**Step-1:**

**%pip install -q -U python-dotenv langchain google-generativeai**

**!pip install langchain-google-genai**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| The above command %pip install -q -U python-dotenv langchain google-generativeai is used in Python (specifically in environments like Jupyter notebooks) to install or upgrade a set of Python libraries.   |  | | --- | | **%pip:** This is a Jupyter notebook (colab) magic command. In colab, you can use the %pip command to install Python packages directly within a notebook.(OR)  %pip: This is a magic command in Jupyter notebooks to run pip commands directly within the notebook environment. Normally, you'd use !pip or pip in a terminal, but %pip is specific to notebooks. |  |  | | --- | | **install:** This tells pip to install the specified packages |  |  | | --- | | **-q:** This stands for "quiet," meaning the installation process will not show a lot of output, making it less verbose. |  |  | | --- | | **-U:** This flag means "upgrade." If the package is already installed, pip will upgrade it to the latest version. |  |  | | --- | | **python-dotenv:** This package is used to manage environment variables in Python projects. It allows you to read .env files, which typically store sensitive data like API keys or database credentials. |  |  | | --- | | **langchain:** This is a framework designed for developing applications powered by large language models (LLMs). It helps build and integrate applications with LLMs like GPT. |  |  | | --- | | **google-generativeai:** This package provides APIs and tools to interact with Google's generative AI services, such as those used for creating text, images, or other content based on input data. |   **summary,** this command installs or upgrades the python-dotenv, langchain, and google-generativeai packages in a Python environment, with minimal output. |

**Step-2:**

**import os**

**from dotenv import load\_dotenv**

**from google.colab import userdata**

**os.environ['GOOGLE\_API\_KEY'] = userdata.get('GOOGLE\_API\_KEY')**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| This Python code is **used to** **load** an environment variable **from** a user's local environment (specifically in Google Colab) and set it into the environment for use within the program.  **1---Importing Libraries:**   |  | | --- | | **import os:** This imports Python's built-in os module, which allows interaction with the operating system, including manipulating environment variables. |  |  | | --- | | **from dotenv import load\_dotenv:** This imports load\_dotenv from the dotenv module, which is a Python package commonly used to read environment variables from a .env file. This is typically used for securely managing secrets like API keys or database credentials. |  |  | | --- | | **from google.colab import userdata:** This imports the userdata module from Google Colab. The userdata module allows you to store user-specific data, such as API keys or credentials, in a safe way. |   **2---Setting an Environment Variable:**   |  | | --- | | **os.environ['GOOGLE\_API\_KEY'] = userdata.get('GOOGLE\_API\_KEY'):**  This line sets an **environment variable** called **GOOGLE\_API\_KEY** using the value stored in Google Colab's userdata. It retrieves the value of the GOOGLE\_API\_KEY from userdata and stores it in the environment variables using os.environ. |   **3--Other Points:**  **Purpose:**  The purpose of this code is likely to **securely load an API key** (GOOGLE\_API\_KEY) from a user's personal or environment-specific storage (in this case, Google Colab) and then make it available as an environment variable within the code. This would allow other parts of the code or libraries (like Google Cloud SDKs) to use this key for authentication without hardcoding it into the code itself.   |  | | --- | | **Potential issues or things to clarify:**  userdata.get method: userdata does not have a get method in the typical Google Colab environment. we might need to use something like os.getenv() or manually handle the environment variable retrieval.  load\_dotenv: we imported this but didn't use it. If you intend to load environment variables from a .env file, we'd need to call load\_dotenv() at the start of your script.  **to load an API key from an environment variable or .env file in Colab, here's how you might do it correctly:**  import os  from dotenv import load\_dotenv  # Load environment variables from a .env file  load\_dotenv()  # Access the environment variable  google\_api\_key = os.getenv('GOOGLE\_API\_KEY')  # Set the environment variable for the session  os.environ['GOOGLE\_API\_KEY'] = google\_api\_key | |

