Introduction

This workshop is designed to teach you about the Azure AI Agents Service and the associated SDK. It consists of multiple parts, each highlighting a specific feature of the Azure AI Agents Service. They are meant to be completed in order, as each one builds on the knowledge and work from the previous part.

Prerequisites

1. Access to an Azure subscription. If you don't have an Azure subscription, create a free account before you begin.
2. You need a GitHub account. If you don't have one, create it at GitHub.

Open the Workshop

We will run this lab using GitHub Codespaces. This option provides a pre-configured environment with all the tools and resources needed to complete the workshop.

Select **Open in GitHub Codespaces** to open the project in GitHub Codespaces.

Open in GitHub Codespaces

!!! Warning "Building the Codespace will take several minutes. You can continue reading the instructions while it builds."

Authenticate with Azure

You need to authenticate with Azure so the agent app can access the Azure AI Agents Service and models. Follow these steps:

1. Ensure the Codespace has been created.
2. In the Codespace, open a new terminal window by selecting **Terminal** > **New Terminal** from the **VS Code menu**.
3. Run the following command to authenticate with Azure:

   ```
   az login --use-device-code
   ```

   !!! note You'll be prompted to open a browser link and log in to your Azure account. Be sure to copy the authentication code first. 1. A browser window will open automatically, select your account type and click **Next**. 2. Sign in with your Azure subscription **Username** and **Password**. 3. **Paste** the authentication code. 4. Select **OK**, then **Done**. !!! warning If you have multiple Azure tenants, then you will need to select the appropriate tenant when authenticating. `shell az login --use-device-code --tenant <tenant_id>`
4. Next, select the appropriate subscription from the command line.
5. Leave the terminal window open for the next steps.

Deploy the Azure Resources

The following resources will be created in the `rg-contoso-agent-workshop` resource group in your Azure subscription.

- An **Azure AI Foundry hub** named **agent-wksp**
- An **Azure AI Foundry project** named **Agent Service Workshop**
- A **Serverless (pay-as-you-go) GPT-4o model deployment** named **gpt-4o-mini (Global 2024-07-18)**. See pricing details here.

!!! warning "If you get quota issues, review your quota availability in the AI Foundry Management Center."

We have provided a bash script to automate the deployment of the resources required for the workshop.

The script `deploy.sh` deploys to the `eastus2` region by default; edit the file to change the region or resource names. To run the script, open the VS Code terminal and run the following command:

```
cd infra && ./deploy.sh
```

Workshop Configuration

The deploy script generates the **.env** file, which contains the project connection string, model deployment name. The Bing connection name is leftover from the original lab, but Grounding with Bing unfortunately cannot be used with **gpt-4o-mini**. If you enjoyed this lab and have more available quota, try out the original lab to see Grounding with Bing!

Your **.env** file should look similar to this but with your project connection string.

```
 MODEL_DEPLOYMENT_NAME="gpt-4o-mini"
BING_CONNECTION_NAME="groundingwithbingsearch"
PROJECT_CONNECTION_STRING="<your_project_connection_string>"
```

Project Structure

Be sure to familiarize yourself with the key **subfolders** and **files** you'll be working with throughout the workshop.

The workshop folder

- The **main.py** file: The entry point for the app, containing its main logic.
- The **sales_data.py** file: The function logic to execute dynamic SQL queries against the SQLite database.
- The **stream_event_handler.py** file: Contains the event handler logic for token streaming.

The shared folder

- The **files** folder: Contains the files created by the agent app.
- The **fonts** folder: Contains the multilingual fonts used by Code Interpreter.
- The **instructions** folder: Contains the instructions passed to the LLM.

PART 1

Introduction

What is Function Calling

Function calling enables Large Language Models to interact with external systems. The LLM determines when to invoke a function based on instructions, function definitions, and user prompts. The LLM then returns structured data that can be used by the agent app to invoke a function.

It's up to the developer to implement the function logic within the agent app. In this workshop, we use function logic to execute SQLite queries that are dynamically generated by the LLM.

