1. Modelos (Models)

Es la abstracción de LangChain para interactuar con diferentes LLMs.

- **Concepto:** En lugar de usar la SDK de OpenAI directamente, usás `ChatOpenAI`. Esto te permite cambiar de modelo (ej. de GPT a otro) cambiando solo una línea de código.
- **Parámetro Clave:** `temperature=0.0`. En desarrollo de aplicaciones (como lo que busca Ualá), siempre se usa en 0 para que la respuesta sea determinista y no invente cosas distintas cada vez.

2. Prompts (Templates)

Es la forma de profesionalizar las instrucciones.

- **`ChatPromptTemplate`**: En lugar de concatenar strings (`f-strings`), creás una plantilla con variables entre llaves `{variable}`.
- **Ventaja técnica:** Separa la **lógica de la instrucción** de los **datos del usuario**. LangChain se encarga de combinar el "mensaje del sistema" con el "mensaje del usuario" de forma estructurada.
- **Uso:** `prompt_template.format_messages(style=customer_style, text=customer_review)`.

3. Output Parsers (La pieza que conecta con el Backend)

Esta es la parte fundamental para vos como desarrollador. Los LLMs devuelven **texto**, pero tu código necesita **objetos**.

- **`ResponseSchema`**: Definís qué campos querés extraer (ej: `name`, `gift`, `delivery_days`) y qué tipo de dato son.
- **`StructuredOutputParser`**:
    1. **Inyección de instrucciones:** El parser genera un texto técnico (format instructions) que se añade al prompt diciéndole a la IA: *"Respondé solo en formato JSON con estas claves"*.
    2. **Parsing:** Una vez que la IA responde, el parser toma ese string y lo convierte en un **diccionario de Python** (`dict`).
- **Resultado:** Podés hacer `response_as_dict['delivery_days']` y operar con ese dato (guardarlo en PostgreSQL, por ejemplo).

El "Workflow" de Código (Resumen del Jupyter)

1. **Definir Esquema:** Decís qué querés obtener (JSON).
2. **Crear Prompt:** Armás la plantilla que incluye las instrucciones de formato.
3. **Llamar al Modelo:** Mandás el prompt formateado al LLM.

4. **Parsear:** Pasás el texto del LLM por el parser para obtener el diccionario.

MEMORY:

Resumen Técnico: Gestión de Memoria en LangChain

1. El Problema de la Amnesia (Statelessness)

- Los modelos de lenguaje (LLMs) por diseño no recuerdan interacciones pasadas.
- Cada petición a la API es independiente; sin un sistema de memoria, la IA no sabría quién es "Luciano" en el segundo mensaje.

2. La Solución: ConversationSummaryBufferMemory

Este es el tipo de memoria más avanzado que vimos porque es **híbrida**:

- **Buffer Reciente**: Mantiene los últimos mensajes con exactitud para no perder el hilo inmediato de la charla.
- **Resumen (Summary)**: Cuando la charla supera un límite (en tu código, `max_token_limit=100`), utiliza al propio LLM para resumir la historia antigua.
- **Optimización**: Esto ahorra dinero (menos tokens enviados) y evita chocar contra el límite de contexto del modelo.