**Step-3:**

**class Calculator:**

**def calculate(self, expression: str) -> str:**

**try:**

**# Use (Python's eval) to compute the safe result**

**result = eval(expression, {"\_\_builtins\_\_": None}, {})**

**return str(result)**

**except Exception as e:**

**return f"Error: {e}"**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | This code defines a Calculator class with a calculate method that takes an expression as a string, evaluates it, and returns the result as a string.   |  | | --- | | **1-- Class:**  **class Calculator:**   * This defines a class called Calculator. |  |  | | --- | | **2--Method:**  **def calculate(self, expression: str) -> str:**   * The calculate method accepts one argument, expression, which is expected to be a mathematical expression in string form (like "2 + 3 \* 5"). * The method is supposed to return a string (-> str). |  |  |  | | --- | --- | | **Using eval:**  The method uses Python's **built-in eval() function** **to evaluate the expression**.   |  | | --- | | **Note:**  **eval()** is powerful **but** potentially dangerous **if** used with **untrusted** input, **as it can** execute **arbitrary code**(an attacker's ability). |   **In this case**, eval is used **with restrictions**. The **second argument** {"\_\_builtins\_\_": None} **disables access** to built-in functions and variables (like open, os, etc.), **making** the evaluation safer. The **third argument** is an empty dictionary {}, which means no external variables are passed to the expression. |  |  | | --- | | **Error Handling:**  If any exception occurs during the evaluation (**e.g., if the expression is invalid**🡪see example blow), **the except Exception as e** block catches it and returns a string **"Error: {e}"** where **{e}** is the exception message.  **Example:**  calc = Calculator()  print(calc.calculate("2 + 3 \* 5")) # Output: "17"  print(calc.calculate("10 / 2")) # Output: "5.0"  print(calc.calculate("5 / 0")) # Output: "Error: division by zero"  print(calc.calculate(**"invalid expression"**)) # Output: "Error: invalid syntax" | | |

|  |
| --- |
| **Limitation:**  **Security Concerns:** Even though the **eval() function** is **restricted** **in this case** **by disabling built-ins, eval()** **can still present security risks if mishandled**. **It’s important to carefully review any method that uses eval().** For this specific example, using eval() may not be the safest choice for evaluating user input in many real-world scenarios. |

**Step-4:**

**from langchain.tools import tool**

**# the tool using as decorator**

**@tool**

**def calculator(expression: str) -> str:**

**"""**

**Perform arithmetic calculations.**

**Input: A mathematical expression as a string (e.g., "2 + 2").**

**Output: Result of the calculation as a string.**

**"""**

**calc = Calculator()**

**return calc.calculate(expression)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| This code is from LangChain, a library designed to help developers build applications that interact with language models (like GPT). It specifically demonstrates the use of a tool decorator from the langchain.tools module. Let's break down what each part of this code does:   |  | | --- | | **1- Import the tool decorator:**    This imports the tool decorator from LangChain's tools module. This decorator is used to mark functions as tools that can be called and used by a LangChain agent (such as a language model). |  |  | | --- | | **2- The function decorated with @tool:**   * **@tool:**   This decorator turns the calculator function into a tool that can be invoked by a  LangChain agent.   * **calculator function:**   This function takes a mathematical expression (as a string, like "2 + 2") and  returns the result of the calculation as a string.   * Inside the function, it creates an instance of the Calculator class and calls its calculate method to process the input expression. |  |  | | --- | | **3-The Calculator class (assumed):**   * The Calculator class is assumed to have a method called calculate that performs the arithmetic operations. The calculator function is calling calculate with the input expression to evaluate and return the result. |  |  | | --- | | **In essence:**   * The calculator function is an arithmetic tool that can be invoked as part of a LangChain-powered agent. * LangChain tools can be called by the language model itself or by other parts of the chain to perform specific actions, such as calculations in this case.   This tool makes it easy to integrate specialized functionality (like calculations) into a language model-powered workflow.  If you'd like to explore this further or have more specific questions about LangChain, feel free to ask! | |