Enabling Function Calling

If you're familiar with Azure OpenAI Function Calling (https://learn.microsoft.com/azure/ai-services/openai/how-to/function-calling), you know it requires you to define a function schema for the LLM.

With the Azure AI Agent Service and its Python SDK, you can define the function schema directly within the Python function's docstring. This approach keeps the definition and implementation together, simplifying maintenance and enhancing readability.

For example, in the **sales_data.py** file, the **async_fetch_sales_data_using_sqlite_query** function uses a docstring to specify its signature, inputs, and outputs. The SDK parses this docstring to generate the callable function for the LLM:

```
async def async_fetch_sales_data_using_sqlite_query(self: "SalesData", sqlite_query: str) -> str:
    """
    This function is used to answer user questions about Contoso sales data by executing SQLite queries against the database.

    :param sqlite_query: The input should be a well-formed SQLite query to extract information based on the user's question. The que
    :return: Return data in JSON serializable format.
    :rtype: str
    """
```

Dynamic SQL Generation

When the app starts, it incorporates the database schema and key data into the instructions for the Azure AI Agent Service. Using this input, the LLM generates SQLite-compatible SQL queries to respond to user requests expressed in natural language.

Lab Exercise

In this lab, you will enable the function logic to execute dynamic SQL queries against the SQLite database. The function is called by the LLM to answer user questions about Contoso sales data.

1. Open the `main.py`.
2. **Uncomment** the following lines by removing the **"# "** characters

   ```
   # INSTRUCTIONS_FILE = "instructions/instructions_function_calling.txt"

   # toolset.add(functions)
   ```

   !!! warning The lines to be uncommented are not adjacent. When removing the # character, ensure you also delete the space that follows it.

3. Review the Code in main.py.

   After uncommenting, your code should look like this:

```
INSTRUCTIONS_FILE = "instructions/function_calling.txt"
# INSTRUCTIONS_FILE = "instructions/file_search.txt"
# INSTRUCTIONS_FILE = "instructions/code_interpreter.txt"
# INSTRUCTIONS_FILE = "instructions/code_interpreter_multilingual.txt"

async def add_agent_tools() -> None:
    """Add tools for the agent."""
    font_file_info = None

    # Add the functions tool
    toolset.add(functions)

    # Add the tents data sheet to a new vector data store
    # vector_store = await utilities.create_vector_store(
    #     project_client,
    #     files=[TENTS_DATA_SHEET_FILE],
    #     vector_store_name="Contoso Product Information Vector Store",
    # )
    # file_search_tool = FileSearchTool(vector_store_ids=[vector_store.id])
    # toolset.add(file_search_tool)

    # Add the code interpreter tool
    # code_interpreter = CodeInterpreterTool()
    # toolset.add(code_interpreter)

    # Add multilingual support to the code interpreter
    # font_file_info = await utilities.upload_file(project_client, utilities.shared_files_path / FONTS_ZIP)
    # code_interpreter.add_file(file_id=font_file_info.id)

    return font_file_info
```

Review the Instructions

Note that there is nothing you need to do in this section, but just review the resources and instructions to understand what the agent is doing.

1. Open the **shared/instructions/function_calling.txt** file.

   !!! tip "In VS Code, press Alt + Z (Windows/Linux) or Option + Z (Mac) to enable word wrap mode, making the instructions easier to read."

2. Review how the instructions define the agent app's behavior:

   - **Role definition**: The agent assists Contoso users with sales data inquiries in a polite, professional, and friendly manner.
   - **Context**: Contoso is an online retailer specializing in camping and sports gear.
   - **Tool description – "Sales Data Assistance"**:
     - Enables the agent to generate and run SQL queries.
     - Includes database schema details for query building.
     - Limits results to aggregated data with a maximum of 30 rows.
     - Formats output as Markdown tables.
   - **Response guidance**: Emphasizes actionable, relevant replies.
   - **User support tips**: Provides suggestions for assisting users.
   - **Safety and conduct**: Covers how to handle unclear, out-of-scope, or malicious queries.