3. Componentes Clave en tu Código

- **`verbose=True`**: Es fundamental en desarrollo. Te permite ver el "Prompt" completo que LangChain construye, incluyendo el historial que se le envía a Groq por detrás.
- **`max_token_limit`**: Es el gobernador de la memoria. Define qué tan "fresca" o "resumida" será la información.
- **`ConversationChain`**: Es el orquestador que une al modelo (`llm`) con la base de datos de la charla (`memory`).

```
# ================================================================
# 1. CONFIGURACIÓN DEL ENTORNO
# ================================================================
import os
import warnings
import langchain
from dotenv import load_dotenv

load_dotenv()
```

```python
# Parche de compatibilidad
if not hasattr(langchain, 'verbose'):
    langchain.verbose = False

from langchain_groq import ChatGroq
from langchain.chains import ConversationChain
from langchain.memory import ConversationSummaryBufferMemory

# Solución para errores de definición de Pydantic
try:
    from langchain_core.caches import BaseCache
    from langchain_core.callbacks import Callbacks
    ConversationSummaryBufferMemory.model_rebuild()
except Exception:
    pass

warnings.filterwarnings('ignore')

# ==========================================================================
# 2. INICIALIZACIÓN DEL MODELO Y MEMORIA
# ==========================================================================

# Usamos ChatGroq directamente para evitar errores de conteo de tokens
llm = ChatGroq(
    temperature=0.0,
    model_name="llama-3.1-8b-instant",
    groq_api_key=os.environ.get("GROQ_API_KEY")
)

memory = ConversationSummaryBufferMemory(llm=llm, max_token_limit=100)

conversation = ConversationChain(
    llm=llm,
    memory=memory,
    verbose=True
)

# ==========================================================================
# 3. PRUEBA DE LA CONVERSACIÓN
# ==========================================================================

print("--- Turno 1 ---")
```

```
print(conversation.predict(input="Hola, soy Luciano, Analista
Programador de Argentina."))

print("\n--- Turno 2 (Llenando buffer de tokens) ---")
texto_largo = """
Mi agenda para mañana:
9:00 - Review de código con el equipo senior.
11:00 - Entrevista técnica en Ualá.
14:00 - Despliegue a producción de la nueva API.
16:00 - Gimnasio.
"""
print(conversation.predict(input=f"Anotá mi agenda: {texto_largo}"))

print("\n--- Turno 3 (Verificando memoria histórica) ---")
print(conversation.predict(input="¿Qué recordás de mi perfil y qué
planes tengo para mañana?"))
```

¿Cómo funcionó en tu código?
Fijate en lo que salió en tu consola en el Turno 3:

1. Superaste el límite: Como pusimos max_token_limit=100 (un límite muy bajo), en cuanto le pasaste la agenda larga, el programa dijo: "Upa, si guardo todo esto tal cual, me paso de 100 tokens".
2. El Resumen automático: Antes de responder, LangChain llamó a la IA por detrás y le pidió: "Resumime todo lo que hablamos hasta ahora".
3. El nuevo Prompt: Si miras el Turno 3, verás que dice: System: Current summary: Luciano, an Argentine programmer analyst... La IA ya no está leyendo "Hola soy Luciano...", está leyendo el resumen condensado.

¿Por qué esto es mejor que otras memorias?

Una **Chain** combina un LLM con un Prompt, pero su verdadero poder reside en que puedes unir estos bloques para realizar secuencias de operaciones complejas sobre tus datos.

---

1. LLMChain: El bloque básico

Es la cadena más sencilla: une un modelo con un prompt específico.

Ejemplo: Queremos que la IA invente nombres para empresas basadas en un producto.

Python
from langchain_groq import ChatGroq
from langchain.prompts import ChatPromptTemplate
from langchain.chains import LLMChain

```
# Configuración con Groq
llm = ChatGroq(temperature=0.9, model_name="llama-3.1-8b-instant")

prompt = ChatPromptTemplate.from_template(
    "¿Cuál es el mejor nombre para una empresa que fabrica {product}?"
)

# Creamos la cadena básica
chain = LLMChain(llm=llm, prompt=prompt)

# Ejecución
product = "Sábanas de seda"
print(chain.run(product))
```

---

2. SimpleSequentialChain: Flujo lineal

Se usa cuando tienes una serie de pasos donde **la salida de uno es la entrada del siguiente**. Solo admite una entrada y una salida por paso.

Ejemplo: 1) Crear el nombre de la empresa -> 2) Crear una descripción para ese nombre.

Python
```
from langchain.chains import SimpleSequentialChain

# Paso 1: Generar nombre
prompt1 = ChatPromptTemplate.from_template("Nombre para empresa de {product}?")
chain1 = LLMChain(llm=llm, prompt=prompt1)

# Paso 2: Generar descripción
prompt2 = ChatPromptTemplate.from_template("Escribe una descripción de 20 palabras
para: {company_name}")
chain2 = LLMChain(llm=llm, prompt=prompt2)

# Unimos las piezas
overall_simple_chain = SimpleSequentialChain(chains=[chain1, chain2], verbose=True)
overall_simple_chain.run("Sábanas de seda")
```

---

3. SequentialChain: Flujos complejos (Múltiples entradas/salidas)

A diferencia de la anterior, aquí puedes manejar **múltiples variables** que se pasan entre distintos pasos. Es crucial que los `input_keys` y `output_keys` coincidan exactamente.

Ejemplo del video:

1. Traducir una reseña a inglés (`English_Review`).
2. Resumir esa traducción (`summary`).
3. Detectar el idioma original (`language`).
4. Escribir una respuesta en el idioma original usando el resumen.

