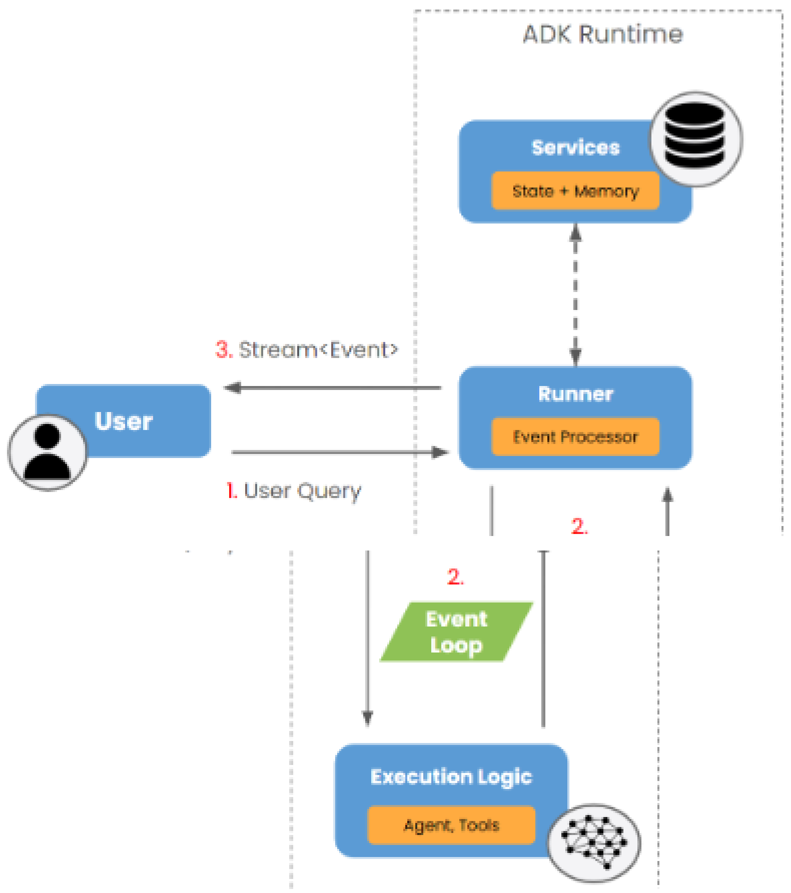

Lesson 3 - Introduction to Google's ADK - Part I

In this lesson, you will familiarize yourself with Google's Agent Development Kit (ADK) that you will use in the next lessons to build your multi-agent system.

You'll learn:

- how to create and run an agent using ADK (Part I)
- how to create a team of Agents consisting of a root agent and 2 sub-agents (Part II)
- how the team of agents can access a sharable context (Part II)

For each agent, you'll define a tool that allows the agent to interact with the Neo4j database we setup for this course.

3.1. Setup

```python
# Import necessary libraries
import os
from google.adk.agents import Agent
from google.adk.models.lite_llm import LiteLlm # For OpenAI support
from google.adk.sessions import InMemorySessionService
from google.adk.runners import Runner
from google.genai import types # For creating message Content/Parts
from typing import Optional, Dict, Any

import warnings
warnings.filterwarnings("ignore")

import logging
logging.basicConfig(level=logging.CRITICAL)

print("Libraries imported.")
# Define Model Constants for easier use
MODEL_GPT = "openai/gpt-4o"

llm = LiteLlm(model=MODEL_GPT)

# Test LLM with a direct call
print(llm.llm_client.completion(model=llm.model,
                               messages=[{"role": "user",
                                          "content": "Are you ready?"}],
                               tools=[]))

print("\nOpenAI is ready for use.")
```

3.2. Explore `neo4j_for_adk`

In your lab environment, you are provided with a graph database that your agent will interact with. For that, you are provided with a helper called `neo4j_for_adk` from which you'll import the instance `graphdb`, which wraps the Neo4j Python driver to make it ADK friendly.

```python
# Convenience libraries for working with Neo4j inside of Google ADK
from neo4j_for_adk import graphdb
```

💻 **To access neo4j_for_adk.py** 1) click on the *"File"* option on the top menu of the notebook and then 2) click on *"Open"*.

