta(chat_completo))
    actualizar_historial("assistant", respuesta_completa, "🤖")

    st.rerun()
    
```


- Con **with st.chat_message("assistant")**, abro un bloque de streamlit, que nos crea un cuadro de diálogo con el nuevo mensaje en la interfaz de streamlit, en nuestro caso, con el formato de asistente.
- Debajo creo una variable llamada **respuesta_completa**, donde obtengo con **st.write_stream**, la respuesta completa de nuestro chatBot. Recordá que cuando configuramos el modelo, le dimos el parámetro **stream = True**, Lo que nos va a permitir que la respuesta se genere y muestre progresivamente, dando la sensación de una respuesta en tiempo real.
La función **write_stream**, (Explicar) va a ir escribiendo ese mensaje que nos va dando la función **generar_respuesta** de forma progresiva.
- Luego actualizamos el historial y recargamos la aplicación, de la misma forma que lo veníamos haciendo.

El código completo de nuestra función principal debería quedarnos de esta forma:

```
def main():
    modelo = configurar_pagina()

    clienteUsuario = crear_usuario_groq()
    inicializar_estado()

    mensaje = st.chat_input("Escribí tu mensaje: ")

    area_chat()
    if mensaje:
        actualizar_historial("user", mensaje, "👤")

        chat_completo = configurar_modelo(clienteUsuario, modelo, mensaje)

        if chat_completo:
            with st.chat_message("assistant"):
                respuesta_completa =
st.write_stream(generar_respuesta(chat_completo))
                actualizar_historial("assistant", respuesta_completa, "🤖")

            st.rerun()
```

Una vez realizada mi función principal, voy a necesitar escribir un condicional que permita que mi archivo se ejecute como un archivo principal. Para eso, escribo lo siguiente:

```
if __name__ == "__main__":  
    main()
```

Con esta línea de código nos aseguramos de que nuestro archivo se ejecute únicamente cuando se ejecuta directamente y no cuando es importado como una librería.

Para entender mejor esta línea de código, podemos pensar que hemos creado un archivo que contiene varias funciones, y todos los archivos de Python tienen una variable especial llamada `__name__` (ya creada).

Si un archivo se está ejecutando directamente (por ejemplo, usando `python script.py`), entonces `__name__` pasa a ser `"__main__"`. Si el archivo se está importando como una librería o módulo en otro archivo, `__name__` pasa a valer el nombre del archivo (sin la extensión `.py`).

Esto es lo que nos permite llamar a cada una de sus funciones desde un archivo secundario.

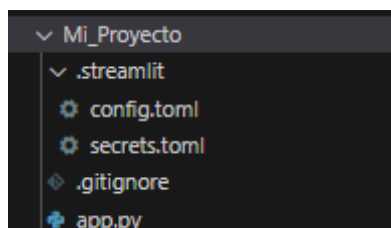
Esto es muy útil porque nos permite definir un único punto de entrada a nuestras funciones (el condicional), asegurando que si alguien quiere reutilizar funciones de nuestro código importándolas en otro archivo, no se ejecute el bloque de código que contiene el condicional, en nuestro caso, la función `main`.

¡Preparando para Deploy!

Deployar una aplicación o sitio web, quiere decir que vamos a poner en funcionamiento o hacer disponible nuestra app en un entorno donde los usuarios finales puedan acceder desde cualquier dispositivo. Esto es un proceso que se hace al final pero nosotros fuimos haciéndolo a medida que avanzamos en el proyecto.

Para configurar que cosas queremos subir y que cosas no, vamos a crear un archivo en mi carpeta raíz, por encima de nuestra aplicación, con el siguiente nombre: **.gitignore** (punto gitignore)

La carpeta principal del proyecto debería tener una estructura similar a esta:



Este archivo será importante para poder cargar solamente lo necesario de nuestra aplicación. Es un archivo que hace referencia a las cosas que tenemos que ignorar al subir la app.

Entonces vamos a agregar las cosas que están en la carpeta de mi aplicación pero que NO queremos subir a nuestro servidor. Te dejamos una imagen con algunas formas de ignorar diferentes archivos.

```

Mi_Proyecto > .gitignore
1  # ignora todos los archivos con extension .txt
2  *.txt
3
4  # Ignora el archivo preguntas.html puntualmente
5  . preguntas.tml
6
7  # Ignora la carpeta capturas_de_pantallas y la carpeta .env
8  capturas_de_pantallas
9  .env
10
    