**Step-5:**

**from langchain\_google\_genai import ChatGoogleGenerativeAI**

**from langchain.chains import ConversationChain**

**from langchain.agents import initialize\_agent, Tool, AgentType, AgentExecutor**

**# Initializing Gemini model**

**gemini\_model = ChatGoogleGenerativeAI(**

**model="gemini-2.0-flash", api\_key=os.environ["GOOGLE\_API\_KEY"])**

**# Defined tools**

**tools = [calculator]**

**# ToolAgent for calculator**

**agent = initialize\_agent(tools, gemini\_model, agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION)**

**executor = AgentExecutor.from\_agent\_and\_tools(agent, tools)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| The code is related to **using LangChain** to interact with **Google's generative AI model** (as hare Gemini), with a particular setup for a calculator tool.   |  |  | | --- | --- | | **1-Imports:**  **i-from langchain\_google\_genai import ChatGoogleGenerativeAI**  **ii-from langchain.chains import ConversationChain**  **iii-from langchain.agents import initialize\_agent, Tool, AgentType,**  **i-ChatGoogleGenerativeAI:**  This is a class that likely wraps Google's Gemini generative AI for chat-based interactions, allowing you to query or interact **with** Google's AI models (such as here Gemini 2.0).  **ii-ConversationChain:**  A class used to handle conversational contexts, **allowing the system** to maintain state and context **throughout** a conversation.  **iii-initialize\_agent, Tool, AgentType, AgentExecutor:**  There are **four(4) distinct classes**.These classes are **part of** LangChain's agent framework, **which facilitates** creating agents **that** **can** perform tasks **using** a set of tools.  **Explanation:**   |  | | --- | | **1-initialize\_agent:**   * This is likely a function, not a class, based on typical naming conventions in Python.   **2-Tool:**   * This seems like a class, possibly used to define tools that the agent can interact with.   **3-AgentType:**   * This is likely an enumeration or class to define different types of agents.   **4-AgentExecutor:**   * This appears to be a class that might be responsible for running or executing the agent's tasks.   So, the actual classes are Tool, AgentType, and AgentExecutor. The function is initialize\_agent. | |   **gemini\_model = ChatGoogleGenerativeAI(**  **model="gemini-2.0-flash", api\_key=os.environ["GOOGLE\_API\_KEY"])**   |  | | --- | | **2-Gemini Model Initialization:**   * The code initializes the ChatGoogleGenerativeAI model with the name "gemini-2.0-flash" and an API key (os.environ["GOOGLE\_API\_KEY"]). This is the AI model used to generate responses based on the input provided. |   **tools = [calculator]**   |  | | --- | | **3-** **Defining Tools:**   * The code mentions **tools = [calculator]**. This is likely a predefined tool (such as a calculator), which **allows the agen**t to **perform** calculations **or** handle specific tasks **like math**. |   **agent = initialize\_agent(tools, gemini\_model, agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION)**   |  | | --- | | **4-Agent Initialization:**   * The **initialize\_agent** function sets up the agent with the tools defined **(tools)** and associates it with the generative model **(gemini\_model)**. * The agent type specified here is AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION, which suggests that the agent is designed to handle tasks without needing prior training on specific instructions, responding reactively based on its description. |   **executor = AgentExecutor.from\_agent\_and\_tools(agent, tools)**   |  |  | | --- | --- | | **5-Executing the Agent:**   * **AgentExecutor**   AgentExecutor is a class in the langchain library, specifically designed to handle the execution of an agent with its tools. The from\_agent\_and\_tools method is used to instantiate an AgentExecutor object by passing an agent and tools. This method is likely a factory method that creates an instance of the AgentExecutor class, which manages the process of executing the agent with its provided tools.   |  | | --- | | **Additional Explanation:**  **Class:** A class is a blueprint for creating objects. In this case, AgentExecutor is a class that wraps the functionality of running an agent, along with the tools it needs, to complete tasks.  **Object:** When we call **AgentExecutor.from\_agent\_and\_tools**(agent, tools), we're creating an instance (object) of the **AgentExecutor class**. The executor variable is the reference to that instance. |  * **AgentExecutor.from\_agent\_and\_tools:**   This creates an executor **from** the agent and tools. It will manage the flow of execution **where** the agent interacts **with** the tools **to perform tasks** and **respond** to requests. | |

|  |  |  |
| --- | --- | --- |
| |  | | --- | | **Purpose of the Code:**  This is essentially setting up a **conversational AI agent** that uses **Google's Gemini model** to process queries **and** utilize the calculator tool (or any other tools we may add). The agent is initialized to respond **in a zero-shot manner**, meaning it doesn't require prior training or specific task examples—rather, it interprets requests and uses the defined tools to respond or complete tasks. |  |  | | --- | | **Example Usage:**  This could be used for building a chatbot that can answer questions and perform calculations using the calculator tool.  If we expand this code to include more tools, the agent could potentially access databases, perform web searches, or integrate with other APIs.(I will show this code soon). | |