`graphdb` has a method `send_query` which expects a cypher query, runs the query and then formats the results as follows:

```python
# Sending a simple query to the database
```

```
neo4j_is_ready = graphdb.send_query("RETURN 'Neo4j is Ready!' as message")

print(neo4j_is_ready)
```

Optional Note: Neo4j Database Setup

We set up the database as a sidecar container. You can find the Docker installation instructions (and others) here. We configured the username and password as part of the database setup. We also installed a plugin called APOC (which will be needed in the last notebook). We defined these environment variables, which are used by `neo4j_for_adk.py`:

- `NEO4J_URI="bolt://localhost:7687"`
- `NEO4J_USERNAME="your_database_username"`
- `NEO4J_PASSWORD="your_database_password"`

3.3. Define your Agent's Tool

```
# Define a basic tool -- send a parameterized cypher query
def say_hello(person_name: str) -> dict:
   """Formats a welcome message to a named person.

   Args:
       person_name (str): the name of the person saying hello

   Returns:
       dict: A dictionary containing the results of the query.
             Includes a 'status' key ('success' or 'error').
             If 'success', includes a 'query_result' key with an array of result
rows.
             If 'error', includes an 'error_message' key.
   """
   return graphdb.send_query("RETURN 'Hello to you, ' + $person_name AS reply",
   {
       "person_name": person_name
   })

# Example tool usage (optional test)
print(say_hello("ABK"))
# Example tool usage (optional test)
print(say_hello("RETURN 'injection attack avoided'"))
```

3.4. Define the Agent `friendly_cypher_agent`

Optional Reading

- An `Agent` in Google ADK orchestrates the interaction between the user, the LLM, and the available tools
- you configure it with several key parameters:

    - `name`: A unique identifier for this agent (e.g., "friendly_cypher_agent_v1").
    - `model`: Specifies which LLM to use. you'll use the `llm` variable we defined above.
    - `description`: A summary of the agent's overall purpose. This is like public documentation that helps other agents decide when to delegate tasks to *this* agent.
    - `instruction`: Detailed guidance given to the LLM on how this agent should behave, its persona, goals, and specifically *how and when* to utilize its assigned `tools`.
    - `tools`: A list containing the actual Python tool functions the agent is allowed to use (e.g., `[say_hello]`).

Best Practice:

- Provide clear and specific `instruction` prompts. The more detailed the instructions, the better the LLM can understand its role and how to use its tools effectively. Be explicit about error handling if needed.
- Choose descriptive `name` and `description` values. These are used internally by ADK and are vital for features like automatic delegation (covered later).

```python
# Define the Cypher Agent
hello_agent = Agent(
   name="hello_agent_v1",
   model=llm, # defined earlier in a variable
   description="Has friendly chats with a user.",
   instruction="""You are a helpful assistant, chatting with a user.
                 Be polite and friendly, introducing yourself and asking who the
user is.

                 If the user provides their name, use the 'say_hello' tool to get
a custom greeting.
                 If the tool returns an error, inform the user politely.
                 If the tool is successful, present the reply.
                 """,
   tools=[say_hello], # Pass the function directly
)

print(f"Agent '{hello_agent.name}' created.")
```

3.5. Run the Agent

To run an agent, you'll need some additional components namely an execution environment and memory.

3.5.1. Event Loop

The ADK Runtime orchestrates agents throughout execution. The main component is an event-driven loop intermediated by a Runner. When the runner receives a user query, it asks the agent to start processing. The agent processes the query and emits an event. The Runner receives the event, records state changes, updates memory and forwards the event to the user interface. After that, the agent's logic resumes and the cycle repeats until no further events are produced by the agent and the user has a response.