   During the workshop, we'll extend these instructions by introducing new tools to enhance the agent's capabilities.

   !!! info The placeholder in the instructions is replaced with the database schema when the app initializes.

~~~
```python
# Replace the placeholder with the database schema string
instructions = instructions.replace("{database_schema_string}", database_schema_string)
```
~~~

Run the Agent App

1. Press `F5` to run the app, or right click on `main.py` and select **Run Python file in Terminal**.

2. In the terminal, you'll see the app start, and the agent app will prompt you to **Enter your query**.

Start a Conversation with the Agent

Start asking questions about Contoso sales data. For example:

1. **Help**

   Here is an example of the LLM response to the **help** query:

   I'm here to help with your sales data inquiries at Contoso. Could you please provide more details about what you need assistance with? Here are some example queries you might consider:

   - *What were the sales by region?*
   - *What was last quarter's revenue?*
   - *Which products sell best in Europe?*
   - *Total shipping costs by region?*

   Feel free to ask any specific questions related to Contoso sales data!

   !!! tip The LLM will provide a list of starter questions that were defined in the instructions file. Try asking for help in your language, for example `help in Hindi`, `help in Italian`, or `help in Korean`.

2. **Show the 3 most recent transaction details**

   In the response you can see the raw data stored in the SQLite database. Each record is a single sales transaction for Contoso, with information about the product, product category, sale amount and region, date, and much more.

   !!! warning The agent may refuse to respond to this query with a message like "I'm unable to provide individual transaction details". This is because the instructions direct it to "provide aggregated results by default". If this happens, try again, or reword your query.

   ```
   Large Language models have non-deterministic behavior, and may give different responses even if you repeat the same query.
   ```

3. **What are the sales by region?**

   Here is an example of the LLM response to the **sales by region** query:

   ```
   | Region         | Total Revenue  |
   |----------------|----------------|
   | AFRICA         | $5,227,467     |
   | ASIA-PACIFIC   | $5,363,718     |
   | CHINA          | $10,540,412    |
   | EUROPE         | $9,990,708     |
   | LATIN AMERICA  | $5,386,552     |
   | MIDDLE EAST    | $5,312,519     |
   | NORTH AMERICA  | $15,986,462    |
   ```

   !!! info So, what's happening behind the scenes to make it all work?

   ```
   The LLM orchestrates the following steps:

   1. The LLM generates a SQL query to answer the user's question. For the question **"What are the sales by region?"**, the foll

       **SELECT region, SUM(revenue) AS total_revenue FROM sales_data GROUP BY region;**

   2. The LLM then asks the agent app to call the **async_fetch_sales_data_using_sqlite_query** function, which retrieves the req
   3. The LLM uses the retrieved data to create a Markdown table, which it then returns to the user. Check the instructions file,
   ```

4. **Show sales by category in Europe**

In this case, an even more complex SQL query is run by the agent app.

5. **Breakout sales by footwear**

   Notice how the agent figures out which products fit under the "footwear" category and understands the intent behind the term "**breakout**".

6. **Show sales by region as a pie chart**

   Our agent can't create charts ... yet. We'll fix that in the next lab.

7. **Stop the Agent App**

   When you're done, type **exit** to clean up the agent resources and stop the app.

(Optional) Debug the App

Set a breakpoint (https://code.visualstudio.com/Docs/editor/debugging) in the `async_fetch_sales_data_using_sqlite_query` function located in `sales_data.py` to observe how the LLM requests data.

!!! info "Note: To use the debug feature, exit the previous run. Then set the breakpoint. Then run the application using the debugger icon in the sidebar. This will open up the debug sidebar, allowing you to watch stack traces and step through execution."

Ask More Questions

Now that you've set a breakpoint, ask additional questions about Contoso sales data to observe the function logic in action. Step through the function to execute the database query and return the results to the LLM.

Try these questions:

1. **What regions have the highest sales?**
2. **What were the sales of tents in the United States in April 2022?**

Disable the Breakpoint

Remember to disable the breakpoint before running the app again.

