MCP Workshop: Exercise 1

File: 01_no_mcp_langgraph_agent.py

This file demonstrates a basic LangGraph ReAct (Reasoning and Acting) agent that uses local Python functions as tools. The agent has three tools available: add, multiply, and divide. These are simple Python functions that are bound to the LLM, allowing it to call them when needed. This demonstrates the core LangGraph pattern without MCP integration. In later examples, we'll see how MCP servers replace these local functions.

The agent follows a simple flow:
1. Human Question → The user asks a question
2. Assistant → The LLM decides whether to go to end node, or to call the tool(s) node
3. Tool Execution → The selected tool(s) are executed
4. Assistant → The LLM summarizes the tool execution and goes to step 2; go to end node, or call tool(s) node once more.

Exercise 1: Which part(s) of the function (doc string, type hints, function body code, function body comments) output will NOT be included in the MCP JSON format when the tool is sent to the LLM?

Exercise 2: Is the agent limited to using only 1 tool call per question, or can it chain multiple tool calls together? Try asking a question that requires multiple operations, such as: "What is (3 + 4) * 2?" Observe how many tool calls are made.

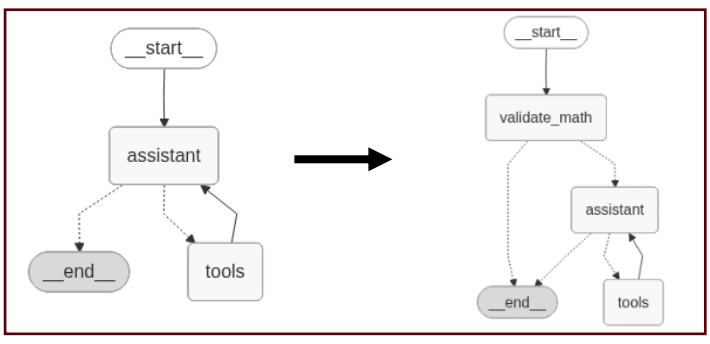
Exercise 3: Uncomment the lines at the bottom of the file. This adds a second question: "What was the first question asked?" Run the code again. How does the ReAct_graph remember the first question? What component enables this memory?

Exercise 4: Make the "add" function return a * 2 instead of a + b. Run the code and observe the result. The tool result will return 6 when called with add (3, 4). What does the LLM say about the result? Is there something you can change to the docstring, or the input query to make the final answer correct?

Exercise 5: How does the graph know which tool to call? Where is this information stored? What problem might arise if we have too many tools available?

Exercise 6: Modify the pretty_print() function to print the entire message "m". What kind of guardrails are being checked by this LLM?

Bonus: Add another node to the graph that checks if the question is about math (addition, multiplication, or division) before proceeding to the assistant. If the question is not about math, bind to the END node.
Steps:
1. Create a node validate_math_question(state: MessageState) that checks if the user question's intent for math
2. Add this as a node in the graph between START and "assistant"
3. Add conditional logic to either proceed to the assistant or return an error message



Find a possible answer with:
Git checkout
exercise-01-bonus-question

MCP Workshop: Exercise 2

File: 02_mcp_stdio_local.py

This is a LangGraph ReAct agent that loads tools from local MCP servers running as subprocesses via stdio transport (found in the local_mcp_servers folder). Unlike example 01, which uses direct Python functions, it uses tools exposed through @mcp.tool() decorators, demonstrating the transition from local tooling to MCP-based servers. These servers are fully compatible with standard MCP clients such as LangChain's MultiServerMCPClient.

Exercise 1: What happens if one of the MCP servers fails to start? How does `validate_servers()` handle this? Try modifying the server config to point to a non-existent file.

Exercise 2: Compare the tools available in file 01.py vs file 02.py. Are they the same? What's the difference in how they're defined and loaded?

Exercise 3: Try asking a question that requires both math and weather tools: "What's 5 * 3 and what's the weather in Paris?" Observe how the agent chains multiple tool calls from different servers.

Exercise 4: What is stdio transport? How does it differ from HTTP transport? When would you choose one over the other?

---

bonus: In file 02, the `assistant` function is defined as `async def assistant(state: MessageState)`. What is the benefit for it to be async? Notice that the llm calls also use `invoke()` instead of `ainvoke()`?

MCP Workshop: Exercise 3

File: 03_mcp_stdio_external_package.py

LangGraph agent combining local MCP servers with external MCP packages (like office-word-mcp-server) via stdio.
This file includes a FastAPI web interface with streaming chat. This example demonstrates how to integrate both locally developed MCP servers and external packages from the MCP ecosystem. User accesses FastAPI web interface at http://localhost:8000/chat or https://127.0.0.1:8000/chat

Exercise 1: What is the danger of using external packages from MCP? What security considerations should you keep in mind when running external MCP servers?

Exercise 2.a: How many tool calls are loaded from the wordmcp server? Check the console output when the server starts and scan the list of all the tools available from the office-word-mcp-server. Do you think the LLM has enough context for when to use what tool

---

Exercise 2.b: How could we guide the LLM to have a better understanding of when to use what tool? Hint: system prompt

---

Exercise 3: Try to make a query that combines the weather server, math server and creates a word document with the answer of your query. Make sure to give the word document a name.docx. For example: "What's the weather in Paris and what's 15 * 7? Create a word document called results.docx with both answers."

Exercise 4: Compare the server configuration for local servers vs external packages. Where is the uv package installed?

Exercise 5: The FastAPI endpoint uses `async def chat_endpoint()`. How does this enable multiple users to use the web interface simultaneously? What would happen if it were synchronous?

Bonus: Can you come up with a way to create a Thesis-style word document (with in-text apa references and a reference list)?
NOTE: you may need to remove the truncate_messages_safely() function as this can require quite some memory

Find a step towards the answer on:
Git checkout
exercise-03-bonus-question

MCP Workshop: Exercise 4

File: 04_mcp_http_external_pacakge.py

This file consists of a LangGraph agent that connects to remote HTTP-based MCP servers (e.g. the Supabase MCP server or the Code Explorer server) rather than local stdio subprocesses. It includes a FastAPI web interface with streaming chat. The agent accesses a Supabase database through a remote MCP server hosted on Railway (mcp-workshop-server.up.railway.app) and can interact with the connected application at vibify.up.railway.app via MCP.
Using streamable_http provides several benefits:

- Credentials (e.g., Supabase database keys) stay securely on the remote MCP server rather than on the client.
- The agent can connect to hosted MCP services without running local processes (no dangerous code on your pc)
- Multiple clients can share the same remote MCP tools.
- Remote servers can scale independently.

Exercise 1: Name a song in the database.

Exercise 2: Who listens to most music?

Exercise 3: What song is streamed the most

Exercise 4: Are there any database integrity issues if a user is deleted?

Exercise 5: Can all songs be shared in a playlist?

Exercise 6: Ask question 3) again, do you get a tool call? Why or why not?

Exercise 7A: Name a new song in the database, and find the genre(s) of this song. (Requires 2+ tool calls)
Exercise 7B: Can you find another song with the same **exactly** same genre(s), how many songs in entire database have these genres? (Requires multiple tool calls and SQL joins)

Exercise 9: Can you show the top playlist owner, three songs from their largest playlist, and each songs stream count? (Requires complex SQL query with joins)

Exercise 10: What is the minimum duration for a song for it to be a "streamed song"?

Exercise 11: Can you increase the stream count of a song by 10?

Exercise 15: Try uncommenting the code-explorer server. What tools does it provide? Ask the agent to list the files in this repository.

Bonus: What kind of MCP server would you like to make? Think about which applications you would like to integrate with a chatbot!