3.5.2. Create the Runner and SessionService

Let's assume we have a single user talking to the agent in a single session. Let's create this user, the session and the runner:

- `SessionService`: Responsible for managing conversation history and state for different users and sessions. The `InMemorySessionService` is a simple implementation that stores everything in memory, suitable for testing and simple applications. It keeps track of the messages exchanged.
- `Runner`: The engine that orchestrates the interaction flow. It takes user input, routes it to the appropriate agent, manages calls to the LLM and tools based on the agent's logic, handles session updates via the `SessionService`, and yields events representing the progress of the interaction.

```python
app_name = hello_agent.name + "_app"
user_id = hello_agent.name + "_user"
session_id = hello_agent.name + "_session_01"

# Initialize a session service and a session
session_service = InMemorySessionService()
await session_service.create_session(
   app_name=app_name,
   user_id=user_id,
   session_id=session_id
)

runner = Runner(
   agent=hello_agent,
   app_name=app_name,
   session_service=session_service
)
```

3.5.3. Run the Agent

Here's what's happening:

1. Package the user query into the ADK `Content` format.
2. Call`runner.run_async` (providing it with user/session context and the new message)
3. Iterate through the **Events** yielded by the runner. Events represent steps in the agent's execution (e.g., tool call requested, tool result received, intermediate LLM thought, final response).
4. Identify and print the **final response** event using `event.is_final_response()`.

Why `async`? Interactions with LLMs and potentially tools (like external APIs) are I/O-bound operations. Using `asyncio` allows the program to handle these operations efficiently without blocking execution.

```
user_message = "Hello, I'm ABK"
print(f"\n>>> User Message: {user_message}")

# Prepare the user's message in ADK format
content = types.Content(role='user', parts=[types.Part(text=user_message)])

final_response_text = "Agent did not produce a final response." # Default will
be replaced if the agent produces a final response.


# We iterate through events to find the final answer.
verbose = False
async for event in runner.run_async(user_id=user_id, session_id=session_id,
new_message=content):
   if verbose:
       print(f"  [Event] Author: {event.author}, Type: {type(event).__name__},
Final: {event.is_final_response()}, Content: {event.content}")

   # Key Concept: is_final_response() marks the concluding message for the turn.
   if event.is_final_response():
       if event.content and event.content.parts:
           final_response_text = event.content.parts[0].text # Assuming text
response in the first part
       elif event.actions and event.actions.escalate: # Handle potential
errors/escalations
           final_response_text = f"Agent escalated: {event.error_message or 'No
specific message.'}"
       break # Stop processing events once the final response is found

print(f"<<< Agent Response: {final_response_text}")
```

3.6. Create Helper Class: `AgentCaller`

3.6.1 Set up AgentCaller

Let's wrap the runner in the helper class: `AgentCaller`. This helper will make it easier to make repeated calls to the agent by assuming we have a single user talking to the agent in a single session.

```
class AgentCaller:
   """A simple wrapper class for interacting with an ADK agent."""

   def __init__(self, agent: Agent, runner: Runner,
                user_id: str, session_id: str):
       """Initialize the AgentCaller with required components."""
       self.agent = agent
       self.runner = runner
       self.user_id = user_id
       self.session_id = session_id


   def get_session(self):
       return
self.runner.session_service.get_session(app_name=self.runner.app_name,
user_id=self.user_id, session_id=self.session_id)


   async def call(self, user_message: str, verbose: bool = False):
       """Call the agent with a query and return the response."""
       print(f"\n>>> User Message: {user_message}")

       # Prepare the user's message in ADK format
       content = types.Content(role='user',
parts=[types.Part(text=user_message)])

       final_response_text = "Agent did not produce a final response."

       # Key Concept: run_async executes the agent logic and yields Events.
       # We iterate through events to find the final answer.
       async for event in self.runner.run_async(user_id=self.user_id,
session_id=self.session_id, new_message=content):
           # You can uncomment the line below to see *all* events during
execution
           if verbose:
               print(f"  [Event] Author: {event.author}, Type:
{type(event).__name__}, Final: {event.is_final_response()}, Content:
{event.content}")

           # Key Concept: is_final_response() marks the concluding message for
the turn.
           if event.is_final_response():
               if event.content and event.content.parts:
                   # Assuming text response in the first part
                   final_response_text = event.content.parts[0].text
               elif event.actions and event.actions.escalate: # Handle potential
errors/escalations
                   final_response_text = f"Agent escalated: {event.error_message
or 'No specific message.'}"
               break # Stop processing events once the final response is found

       print(f"<<< Agent Response: {final_response_text}")
       return final_response_text
```