Stop the Agent App

When you're done, type **exit** to clean up the agent resources and stop the app.

PART 2

Introduction

Grounding a conversation with documents is highly effective, especially for retrieving product details that may not be available in an operational database. The Azure AI Agent Service includes a File Search tool (https://learn.microsoft.com/en-us/azure/ai-services/agents/how-to/tools/file-search), which enables agents to retrieve information directly from uploaded files, such as user-supplied documents or product data, enabling a RAG-style (https://learn.microsoft.com/azure/ai-studio/concepts/retrieval-augmented-generation) search experience.

In this lab, you'll learn how to enable the document search and upload the Tents Data Sheet to a vector store for the agent. Once activated, the tool allows the agent to search the file and deliver relevant responses. Documents can be uploaded to the agent for all users or linked to a specific user thread, or linked to the Code Interpreter.

When the app starts, a vector store is created, the Contoso tents datasheet PDF file is uploaded to the vector store, and it is made available to the agent.

Normally, you wouldn't create a new vector store and upload documents each time the app starts. Instead, you'd create the vector store once, upload potentially thousands of documents, and connect the store to the agent.

A vector store (https://en.wikipedia.org/wiki/Vector_database) is a database optimized for storing and searching vectors (numeric representations of text data). The File Search tool uses the vector store for semantic search (https://en.wikipedia.org/wiki/Semantic_search) to search for relevant information in the uploaded document.

Lab Exercise

1. Open the **shared/datasheet/contoso-tents-datasheet.pdf** file. The PDF file includes detailed product descriptions for the tents sold by Contoso.

2. **Review** the file's contents to understand the information it contains, as this will be used to ground the agent's responses.

3. Open the file `main.py`.

4. **Uncomment** the following lines by removing the **"# "** characters

```
# INSTRUCTIONS_FILE = "instructions/file_search.txt"

# vector_store = await utilities.create_vector_store(
#     project_client,
#     files=[TENTS_DATA_SHEET_FILE],
#     vector_name_name="Contoso Product Information Vector Store",
# )
# file_search_tool = FileSearchTool(vector_store_ids=[vector_store.id])
# toolset.add(file_search_tool)
```

!!! warning The lines to be uncommented are not adjacent. When removing the # character, ensure you also delete the space that follows it.

3. Review the code in the `main.py` file.

After uncommenting, your code should look like this:

```
INSTRUCTIONS_FILE = "instructions/function_calling.txt"
INSTRUCTIONS_FILE = "instructions/file_search.txt"
# INSTRUCTIONS_FILE = "instructions/code_interpreter.txt"
# INSTRUCTIONS_FILE = "instructions/code_interpreter_multilingual.txt"

async def add_agent_tools() -> None:
    """Add tools for the agent."""
    font_file_info = None

    # Add the functions tool
    toolset.add(functions)

    # Add the tents data sheet to a new vector data store
    vector_store = await utilities.create_vector_store(
        project_client,
        files=[TENTS_DATA_SHEET_FILE],
        vector_store_name="Contoso Product Information Vector Store",
    )
    file_search_tool = FileSearchTool(vector_store_ids=[vector_store.id])
    toolset.add(file_search_tool)

    # Add the code interpreter tool
    # code_interpreter = CodeInterpreterTool()
    # toolset.add(code_interpreter)

    # Add multilingual support to the code interpreter
    # font_file_info = await utilities.upload_file(project_client, utilities.shared_files_path / FONTS_ZIP)
    # code_interpreter.add_file(file_id=font_file_info.id)

    return font_file_info
```

Review the Instructions

1. Review the **create_vector_store** function in the **utilities.py** file. The create_vector_store function uploads the Tents Data Sheet and saves it in a vector store.

   If you are comfortable using the VS Code debugger, then set a breakpoint (https://code.visualstudio.com/Docs/editor/debugging) in the **create_vector_store** function to observe how the vector store is created.

Run the Agent App

1. Press `F5` to run the app, or right click on `main.py` and select **Run Python file in Terminal**.
2. In the terminal, the app starts, and the agent app will prompt you to **Enter your query**.

