Getting started with Semantic Kernel

11/09/2024

In just a few steps, you can build your first AI agent with Semantic Kernel in either Python, .NET, or Java. This guide will show you how to...

- Install the necessary packages
- Create a back-and-forth conversation with an AI
- Give an AI agent the ability to run your code
- Watch the AI create plans on the fly

Installing the SDK

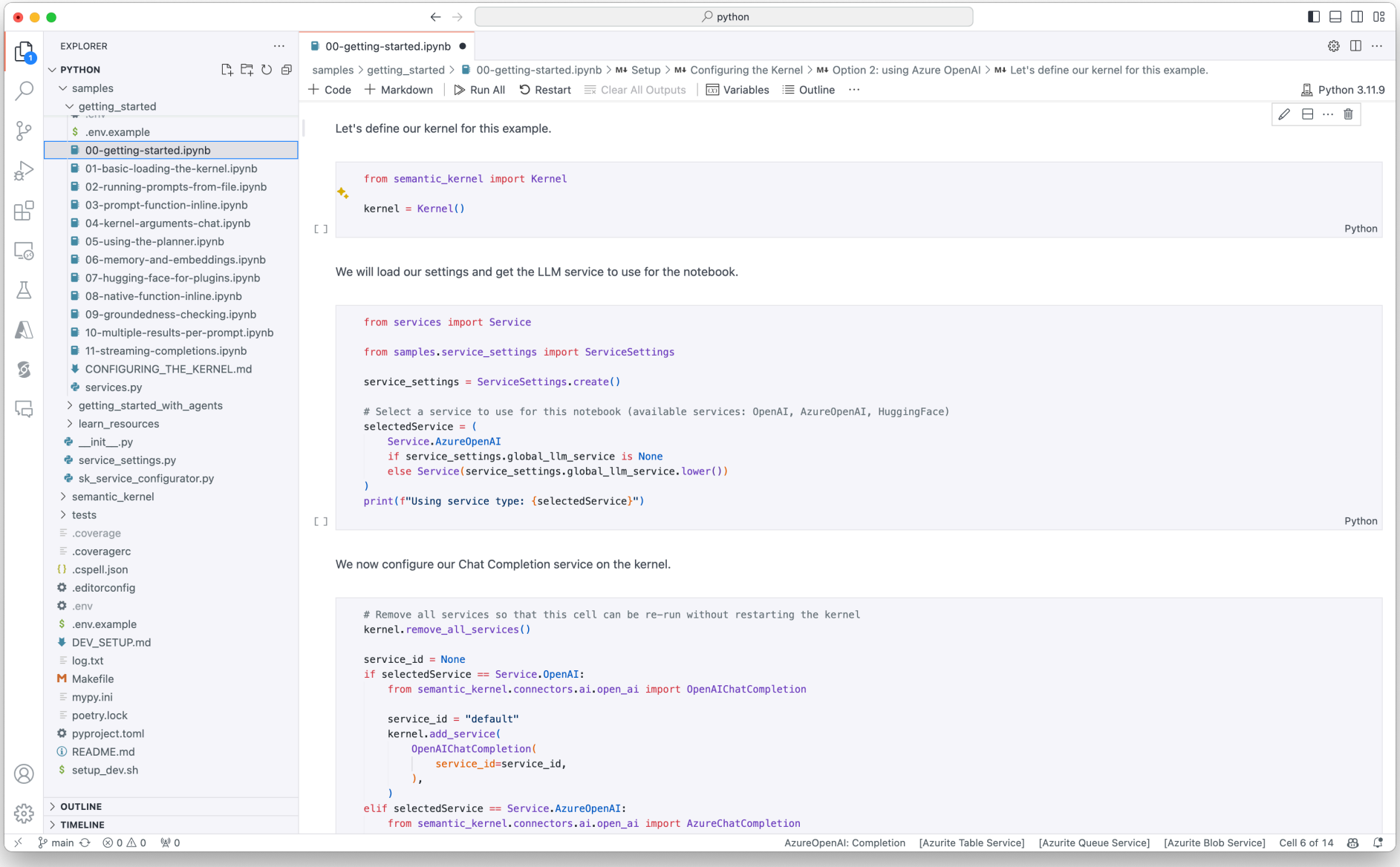
Instructions for accessing the `SemanticKernel` Python package is available here . It's as easy as:

PyPI

```
pip install semantic-kernel
```

Quickly get started with notebooks

If you're a Python or C# developer, you can quickly get started with our notebooks. These notebooks provide step-by-step guides on how to use Semantic Kernel to build AI agents.