3.6.2 Make an instance of the AgentCaller

Rather than a class constructor, you'll use a factory method which needs to make some async calls to initialize the components (user_id, session_id and runner) before passing to them to the AgentCaller.

The factory method takes some parameters:

- `Agent`: the agent that we defined earlier
- `initial_state`: optional initialization of the agent's "memory"

Inside, the method will create memory for the agent and a runner.

```
async def make_agent_caller(agent: Agent, initial_state: Optional[Dict[str,
Any]] = {}) -> AgentCaller:
    """Create and return an AgentCaller instance for the given agent."""
    app_name = agent.name + "_app"
    user_id = agent.name + "_user"
    session_id = agent.name + "_session_01"

    # Initialize a session service and a session
    session_service = InMemorySessionService()
    await session_service.create_session(
        app_name=app_name,
        user_id=user_id,
        session_id=session_id,
        state=initial_state
    )

    runner = Runner(
        agent=agent,
        app_name=app_name,
        session_service=session_service
    )
```

```
   return AgentCaller(agent, runner, user_id, session_id)
```

3.6.3. Run the Conversation

Now you can define an async function to run the conversation.

```
hello_agent_caller = await make_agent_caller(hello_agent)

# We need an async function to await our interaction helper
async def run_conversation():
   await hello_agent_caller.call("Hello I'm ABK")

   await hello_agent_caller.call("I am excited")

# Execute the conversation using await
await run_conversation()
```