Start a Conversation with the Agent

The following conversation uses data from both the Contoso sales database and the uploaded Tents Data Sheet, so the results will vary depending on the query.

1. **What brands of hiking shoes do we sell?**

   !!! info We haven't provided the agent with any files containing information about hiking shoes.

   ```
   In the first lab you may have noticed that the transaction history from the underlying database did not include any product br
   ```

2. **What brands of tents do we sell?**

   The agent responds with a list of distinct tent brands mentioned in the Tents Data Sheet.

   !!! info The agent can now reference the provided data sheet to access details such as brand, description, product type, and category, and relate this data back to the Contoso sales database.

3. **What product type and categories are these brands associated with?**

   The agent provides a list of product types and categories associated with the tent brands.

4. **What were the sales of tents in 2024 by product type? Include the brands associated with each.**

!!! info It's possible the agent might get this wrong, and suggest incorrectly that AlpineGear has a Family Camping tent. To address this, you could provide further context in the instructions or the datasheet, or provide context to the agent directly as in next prompt.

5. **What were the sales of AlpineGear in 2024 by region?**

   The agent responds with sales data from the Contoso sales database.

   !!! info The agent interprets this as a request to find all sales of tents in the "CAMPING & HIKING' category, since it now has access to information that Alpine Gear is a brand of backpacking tent.

6. **Contoso does not sell Family Camping tents from AlpineGear. Try again.**

   That's better!

Stop the Agent App

When you're done, type **exit** to clean up the agent resources and stop the app.

PART 3

Introduction

The Azure AI Agent Service Code Interpreter enables the LLM to safely execute Python code for tasks such as creating charts or performing complex data analyses based on user queries. It makes use of natural language processing (NLP), sales data from an SQLite database, and user prompts to automate code generation. The LLM-generated Python code executes within a secure sandbox environment, running on a restricted subset of Python to ensure safe and controlled execution.

Lab Exercise

In this lab, you'll enable the Code Interpreter to execute Python code generated by the LLM.

1. Open the `main.py`.
2. Define a new instructions file for our agent: **uncomment** the following lines by removing the **"# "** characters

   ```
   # INSTRUCTIONS_FILE = "instructions/code_interpreter.txt

   # code_interpreter = CodeInterpreterTool()
   # toolset.add(code_interpreter)
   ```

   !!! warning The lines to be uncommented are not adjacent. When removing the # character, ensure you also delete the space that follows it.

3. Review the code in the `main.py` file.

   After uncommenting, your code should look like this:

   ```
   INSTRUCTIONS_FILE = "instructions/function_calling.txt"
   INSTRUCTIONS_FILE = "instructions/file_search.txt"
   INSTRUCTIONS_FILE = "instructions/code_interpreter.txt"
   # INSTRUCTIONS_FILE = "instructions/code_interpreter_multilingual.txt"

   async def add_agent_tools() -> None:
       """Add tools for the agent."""
       font_file_info = None

       # Add the functions tool
       toolset.add(functions)

       # Add the tents data sheet to a new vector data store
       vector_store = await utilities.create_vector_store(
           project_client,
           files=[TENTS_DATA_SHEET_FILE],
           vector_store_name="Contoso Product Information Vector Store",
       )
       file_search_tool = FileSearchTool(vector_store_ids=[vector_store.id])
       toolset.add(file_search_tool)

       # Add the code interpreter tool
       code_interpreter = CodeInterpreterTool()
       toolset.add(code_interpreter)

       # Add multilingual support to the code interpreter
       # font_file_info = await utilities.upload_file(project_client, utilities.shared_files_path / FONTS_ZIP)
       # code_interpreter.add_file(file_id=font_file_info.id)

       return font_file_info
   ```

Review the Instructions

1. Open the **shared/instructions/code_interpreter.txt** file. This file replaces the instructions used in the previous lab.
2. The **Tools** section now includes a “Visualization and Code Interpretation” capability, allowing the agent to:
   - Use the code interpreter to run LLM generated Python code. (e.g., for downloading or visualizing data).
   - Create charts and graphs, using the user’s language for labels, titles, and other chart text.