To get started, follow these steps:

1. Clone the Semantic Kernel repo
2. Open the repo in Visual Studio Code
3. Navigate to _/python/samples/getting_started
4. Open *00-getting-started.ipynb* to get started setting your environment and creating your first AI agent!

Writing your first console app

Python

```python
import asyncio

from semantic_kernel import Kernel
from semantic_kernel.utils.logging import setup_logging
from semantic_kernel.functions import kernel_function
from semantic_kernel.connectors.ai.open_ai import AzureChatCompletion
from semantic_kernel.connectors.ai.function_choice_behavior import FunctionChoiceBehavior
from semantic_kernel.connectors.ai.chat_completion_client_base import ChatCompletionClientBase
from semantic_kernel.contents.chat_history import ChatHistory
from semantic_kernel.functions.kernel_arguments import KernelArguments

from semantic_kernel.connectors.ai.open_ai.prompt_execution_settings.azure_chat_prompt_execution_settings import (
    AzureChatPromptExecutionSettings,
)

async def main():
    # Initialize the kernel
    kernel = Kernel()

    # Add Azure OpenAI chat completion
    chat_completion = AzureChatCompletion(
        deployment_name="your_models_deployment_name",
        api_key="your_api_key",
        base_url="your_base_url",
    )
```

```
kernel.add_service(chat_completion)

# Set the logging level for  semantic_kernel.kernel to DEBUG.
setup_logging()
logging.getLogger("kernel").setLevel(logging.DEBUG)

# Add a plugin (the LightsPlugin class is defined below)
kernel.add_plugin(
    LightsPlugin(),
    plugin_name="Lights",
)

# Enable planning
execution_settings = AzureChatPromptExecutionSettings()
execution_settings.function_choice_behavior = FunctionChoiceBehavior.Auto()

# Create a history of the conversation
history = ChatHistory()

# Initiate a back-and-forth chat
userInput = None
while True:
    # Collect user input
    userInput = input("User > ")

    # Terminate the loop if the user says "exit"
    if userInput == "exit":
        break

    # Add user input to the history
    history.add_user_message(userInput)

    # Get the response from the AI
    result = await chat_completion.get_chat_message_content(
        chat_history=history,
        settings=execution_settings,
        kernel=kernel,
```

```
        )

        # Print the results
        print("Assistant > " + str(result))

        # Add the message from the agent to the chat history
        history.add_message(result)

# Run the main function
if __name__ == "__main__":
    asyncio.run(main())
```

The following back-and-forth chat should be similar to what you see in the console. The function calls have been added below to demonstrate how the AI leverages the plugin behind the scenes.

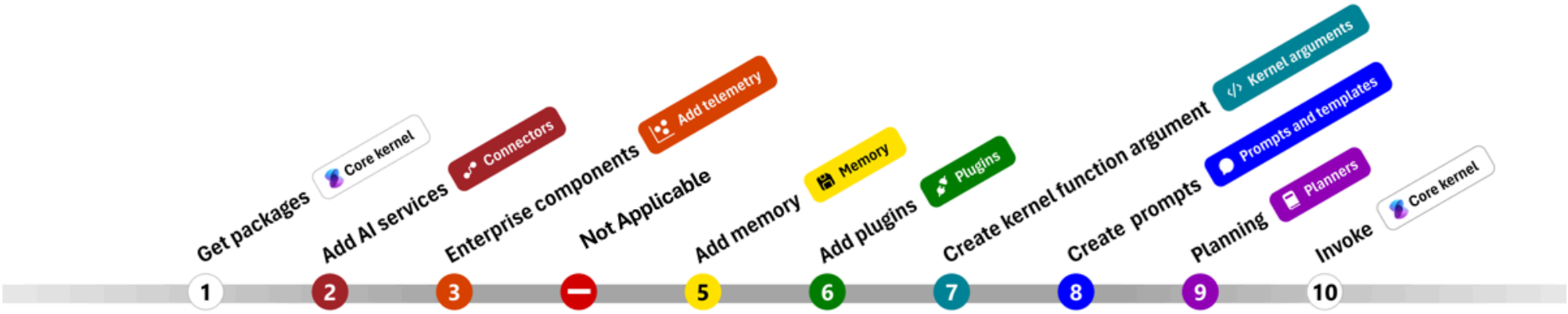
Expand table

Role	Message
User	Please toggle the light
Assistant (function call)	`LightsPlugin.GetState()`
Tool	`off`
Assistant (function call)	`LightsPlugin.ChangeState(true)`
Tool	`on`
Assistant	The light is now on

If you're interested in understanding more about the code above, we'll break it down in the next section.