To build a multi-agent system, you'll use Google's ADK or agent development kit, which is a framework for developing agents. In this lesson, you'll learn how to create and run one agent as well as a team of agents. All right, let's go. In this lesson, you will familiarize yourself with Google's agent development kit, and we'll use that for building agents throughout this multi-agent system. In this lesson, you'll learn how to create and run an agent using ADK. The imports that we're doing here, of course is standard OS, and then a bunch of stuff from Google ADK itself. We're going to import this agent, which is going to be the key thing for actually describing what an agent is. For interacting with an LLM, we're going to use the LiteLLM library, so we're going to import that here, and this is a little wrapper for letting Google ADK talk to OpenAI. And then we're also going to need some memory for the agent. For the memory, we're going to use the memory session service, and then finally the agent also needs a way to run, so we're going to import the runner. We also have some typing stuff here that's not really so important for how all this works. Let's go ahead and make sure that all that imports. Great. All the libraries are imported. So the next part then is to start defining the models that we're going to use. We're going to be using OpenAI for this course. So let's go ahead and import, we'll define what the model is going to be itself is OpenAI's GPT-4o. Again, we're going to be using LiteLLM for actually reaching out to the OpenAI API. And we're just kind of going to go ahead and try out a sample execution of completing a chat with that. So if you look at the chat here, you'll notice that we're just setting up a message that we're going to send to OpenAI and we're going to ask it the question, are you ready? So if that all works successfully, we'll get a good message from OpenAI letting us know that this works. Looking at the model response, there's a lot of information here, but the key part is that the response includes a message from OpenAI saying, yes, I'm ready. How can I assist you today? OpenAI is all set to go. We're also using Neo4j today, so we're going to import a helper library that combines Neo4j with Google ADK in a nice way. And that library is something we can take a look at to see the details of what's going on. The import that we're going to use is just from Neo4j for ADK, import the GraphDB library. Let's take a look at those details. Okay, now we're inside Neo4j for ADK. And you can see it starts off pretty simply, we import some stuff from the operating system, we of course load up the environment. And then from Neo4j, the Python package, we load up the graph database and you also get a results class definition. Now a bunch of these functions that are in here are just really nice for just wrapping up how you interact with Neo4j and presenting the results from Neo4j in a way that Google ADK likes to see things. One of the key things is that the results that come back either as a dictionary with a status that is a success or a status that is an error. These two helper functions help do that. You can call tool success or tool error along with some extra arguments and actually get a nicely formatted result from any kind of call. To do that, we then go ahead and also integrate a helper library. Results that are handled by Google ADK can be easily persisted. So this function called here to Python, basically goes through all the different kinds of results you can get back from Neo4j and makes them easily serializable in a nice format for for Google ADK. You can look at the details if you want to see what's going on here. The key part of course is this function that takes the Neo4j result. You're passing in is just a result. Basically calls that to Python package, packages all that up into a result that is easily used by Google ADK. Each of those helper functions is then also used by this upper level class called Neo4j for ADK. And this is really a wrapper that combines the Neo4j driver, initializes the environment variables there, allows access to the driver, and has a special function here for sending queries. And it'll send any kind of query via two, but then uses those helper functions to actually reformat the angles results into a nice Google ADK result. At the end then, because we're just going to have one driver and one connection that we're using, we're going to have a singleton of this class that we're going to use. This is the GraphDB that we're going to actually expose as a variable and use that for all of our notebooks. Okay, we took a look at Neo4j for ADK. Let's continue with the notebook. So now we've got everything set up, we can start to define the agent itself. Okay, with our Neo4j wrapper ready, let's go ahead and give it a try. You can just send in any kind of Neo4j query that you like to, as long as it's valid Cypher. Here the Cypher that we're sending is a very simple Cypher statement that doesn't do any work other than directly return a result. Here the result is a string, Neo4j is ready. Passing that back aliased as a message variable. So if we run this and then print the results, you'll see that it's been packaged up into this nice little dictionary with a status success and the query result being an array of all the rows returned from the database. Great, so OpenAI ready and Neo4j ready, we can start to define our tools themselves and the agent eventually. Tool definition is an important part in actually defining an agent. Without tools, an agent can do a lot of thinking, maybe have some chats, but it can't really do any interactions with the environment or with the world around them. So we're going to define a very simple tool just to illustrate what kind of things an agent can do. Hello world is the classic of course. So we're going to develop the hello world version of for an agent's tool and we're going to call it say hello. So tools can be simply defined as functions. Here the function is just say hello, it takes a single argument, person name as a string, and it returns