- Export visualizations as PNG files and data as CSV files.

Run the Agent App

1. Press `F5` to run the app, or right click on `main.py` and select **Run Python file in Terminal**.
2. In the terminal, the app will start, and the agent app will prompt you to **Enter your query**.

Start a Conversation with the Agent

Try these questions:

1. **Show sales by region as a pie chart**

   Once the task is complete, the pie chart image will be saved in the **shared/files** subfolder. Note that this subfolder is created the first time this task is run, and is never checked into source control.

   Open the folder in the side menu and click on the image file to view it. (Tip: in Codespaces, you can Control-Click the link that the agent outputs in its response to view the file.)

   !!! info This might feel like magic, so what’s happening behind the scenes to make it all work?

   ```
   Azure AI Agent Service orchestrates the following steps:

   1. The LLM generates a SQL query to answer the user's question. In this example, the query is:

       **SELECT region, SUM(revenue) AS total_revenue FROM sales_data GROUP BY region;**

   2. The LLM asks the agent app to call the **async_fetch_sales_data_using_sqlite_query** function. The SQL command is executed,
   3. Using the returned data, the LLM writes Python code to create a Pie Chart.
   4. Finally, the Code Interpreter executes the Python code to generate the chart.
   ```

2. **Download the sales by region data**

   Once the task is complete, check the **shared/files** folder to see the downloaded file.

   !!! info By default, the instructions specify that data downloads in CSV format. You can request other formats, such as JSON or Excel, by including the desired format in your query (e.g., ‘Download as JSON’).

3. **Download as JSON**

   Once the task is complete, check the **shared/files** folder to see the downloaded file.

   !!! info The agent inferred from the conversation which file you wanted to create, even though you didn't explicitly specify it.

4. Continue asking questions about Contoso sales data to see the Code Interpreter in action.

Stop the Agent App

When you're done, type **exit** to clean up the agent resources and stop the app.

PART 4

Introduction

We'll enhance the Code Interpreter by uploading a ZIP file with fonts for multilingual visualizations—just one example of how file uploads (https://learn.microsoft.com/azure/ai-services/agents/how-to/tools/code-interpreter) can extend its functionality.

!!! note The Code Interpreter includes a default set of Latin-based fonts. Since the Code Interpreter runs in a sandboxed Python environment, it can't download fonts directly from the internet.

Lab Exercise

Earlier labs didn't include multilingual support because uploading the required font ZIP file and linking it to the Code Interpreter is time-consuming. In this lab, we'll enable multilingual support by uploading the necessary fonts. You'll also learn some tips on how to guide the Code Interpreter using extended instructions.

Rerun the previous lab

First, we're going to rerun the previous lab so we can see how the Code Interpreter supports multilingual text.

1. Start the agent app by pressing `F5` or right click on `main.py` and select **Run Python file in Terminal**..

2. In the terminal, the app will start, and the agent app will prompt you to **Enter your query**.

3. Try these questions:

    1. `What were the sales by region for 2022`
    2. `In Korean`
    3. `Show as a pie chart`

    ```
    Once the task is complete, the pie chart image will be saved in the **shared/files** subfolder. Review the visualization, and

    ![The image shows korean pie chart without Korean text](media/sales_by_region_2022_pie_chart_korean.png){width=75%}
    ```

4. When you're done, type **exit** to clean up the agent resources and stop the app.

Add Multilingual Font Support

1. Open the `main.py`.

2. Define a new instructions file for our agent: **uncomment** the following lines by removing the **"# "** characters

    ```
    INSTRUCTIONS_FILE = "instructions/code_interpreter_multilingual.txt"

    font_file_info = await utilities.upload_file(project_client, utilities.shared_files_path / FONTS_ZIP)
    code_interpreter.add_file(file_id=font_file_info.id)
    ```

    !!! warning The lines to be uncommented are not adjacent. When removing the # character, ensure you also delete the space that follows it.