Python
```
from langchain.chains import SequentialChain

# Cadena 1: Traducción
chain_one = LLMChain(llm=llm, prompt=prompt1, output_key="English_Review")

# Cadena 2: Resumen (usa English_Review)
chain_two = LLMChain(llm=llm, prompt=prompt2, output_key="summary")

# Cadena 4: Respuesta final (usa summary Y language)
chain_four = LLMChain(llm=llm, prompt=prompt4, output_key="followup_message")

overall_chain = SequentialChain(
    chains=[chain_one, chain_two, chain_three, chain_four],
    input_variables=["Review"],
    output_variables=["English_Review", "summary", "followup_message"],
    verbose=True
)
```

---

4. Router Chain: El "Telefonista" inteligente

Esta es la cadena más avanzada. Permite enviar una pregunta al experto correcto. Tienes varios "sub-prompts" (Física, Matemáticas, Historia) y una **Router Chain** decide a cuál enviárselo según el contenido.

- **LLMRouterChain**: Usa un LLM para decidir el destino basándose en descripciones.
- **Default Chain**: Se usa si la pregunta no encaja en ninguna categoría (ej: preguntas de biología cuando solo tienes expertos en física/mate).

Código Clave:

Python
```
from langchain.chains.router import MultiPromptChain

# Definimos el ruteador que decide
chain = MultiPromptChain(
    router_chain=router_chain,
    destination_chains=destination_chains, # Diccionario de expertos
    default_chain=default_chain,
    verbose=True
)
```

chain.run("¿Qué es la radiación de cuerpo negro?") # Irá a Física

FLUJO MEJOR EXPLICADO:

1. El Concepto: El "Router" o Enrutador

Imaginá que tenés un equipo de expertos (Física, Matemática, Historia, Computación). Cuando llega una pregunta, necesitás un **recepcionista** que entienda de qué trata el tema y se la pase al experto correcto.

- **Si la pregunta es de código**, va al experto en Computer Science.
- **Si es sobre el pasado**, va al Historiador.
- **Si no encaja en nada**, va a una "cadena por defecto" (Default Chain).

---

2. Estructura del Código (Los 3 pilares)

Para que esto funcione en LangChain, necesitás tres componentes clave:

A. Las Cadenas de Destino (Destination Chains)

Son los "expertos". Cada uno tiene su propio `prompt_template` especializado.

Python
# Ejemplo de un experto
physics_template = "Eres un profesor de física brillante... Pregunta: {input}"
# Se guardan en un diccionario
destination_chains = {"physics": physics_chain, "math": math_chain, ...}

B. El Router (El Cerebro)

Es una cadena especial (`LLMRouterChain`) que usa un prompt para decidir el destino. Este prompt contiene las **descripciones** de cada experto (ej: "Bueno para responder preguntas de física").

C. La Cadena por Defecto (Default Chain)

Es la red de seguridad. Si el router no sabe a dónde mandar la pregunta, cae aquí para que el LLM responda de forma genérica.

---

3. ¿Cómo funciona el flujo de datos?

1. **Entrada:** Vos mandás una pregunta: *"¿Qué es la radiación de cuerpo negro?"*.

2. **Decisión:** El Router analiza la pregunta y busca en las descripciones de los expertos.

JSON de salida: El Router genera un JSON interno (gracias al `RouterOutputParser`) que dice:
JSON
{
  "destination": "physics",
  "next_inputs": "What is black body radiation?"
}

3.
4. **Ejecución:** LangChain toma ese JSON y "dispara" la `physics_chain`.

—----------

🏛️ Concepto Core: RAG (Retrieval-Augmented Generation)

Es la técnica para "darle memoria externa" a un LLM. En lugar de re-entrenar el modelo (caro y lento), recuperamos información relevante en el momento y se la inyectamos en el prompt.

🛠️ Los 5 Componentes Clave del Pipeline

1. Document Loaders (Cargadores)

- **Qué son:** Clases que transforman tus archivos (PDF, CSV, TXT, Notion) en objetos `Document` estandarizados de LangChain.
- **Tu ejemplo:** `CSVLoader` toma cada fila de un Excel y crea un "documento" con su contenido.

2. Embeddings (Vectorización)

- **Qué son:** El "traductor" de texto a números. Convierte frases en listas de números flotantes (vectores).
- **La Clave:** Frases con significados semánticos similares (ej: "perro" y "canino") tendrán vectores matemáticamente cercanos. Esto permite buscar por *significado*, no solo por palabras clave.