a dictionary. Of course that dictionary is the Google ADK dictionary that likes to see all results in. And it also has a couple of important parts here. There's a docstring that's here, which is always nice when you're writing code. It's also especially critical when you're developing tools for Google ADK. This docstring is what lets Google ADK understand what actually this tool does. This gets passed along to the LLM and the LLM is told, here's all the tools that are available and also here's what those tools are described as doing. So there's a couple important parts we should go through here. You describe what the tool does. Here it just formats a welcome message to a named person, of course passed in as person name. You describe the arguments that are being passed in and also the result. The result here is consistent for all the tools because the result is a dictionary that Google ADK likes to see. But you can add a little extra detail if you want to here, for instance, if you know the kinds of errors that might come back, you could explain that here as well. Inside, the function is incredibly simple. All we're doing is using Neo4j again to actually do a hello world. We're going to call out to Neo4j, sending a query where we're still going to return hello to you, but then we're also going to concatenate that with this person name, which is passed in as an argument to the function. That's also going to get passed in as a query parameter to this query. Whenever you see that dollar sign in a Neo4j Cypher statement, that means here's a query parameter. So here in this array, in this dictionary that's being passed in, you'll see the person name is being passed in as person name, that value is substituted in as this variable here. Not as a template substitution, but as a value gets passed in, and then this whole query will get executed. So let's define that function. Of course because this function is just a normal function, we can try to run it. So we can print saying hello to ADK and expect to see a very nice Google ADK formatted result that was successful where the query result that was sent off to Neo4j has the reply hello to you ADK, doing that concatenation. Fantastic. Now just a side note here about query parameters, part of why we're doing that, this is good discipline for any kind of query language that you're using. Most query languages have the ability to have query parameters. And one of the reasons you do that is to avoid injection attacks, right? That people can't pass in values that look like SQL statements or Cypher statements or any other query language statement as a value and have that concatenated into a string and cause all kinds of mayhem. So instead you pass in a query parameter. And here for instance, if I pass in the name is actually the name is going to be a return some string here, this could be some malicious code instead that gets passed in as a person's name. If you just concatenated that as a string, all kinds of bad things can happen. Instead because it's a query parameter, that still just behaves as a variable. It gets passed in, it gets concatenated as a string with the other string, and the result is simply hello to the somewhat friendly and malicious injection attack. And of course it's been avoided now. We have a basic tool available. Now we can define an agent that's going to use that tool. Now there are a couple components of defining an agent, it's common across all frameworks, they'll do some variations of the same idea. Let's take a look at what's happening inside of Google ADK to define an agent. Now if you recall earlier, we had imported this agent class from Google ADK. So we're going to make an instance of that class, it takes a bunch of parameters. So the first thing that's important is that every agent is given a name. Here we're going to call it hello_agent_v1. This is in case you have multiple versions of the agent that you want to keep active at the same time or if you want to actually have different version instructions as well, having an idea of what the version of this agent is is very helpful for debugging later on. You also pass in the model that you want to use. If you recall we defined LLM before as being a call through LiteLLM out to OpenAI, so we're going to use that here. And these next two are pretty critical for how the Google ADK actually orchestrates and executes agents. The first part is this description of the agent. This is a little bit like the docstring of the tool, what describes what does this agent do? This lets Google ADK and also the other agents understand what is the purpose of this agent and is there occasions when I should call this agent rather than doing something myself? It's called agent delegation. So that's for other agents to know what to do with the agent here. Then for the agent itself, you provide some instruction. The instruction is basically the same kind of thing you would have done with when you're doing prompt engineering and just applying the system prompt for the LLM. This is the same kind of thing that you'd be doing there. This is a hello world agent, so it is of course very helpful. It's a helpful assistant that's going to chat with the user and it really only has one tool that it wants to use. So you describe that tool here so that the agent understands not just from the tool definition, but also from your instructions that you're giving it, how to think about that tool and when to use that tool. So here if the user provides their name, you know, use the say hello tool to give them a custom greeting. And the final part of course is we have to make those tools available. The tools get passed in as an array of just the function names themselves. So we're going to pass in say hello, and then the agent has access to that tool, it knows what to do with that tool because we gave it instructions, and other agents if there are other agents know what this agent's role is. Now that we have an agent and the agent has a tool and has instructions about how to use that tool, now we need to run the agent. This is the final part of