```

En nuestro caso, no haremos un deploy del archivo .streamlit, porque lo haremos de forma manual, por lo que nuestro .gitignore tiene que tener al menos, la carpeta .streamlit.

En mi .gitignore voy a ignorar subir la carpeta .streamlit.

Una vez creado, el conjunto de carpetas y archivos vamos a dejar lista nuestra aplicación para la clase que viene subirlo a un servidor de internet.

Para continuar con el orden, nos aseguramos de tener en **una carpeta** los siguientes archivos:

1. **Mi archivo principal**

En este archivo es donde tengo todo el código de mi aplicación, incluyendo la función principal (main.py)

2. **Carpeta .streamlit**

Esta carpeta será la que tenga las configuraciones y las palabras secretas. Para eso, esta carpeta tiene que tener dos archivos dentro:

- a. config.toml : Archivo donde trabajamos la configuración del tema de mi aplicación
- b. secrets.toml : Archivo donde guardamos la información sensible, en nuestro caso la API_KEY.

3. **Archivo .gitignore.**

El contenido del archivo puede variar dependiendo los archivos que tengamos cargados en nuestra aplicación.

4. **requirements.txt**

Este archivo hace referencia a los requerimientos que necesita nuestra aplicación para funcionar. Los hosting, suelen buscar un archivo con este nombre para instalar las aplicaciones necesarias para el buen funcionamiento de la aplicación.

En nuestro caso, crearemos el requirements.txt escribiendo en la terminal (ctr + shift + ñ, o yendo a terminal, nueva terminal) lo siguiente:

pip freeze > requirements.txt

¡importante! Es necesario crear el archivo requirements.txt dentro de la carpeta del proyecto. Por eso te pedimos que verifiques la ruta en la que vas a ejecutar tu comando. Si no sabes como llegar a esa carpeta, podés hacer click derecho sobre la carpeta, y darle click a Open in External Terminal. Esto te abrirá una terminal externa en la que podés ejecutar el mismo comando.

Esto nos creará el archivo con todas las dependencias. Esas dependencias son librerías externas que necesitamos instalar. Nosotros además de streamlit, usamos la librería groq, por lo que es indispensable que se encuentren mínimamente esas dos. Dependiendo la computadora, el archivo puede variar, pero se ve mas o menos de esta forma:



```

Mi_Proyecto > requirements.txt
1  accelerate==0.0.1
2  aiohttp==3.8.3
3  aiohttp==1.3.1
4  altair==5.2.0
    
```

Aparecerán listadas todas las librerías de Python que hay en mi sistema. Podemos tener pocas o muchísimas líneas. En verde se ve el nombre de la librería, y al lado de los dos signos de asignación, el número de versión.

El directorio de nuestro proyecto tiene que verse de esta forma:

```

v Mi_Proyecto
> .streamlit
♦ .gitignore
+ app.py
≡ requirements.txt
    
```

¡Nota! Generalmente cuando trabajamos con aplicaciones o programas grandes que requieran tener un archivo `requirements.txt`, es muy útil crear un entorno virtual. Esto encapsula nuestro proyecto, alojando solamente las dependencias necesarias. Nos permite trabajar con diferentes versiones y librerías, que una vez terminado el proyecto y eliminada la carpeta, se eliminan de nuestra máquina.

Luego de que hagamos hecho todos los pasos anteriores deberíamos tener nuestro chatbot similar a esta imagen:




Configuración de la IA

Elegí un Modelo


llama3-8b-8192

Saludar

Mi chat de IA



Ya hice mi desafío número 9 y mi chatbot funciona de manera local excelente, ¿me darías un mensaje de felicitación y ánimo para ahora deployarlo?



¡FELICIDADES!

¡Estoy emocionado de saber que has logrado el desafío número 9 y que tu proyecto de chatbot local funciona de manera impecable!

Ahora que has superado este hito, te quiero animar y felicitar por el próximo paso: DEPLOYAR TU CHATBOT.

Me imagino que debes sentirte orgulloso y emocionado de llevar tu creación al mundo exterior y compartirla con más personas. ¡Eso es un gran reto, y estoy seguro de que podrás superarlo!

Aquí te dejo un mensaje de ánimo para cuando te sientas inseguro o tenso: "Has ido lejos, pero no hay nada imposible que no se pueda hacer. ¡Crea un proceso de despliegue y asegurate de que esté adecuadamente probado antes de enviarlo al mundo!"

Recuerda que depolyar un proyecto puede ser un proceso desafiante, pero es un gran logro cuando se consigue. ¡Estoy aquí para apoyarte y celebrarte cuando lo hagas!

¡Buena suerte, y que tu chatbot dé saltos de alegría en el mundo!

Escribí tu mensaje:



Desafío N° 9:

En esta clase terminamos nuestro chatBot y lo podemos implementar de forma local en nuestra computadora. Esto lo hacíamos ejecutando el comando

streamlit run nombreDeMiArchivo

Esto nos permite ver el desarrollo de nuestra aplicación en una URL local en su navegador predeterminado similar a esta:

`http://localhost:8501/`

Lo que vamos a hacer en este desafío es dejar todo listo para la carga de nuestra aplicación en streamlit cloud, y poder ejecutar nuestro chatbot personalizado desde cualquier dispositivo conectado a internet.

Para la entrega de este desafío, será necesario que entregues una captura de pantalla donde se muestre el chatBot funcionando, es decir, donde se vea la generación de la respuesta, con su .gitignore, requirements.txt y además, el proyecto comprimido.





Buenos Aires
aprende
Agencia de Actividades para el Futuro

BA Buenos
Aires
Ciudad