3. Vector Store (Base de Datos Vectorial)

- **Qué es:** El motor donde guardamos esos embeddings para hacer búsquedas rápidas.

- **Ejemplos:** `DocArrayInMemorySearch` (memoria RAM, volátil), `ChromaDB`, `Pinecone` (persistentes).
- **Función:** Cuando llega una pregunta, calcula la distancia entre el vector de la pregunta y los millones de vectores guardados.

4. The Retriever (El Buscador)

- **Qué es:** La interfaz que conecta tu código con la Vector Store.
- **Configuración clave:** `k=4` (o el número que quieras). Define *cuántos* fragmentos de documentos traerá de la base de datos.

5. Chain Type: "Stuff" (El Relleno)

- **Qué es:** La estrategia de composición del prompt final.
- **Mecánica:** Toma los textos de los documentos recuperados por el Retriever, los concatena (los pega uno tras otro) y los mete dentro de la variable `{context}` del prompt.
- **Limitación:** Solo funciona si la cantidad de texto recuperado cabe en la ventana de contexto del LLM (ej: 8k tokens).

🔄 El Flujo de Datos (Runtime)

Cuando el usuario hace una pregunta ("Query"):

1. **Vectorización:** La pregunta se convierte en vector.
2. **Retrieval (Búsqueda):** La Vector Store encuentra los fragmentos más parecidos.
3. **Stuffing (Relleno):** LangChain une esos fragmentos en un solo string de texto.
4. **Generation (Respuesta):** Se envía el Prompt final (Instrucciones + Contexto Rellenado + Pregunta) al LLM para que genere la respuesta natural.

Chain Types

1. Stuff (El "Embutido") - *Ya lo conocés*

- **Mecánica:** Toma todos los documentos recuperados, los pega uno tras otro y los manda en **un solo prompt** al LLM.
- **Pros:** Es el más rápido (una sola llamada a la API) y mantiene el contexto completo entre documentos.
- **Contras:** Choca de frente contra el límite de tokens (Context Window). Si recuperás mucho, explota.
- **Caso de uso:** Pocos documentos y cortos.

---

2. Map_reduce (Divide y Vencerás)

- **Mecánica:** Funciona en dos etapas:
    1. **Map:** Envía *cada* documento recuperado al LLM por separado con la pregunta. ("Leé el doc 1 y respondé", "Leé el doc 2 y respondé...").
    2. **Reduce:** Toma todas las respuestas individuales y hace una llamada final al LLM para que las "resuma" en una respuesta definitiva.
- **Pros:** Puede procesar **infinitos documentos** (no tiene límite de tokens porque procesa de a uno). Es **paralelizable** (puedes correr los Maps al mismo tiempo).
- **Contras:** Es más caro (muchas llamadas a la API) y puede perder información si la respuesta depende de conectar datos que están en dos documentos distintos (porque el LLM nunca los ve juntos hasta el final).

---

3. Refine (La Bola de Nieve)

- **Mecánica:** Es un proceso **iterativo y secuencial**.
    1. Manda el Doc 1 al LLM y obtiene una respuesta preliminar.
    2. Manda el Doc 2 + la Respuesta Preliminar al LLM y le dice: *"Usá esta nueva info para mejorar/refinar la respuesta anterior"*.
    3. Repite hasta terminar los documentos.
- **Pros:** Genera las respuestas más completas y detalladas porque va acumulando conocimiento.
- **Contras:** Es **lento**. No se puede paralelizar porque el paso 2 depende del paso 1. También es caro.

---

4. Map_rerank (El Casting)

- **Mecánica:**
    1. Manda cada documento al LLM junto con la pregunta.
    2. Le pide al LLM que no solo responda, sino que también devuelva un **Score de Certeza** (ej: "¿Qué tan seguro estás de que la respuesta está acá?").
    3. Al final, **devuelve solo la respuesta con el score más alto**.
- **Pros:** Útil si sabés que la respuesta exacta está en un solo documento y no necesitás combinar información.
- **Contras:** A veces el LLM "miente" con el score y es caro porque procesa todo.

—----------

EVALUATION

Esta lección cubre **Evaluation** (Evaluación) y presenta el concepto de **"LLM-Assisted Evaluation"** (Usar una IA para evaluar a otra IA).