creating a very basic agentic system. An agent has to have an execution environment. So if you look at this diagram here, you'll see that in our box here, this is the execution environment that happens. There's a runner class that actually manages all the event loops basically for calling out to the LLM, passing results from the LLM to particular agents and coordinating all the agents themselves and how they execute. Each of those runners also has access to various services, whether that is in-memory services for like doing actual memory, memory could be in-memory or it could be you could use a database like Neo4j for memory as well. So it has access to storage. Then there's the execution logic that inside the execution loop. Here are the steps for actually running things and coordinating the agent, whether agents are running in parallel or in a sequence or in a loop. This all together actually is the execution environment for the agent. Okay, we're going to do this by hand first, and then show how we can wrap this up through a little convenience method. When you're doing your own development with Google ADK, Google ADK has really great tooling for just defining an agent and using the tool and actually just running that agent by providing the full execution environment that it needs. We're going to be doing that by hand inside of the notebook. So let's do that one step at a time and then wrap it up into a little helper. So there are a couple of things you're going to need to actually set up the execution environment. Every runtime actually runs within a session. You could of course have multiple sessions that are running in some production environment. Here we're going to have a single session where we're going to go ahead and just set up the in-memory service which provides context and state for the agent as it's running. We're going to create a session service based on that memory and also for the particular agent that we're writing, and also assuming that we have a user, we're going to assume that we have a single user, single application that we're running and a single session. All of that is going to be used here in the create session that actually create a single executable state to actually be run. And then the runner is going to run that state given the agent that it has. So here the runner combines the agent that we want to run, our hello world agent, the app name is going to be the same as we passed into the session here, and also the session service is going to be the session within which this agent runs. Again, in a production environment, there might be multiple sessions, there might be multiple applications that are using this agent. That's why these are all different arguments to run. Okay, now there's a lot to walk through just to actually do the running of the agent as well. So let's go through this step-by-step and take our time. With that we just have this runner available that we just defined. And we're going to do just a single loop through with a single user message and allow the runner to execute the agent reacting to that message from the user. So we're defining a user message and I'm simply going to say, hello, my name is ADK. And we're going to go and print that message out in a friendly way so we ourselves can see as this is running what's going on. That message that is just a string has to get packed up into a data structure that Google ADK and the runtime environment expects. Usually this is handled behind the scenes for you of course, but we're going to do this by hand. So we're going to create this content class here, and the role of the content that we're creating is from the user, this is a user message. Content can have multiple parts. So we're going to create an array of parts where one of the parts is some passed in text, where the text is a user message. This is all bundled up to do more complex things. Here we're just trying to run a single message from a single user, but this gives you lots of flexibility when you have lots of other things going on and this is basically creating a content event that then has to get processed by the agent runtime system. Also, we're going to go ahead and set up a response from the agent system in case it doesn't do anything. So after you pass in a message, we're going to actually go through running that message and letting the agent get a chance to respond. It's possible that the agent doesn't respond. There's this notion of having a final response, which is a flag that an agent can set in different ways to actually indicate, okay, I'm done thinking about this, I'm done processing this. So you can go out and go back to the user. In case the agent does nothing, a final response may not get set. So we pre-set it here with a default value to let us know that nothing has happened. We're going to run through without any verbose output right now, and you can see in a second what the verbose output might look like. This is the event loop itself. Well, one step I should say of the event loop. You know, for this one message where given the user message, we're going to asynchronously use the runner and call out to the LLM using the message that was passed in, the agent that was defined to use that model and all the instructions we've given it. So the first thing we're going to notice is that when we call this async, we also pass in all the same information we had for creating the session. This is because you can run things at end to multiple sessions going on to multiple notebooks running around. And asynchronously, you pass in here's the context I'm currently operating in. It's for this user, in this session, and here's the content that's being passed in. Go ahead and execute this once with the LLM to do some work on this. This is inside of a for loop because this is an agentic system. So once the agent is given this message, it might do a bunch of things before actually having a final response. So the event loop here pulls out events from the asynchronous running of the agent. Every event is some update from the agent itself. As the events come in, if the