3. Review the code in the `main.py` file.

    After uncommenting, your code should look like this:

```
INSTRUCTIONS_FILE = "instructions/function_calling.txt"
INSTRUCTIONS_FILE = "instructions/file_search.txt"
INSTRUCTIONS_FILE = "instructions/code_interpreter.txt"
INSTRUCTIONS_FILE = "instructions/code_interpreter_multilingual.txt"


async def add_agent_tools() -> None:
    """Add tools for the agent."""
    font_file_info = None

    # Add the functions tool
    toolset.add(functions)

    # Add the code interpreter tool
    code_interpreter = CodeInterpreterTool()
    toolset.add(code_interpreter)

    # Add the tents data sheet to a new vector data store
    vector_store = await utilities.create_vector_store(
        project_client,
        files=[TENTS_DATA_SHEET_FILE],
        vector_store_name="Contoso Product Information Vector Store",
    )
    file_search_tool = FileSearchTool(vector_store_ids=[vector_store.id])
    toolset.add(file_search_tool)

    # Add multilingual support to the code interpreter
    font_file_info = await utilities.upload_file(project_client, utilities.shared_files_path / FONTS_ZIP)
    code_interpreter.add_file(file_id=font_file_info.id)

    return font_file_info
```

Review the Instructions

1. Open the **shared/instructions/code_interpreter_multilingual.txt** file. This file replaces the instructions used in the previous lab.

2. The **Tools** section now includes an extended “Visualization and Code Interpretation” section describing how to create visualizations and handle non-Latin languages.

   The following is a summary of the instructions given to the Code Interpreter:

   - **Font Setup for Non-Latin Scripts (e.g., Arabic, Japanese, Korean, Hindi):**
     - On first run, verify if the `/mnt/data/fonts` folder exists. If missing, unzip the font file into this folder.
     - **Available Fonts:**
       - Arabic: `CairoRegular.ttf`
       - Hindi: `NotoSansDevanagariRegular.ttf`
       - Korean: `NanumGothicRegular.ttf`
       - Japanese: `NotoSansJPRegular.ttf`
   - **Font Usage:**
     - Load the font with `matplotlib.font_manager.FontProperties` using the correct path.
     - Apply the font to:
       - `plt.title()` using the `fontproperties` parameter.
       - All labels and text using `textprops={'fontproperties': font_prop}` in functions like `plt.pie()` or `plt.bar_label()`.
     - Ensure all text (labels, titles, legends) is properly encoded, without boxes or question marks.
   - **Visualization Text:**
     - Always translate the data to the requested or inferred language (e.g., Chinese, French, English).
     - Use the appropriate font from `/mnt/data/fonts/fonts` for all chart text (e.g., titles, labels).

Run the Agent App

1. Press `F5` to run the app, or right click on `main.py` and select **Run Python file in Terminal**.
2. In the terminal, the app will start, and the agent app will prompt you to **Enter your query**.

Start a Conversation with the Agent

Try these questions:

1. `What were the sales by region for 2022`
2. `In Korean`
3. `Show as a pie chart`

   Once the task is complete, the pie chart image will be saved in the **shared/files** subfolder.

Debugging the Code Interpreter

You can't directly debug the Code Interpreter, but you can gain insight into its behavior by asking the agent to display the code it generates. This helps you understand how it interprets your instructions and can guide you in refining them.

From the terminal, type:

1. `show code` to see the code generated by the Code Interpreter for the last visualization.
2. `list files mounted at /mnt/data` to see the files uploaded to the Code Interpreter.

Restricting Code Interpreter Output

You likely don't want end users to see the code generated by the Code Interpreter or access uploaded or created files. To prevent this, add instructions to restrict the Code Interpreter from displaying code or listing files.

For example, you can insert the following instructions at the beginning of the `2. Visualization and Code Interpretation` section in the `code_interpreter_multilingual.txt` file.

```
- Never show the code you generate to the user.
- Never list the files mounted at /mnt/data.
```

Stop the Agent App

When you're done, type **exit** to clean up the agent resources and stop the app.