---

1. El Problema de Evaluar LLMs

En el video, Harrison Chase explica que al construir apps complejas (como el RAG que hicimos antes), necesitás saber si tus cambios mejoran o empeoran el sistema.

- **El desafío:** Si la respuesta correcta es *"Sí, tiene bolsillos"* y tu bot responde *"Efectivamente, la prenda cuenta con almacenamiento lateral"*, un test de código tradicional (`string match` o `regex`) diría que **FALLÓ** porque las palabras son distintas.
- **La solución:** Usar un LLM para evaluar la **semántica** (el significado) y no la sintaxis.

---

2. Generación de Datos de Prueba (Test Set)

Para evaluar, primero necesitás un examen: preguntas y respuestas correctas (Ground Truth).

A. Método Manual (Hard-coded)

Es lo clásico. Mirás tus documentos y escribís a mano:

```
Python
examples = [
   {
      "query": "¿El set Cozy Comfort tiene bolsillos?",
      "answer": "Sí"
   }
]
```

- **Problema:** No escala. Si tenés 1000 documentos, no vas a escribir 1000 preguntas a mano.

B. Método Automático (`QAGenerateChain`)

Aquí usamos la IA para crear el examen.

- **Cómo funciona:** Le pasamos los documentos al LLM y le decimos: *"Leé esto y generá un par de preguntas y respuestas basadas en el texto"*.

En el código:

```
Python
from langchain.evaluation.qa import QAGenerateChain
```

```
# Creamos la cadena generadora
example_gen_chain = QAGenerateChain.from_llm(ChatOpenAI(model=llm_model))

# Le pasamos los primeros 5 productos del catálogo
new_examples = example_gen_chain.apply_and_parse(
    [{"doc": t} for t in data[:5]]
)
```

- **Resultado:** Obtenemos una lista de diccionarios con `query` y `answer` generados automáticamente. ¡Acabamos de automatizar la creación de Unit Tests!

---

3. Evaluación Manual: El Debugger (`langchain.debug`)

Antes de correr una evaluación masiva, querés ver qué está pasando "bajo el capó" con un solo caso.

- **Herramienta:** `langchain.debug = True`.
- **Qué hace:** Habilita el modo "verbose" extremo. Te muestra cada paso de la cadena:
    1. Qué documentos recuperó el Retriever.
    2. Cómo se hizo el "Stuffing" (el prompt gigante).
    3. Qué tokens exactos entraron y salieron del LLM.
    4. El costo en tokens.

Esto es vital para detectar si el error fue del **Modelo** (alucinó) o del **Retriever** (trajo el documento equivocado).

---

4. Evaluación Asistida por LLM (`QAEvalChain`)

Esta es la joya de la lección. Una vez que tenés tus ejemplos (preguntas + respuestas reales) y hacés que tu bot responda (predicciones), necesitás alguien que corrija el examen.

Usamos **otro LLM** como profesor correttore.

El Flujo:

1. **Input:**
    - Pregunta: *"¿Tiene bolsillos?"*
    - Respuesta Real (Ground Truth): *"Sí"*
    - Respuesta del Bot (Prediction): *"El set Cozy Comfort efectivamente trae bolsillos laterales."*
2. **Proceso:** `QAEvalChain` recibe esto y le pregunta al LLM evaluador: *"¿La respuesta del bot coincide semánticamente con la respuesta real?"*.
3. **Output:** *"CORRECTO"* o *"INCORRECTO"* (y a veces una explicación/grade).

En el código:

Python
from langchain.evaluation.qa import QAEvalChain

# 1. Obtenemos las respuestas de nuestro bot para todos los ejemplos
predictions = qa.apply(examples)

# 2. Creamos la cadena evaluadora
eval_chain = QAEvalChain.from_llm(llm)

# 3. El LLM corrige el examen
graded_outputs = eval_chain.evaluate(examples, predictions)

# 4. Imprimimos los resultados
for i, eg in enumerate(examples):
    print(f"Pregunta: {predictions[i]['query']}")
    print(f"Respuesta Real: {predictions[i]['answer']}")
    print(f"Respuesta Bot: {predictions[i]['result']}")
    print(f"Calificación: {graded_outputs[i]['text']}") # <--- Aquí dice "CORRECT

—----------

AGENTES:

1. ¿Qué es un Agente?