verbose flag is turned on, we'll print out here's an event that happened, here's who created that event. The initial event of course should be that the user message was passed in. So we should see the event author was the user themselves. And you'll see the content of the event as well. You can take a look what's happening here in the debug statement. Lots of things get printed out, which are helpful to understand what's going on. And then this is the key concept for keeping the event loop running for as long as you like it to run. Hopefully the agent will run only as much as it needs to. But as each event comes in, there's a flag that says, okay, this is the final event. I'm done processing this. The agent is done processing. If it has set this, then you can know that, okay, the processing is over. You can take a look at the content, decide what the final response might be. Here we're taking a look at the content if they have some multiple parts. So if it's a non-zero array of text that had come back, we're going to take the first message back from the parts zero here, grab the text from that, and here we're making some assumptions that the text response in the first part is actually the right one. Turn that into the final response text. Alternatively, the agent could have decided this is my final response, and by the way, I don't have actually a response, but I've asked the runtime system to do is actually to escalate. Now here escalation is meaning the agent is running, knows that it's a sub-agent and it can't handle or can't do any more work with what's currently available. So it's basically saying, hey, escalate this to somebody else, my parent or to some other agent that might be appropriate. In that case, the final response text that we're going to set is based on that, we're going to say that the agent has decided to escalate the current handling of the message and we don't have any specific message or there's an error message if there is one there and return that as the final message. All that is continued to run through one round through the event loop instigated by a single user message. And at the end, we'll go and print out the final agent response. It's kind of a lot to go through, but keep that in your head a little bit as you're going through the notebooks. Internally, this is really what's happening to actually make all the the machinery run. For all that work, there we go. Simple chat. Hello, my name is ADK. Hello to you ADK. Of course, because we're going to be doing a lot of calling to agents manually, we're going to set up a helper class that actually lets us do that a little bit more easily. And I'm going to call that helper class agent caller. This agent caller basically wraps up those functions that we did one at a time here, and also the event loop itself into one class. And if you look through the details, it is doing all the things that we've seen above. It's saving the single user ID, the single session ID, the current agent that we're going to be running, and also the runner that's been set up. It has access to getting the session that it uses internally for actually managing stuff. And this call function here is what actually instigates one round trip through the event loop based on a user message. So just as we saw above, it takes the user message, packages it up so that it's available as content to be processed by the agent, and then it runs through the event loop. Same kind of debug options are available as well. And this is a really stylistic choice. I'm not a native Python developer, so this is the way I like to do things. I prefer having factory methods for things that are really complex to construct. So rather than having a complex constructor inside of that class, I created a separate class that actually does the construction. And this here, this utility function, make agent caller, is what we'll create an instance of the agent caller. You will be passing into it all the necessary information it has, what's the agent that you want to be running, and also what is the initial state of that agent, where the state is the agent's internal memory. All of this should look just like what we tried to do before, but now packaged up into, here's the different elements, the major components that should be combined into the agent caller. What is the current app, the user, the session ID? Create a session service for both memory and also for managing the session itself. Then finally also create a runner. Then having the constructor of all those things, pass those into the agent caller which wraps all up into an execution loop. Okay, now that we have some utility functions for running an agent, we can run a conversation with the say hello agent that we said that we created earlier. So to create a new agent caller for our hello agent, we're going to actually call out to the helper function that make agent caller passing in the hello agent itself, and we'll just call that the hello agent caller. Then to run a conversation, we're going to have an asynchronous function that has multiple user messages that we're going to pass in to the agent caller using the call method. First we'll say hello, my name is ADK, and then I'll say I am excited. We'll go ahead and run that conversation and see what that looks like. Awesome. We have created a friendly chat agent. Good work. So now that you've gotten this far, you know the basics about using Google ADK, how to create an agent, how to give that agent tools that it can use, and also how to create an execution environment for that agent. Finally with all that in hand, you can script interactions with the agent here in the notebook, passing in user messages and then let you then test out how the agent is going to work. So this fantastic agent caller class that we created and then this instruction method, the factory method called make agent caller, we're going to use them throughout all the other notebooks to make it easy for once you've defined an agent, you can easily create a runtime environment using that function. So you'll get a lot of use out of that.