**Step-6:**

**from langchain.memory import ConversationBufferMemory**

**# Set up memory for conversation context**

**memory = ConversationBufferMemory()**

**# Create conversational chain**

**chain = ConversationChain(**

**llm=gemini\_model,**

**memory=memory**

**)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| This code is part of a setup for a conversational AI system using LangChain, a framework designed to simplify working with language models (LLMs) like GPT.   |  | | --- | | **from langchain.memory import ConversationBufferMemory:**   * This **imports** the ConversationBufferMemory **class** **from** LangChain’s memory module. ConversationBufferMemory is a memory store **that** holds the conversation history (inputs and outputs). It is **useful** for maintaining context in an ongoing conversation with a language model.   **Key words:**  **ConversationBufferMemory**   * Class * Import from LangChain’s memory * memory store * holds the conversation history |  |  | | --- | | **memory = ConversationBufferMemory():**   * This line **creates an instance** of ConversationBufferMemory, which will **automatically (keep track of)** the conversation history as it progresses. It’s used to store past exchanges **between** the user **and** the model **to ensure that** the **model can respond appropriately based on previous context**. |  |  | | --- | | **chain = ConversationChain(...):**   * ConversationChain is a **chain object** that **connects** a language model (in this case, gemini\_model) **with memory** (memory) to **facilitate** a conversational flow. **This chain ensures that** the memory is **utilized** during the conversation **to maintain context**. When you input a query, **it takes into account** the **entire** conversation history **stored in** memory to **generate** responses. |  |  | | --- | | **llm=gemini\_model:**   * This sets the language model to be used for generating responses. gemini\_model here is presumably a pre-defined LLM that we've chosen to use for the conversation. |   This setup is useful for scenarios like chatbots, virtual assistants, or any application where we need to keep track of previous messages to provide coherent responses in a conversation. |

**Step-7:**

**# User query**

**query = "What is 20 divided by 5?"**

**# gets response from the system**

**response = chain.run(query)**

**print(response)**

|  |  |  |  |
| --- | --- | --- | --- |
| This code is demonstrates how a query is processed and responded to using a system, likely a language model or AI chain.   |  | | --- | | **User query:**   * **The variable query stores the question:**   "What is 20 divided by 5?" |  |  | | --- | | **Get response from the system:**   * The chain.run(query) is a function call to run the query through a system (possibly an AI model or workflow), and **it will return an answer based on the query**. The result is stored in the variable response. |  |  | | --- | | **Print response:**   * The print(response) line displays the system's response to the query. |   **In this case,** if executed, the response would likely be the result of the division: 4. |

**Step-8:**

**queries = [**

**"What is 50 divide by 5?",**

**"Now divide the result by 5.",**

**"Add 49 to that."**

**]**

**for q in queries:**

**print("Query:", q)**

**print("Response:", chain.run(q))**

**print("-" \* 40)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| This is a Python code that runs a series of queries using a loop.   |  | | --- | | **Defines a List of Queries:**  The variable queries is a list of string questions (queries), each representing a mathematical operation. These queries seem to describe sequential arithmetic steps.   * "What is 50 multiplied by 5?" * "Now divide the result by 5." * "Add 49 to that |  |  | | --- | | **Loop Through Queries:**   * A for loop iterates over each query in the list queries. |  |  | | --- | | **Printing the Query and Response:**  **For each query:**   * It first prints the query itself using print("Query:", q). * Then, it calls the chain.run(q) function to get a response based on the query and prints that using print("Response:", chain.run(q)). (Note: chain.run(q) might refer to an execution context or a function that processes the query and returns the result, possibly in the context of a chain of operations or a chatbot interface.) * It separates each query and response pair with a line of dashes (print("-" \* 40)) |  |  | | --- | | **Assumption about chain.run(q):**   * The part chain.run(q) suggests that there is a chain or a model running each query, possibly a reference to some sort of AI, mathematical tool, or system that can interpret and compute responses. Without further context, it is unclear if chain refers to a pre-existing object, a custom class, or some specific library designed to handle these operations. |   **Output Example:**  Query: What is 50 divide by 5?  Response: 10  --------------------------------------------------  Query: Now divide the result by 5.  Response: 2  --------------------------------------------------  Query: Add 49 to that.  Response: 51 |