El concepto clave es que el LLM deja de ser solo quien "responde" y pasa a ser quien **decide qué hacer**.

- **Sin Agente:** Usuario pregunta -> LLM responde (usando lo que recuerda de su entrenamiento).
- **Con Agente:** Usuario pregunta -> LLM piensa: *"No sé esto, debería buscar en Google"* -> LLM ejecuta herramienta de búsqueda -> LLM lee el resultado -> LLM responde.

El Agente es el bucle que orquesta: **Pensamiento -> Acción -> Observación**.

Un Agente en LangChain utiliza un LLM como motor de razonamiento para determinar qué acciones tomar y en qué orden. A diferencia de una cadena (Chain) que es una secuencia fija de pasos, un Agente observa el input del usuario y decide dinámicamente si debe consultar una API, buscar en una base de datos o ejecutar código Python para resolver el problema, basándose en las descripciones (Docstrings) de las herramientas disponibles.

---

2. Componentes del Código

En el Jupyter y el video se usan tres piezas fundamentales:

A. Las Herramientas (Tools)

Son las "manos" del agente. Funciones que el LLM puede invocar.

- `llm-math`: Una calculadora. Los LLMs son malos haciendo cuentas matemáticas complejas (alucinan números). Esta herramienta traduce texto a código Python/Calculadora para tener precisión exacta.
- `wikipedia`: Conexión a la API de Wikipedia para buscar hechos históricos o definiciones.
- `PythonREPLTool`: Una consola de Python real donde el agente puede escribir y ejecutar código (ideal para ordenar listas, procesar datos, etc.).

B. El Agente (AgentType)

Define la estrategia de razonamiento. En el video usan:

- `CHAT_ZERO_SHOT_REACT_DESCRIPTION`:
  - **Chat:** Optimizado para modelos de chat (como GPT-3.5/4 o Llama 3).
  - **Zero-shot:** No necesita ejemplos previos para saber qué hacer.
  - **ReAct:** Usa la metodología "Reasoning + Acting" (Razonar y Actuar).
  - **Description:** Decide qué herramienta usar basándose exclusivamente en la **descripción de texto** de la herramienta.

C. Output Parser

Como el LLM devuelve texto, necesitamos un parser que detecte si el LLM quiso decir "Ejecutar herramienta X" o "Responder al usuario". El parámetro `handle_parsing_errors=True` es vital por si el LLM devuelve un formato "sucio"; esto le permite al agente intentar corregirse a sí mismo.

---

3. Casos de Uso Explicados

Caso 1: Datos Frescos (Mundial 2022)

El modelo fue entrenado antes del mundial.

- **Pregunta:** "¿Quién ganó el mundial 2022?"
- **Razonamiento del Agente:** "Mi conocimiento interno es viejo. Necesito usar la herramienta `DuckDuckGo` o `Search`".
- **Acción:** Busca en internet.
- **Resultado:** Encuentra "Argentina". Responde al usuario.

Caso 2: El Agente de Python (Programador)

Este es el más potente para nosotros.

- **Input:** Una lista de nombres desordenada `[["Harrison", "Chase"], ["Lang", "Chain"]...]`.
- **Instrucción:** "Ordená estos clientes por apellido y luego por nombre".
- **Acción:** El agente **escribe un script de Python real** usando `lambda x: (x[1], x[0])`, lo ejecuta en el REPL, ve el output y te lo devuelve.
- **Valor:** El LLM no "imagina" el orden, lo **computa** realmente.

---

4. Custom Tools (Tus propias herramientas)

Lo más poderoso es conectar el agente a **tu código**. En el video crean una herramienta para saber la fecha actual (ya que el LLM vive en una burbuja atemporal).

El Decorador `@tool`: Es la forma más fácil de convertir una función Python en una Tool de LangChain.

Python
```
from langchain.agents import tool

@tool
def time(text: str) -> str:
    """Returns todays date, use this for any
    questions related to knowing todays date.
    The input should always be an empty string."""
    return str(date.today())
```

⚠️ **EL SECRETO TÉCNICO (DOCSTRINGS):** Fijate en el comentario entre triples comillas (`"""..."""`). **Ese texto NO es para humanos, es el Prompt para el LLM.** Cuando el Agente tiene que decidir qué herramienta usar, lee esa descripción. Si la descripción es mala, el agente no sabrá cuándo usar tu herramienta. Tenés que ser muy explícito: *"Use this for any questions related to..."*.