Understanding the code

To make it easier to get started building enterprise apps with Semantic Kernel, we've created a step-by-step that guides you through the process of creating a kernel and using it to interact with AI services.



In the following sections, we'll unpack the above sample by walking through steps **1**, **2**, **3**, **4**, **6**, **9**, and **10**. Everything you need to build a simple agent that is powered by an AI service and can run your code.

- Import packages
- Add AI services
- Enterprise components
- Build the kernel
- Add memory (skipped)
- Add plugins
- Create kernel arguments (skipped)
- Create prompts (skipped)
- Planning

- Invoke

1) Import packages

For this sample, we first started by importing the following packages:

Python

```python
import asyncio

from semantic_kernel import Kernel
from semantic_kernel.connectors.ai.open_ai import AzureChatCompletion
from semantic_kernel.connectors.ai.function_choice_behavior import FunctionChoiceBehavior
from semantic_kernel.connectors.ai.chat_completion_client_base import ChatCompletionClientBase
from semantic_kernel.contents.chat_history import ChatHistory
from semantic_kernel.functions.kernel_arguments import KernelArguments

from semantic_kernel.connectors.ai.open_ai.prompt_execution_settings.azure_chat_prompt_execution_settings import (
    AzureChatPromptExecutionSettings,
)
```

2) Add AI services

Afterwards, we add the most important part of a kernel: the AI services that you want to use. In this example, we added an Azure OpenAI chat completion service to the kernel builder.

> **Note**
>
> In this example, we used Azure OpenAI, but you can use any other chat completion service. To see the full list of supported services, refer to the supported languages article. If you need help creating a different service, refer to the AI services article. There, you'll find

guidance on how to use OpenAI or Azure OpenAI models as services.

Python

```python
# Initialize the kernel
kernel = Kernel()

# Add Azure OpenAI chat completion
kernel.add_service(AzureChatCompletion(
    deployment_name="your_models_deployment_name",
    api_key="your_api_key",
    base_url="your_base_url",
))
```

3) Add enterprise services

One of the main benefits of using Semantic Kernel is that it supports enterprise-grade services. In this sample, we added the logging service to the kernel to help debug the AI agent.

Python

```python
import logging

# Set the logging level for  semantic_kernel.kernel to DEBUG.
logging.basicConfig(
    format="[%(asctime)s - %(name)s:%(lineno)d - %(levelname)s] %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S",
)
logging.getLogger("kernel").setLevel(logging.DEBUG)
```

4) Build the kernel and retrieve services

Once the kernel has been configured, we then retrieve the chat completion service for later use.

> **Note**
>
> In Python, you don't need to explicitly build the kernel. Instead, you can access the services directly from the kernel object.

Python

```
chat_completion : AzureChatCompletion = kernel.get_service(type=ChatCompletionClientBase)
```

6) Add plugins

With plugins, can give your AI agent the ability to run your code to retrieve information from external sources or to perform actions. In the above example, we added a plugin that allows the AI agent to interact with a light bulb. Below, we'll show you how to create this plugin.

Create a native plugin

Below, you can see that creating a native plugin is as simple as creating a new class.

In this example, we've created a plugin that can manipulate a light bulb. While this is a simple example, this plugin quickly demonstrates how you can support both...

1. Retrieval Augmented Generation (RAG) by providing the AI agent with the state of the light bulb
2. And task automation by allowing the AI agent to turn the light bulb on or off.

In your own code, you can create a plugin that interacts with any external service or API to achieve similar results.

Python

```python
from typing import Annotated
from semantic_kernel.functions import kernel_function

class LightsPlugin:
    lights = [
        {"id": 1, "name": "Table Lamp", "is_on": False},
        {"id": 2, "name": "Porch light", "is_on": False},
        {"id": 3, "name": "Chandelier", "is_on": True},
    ]

    @kernel_function(
        name="get_lights",
        description="Gets a list of lights and their current state",
    )
    def get_state(
        self,
    ) -> str:
        """Gets a list of lights and their current state."""
        return self.lights

    @kernel_function(
        name="change_state",
        description="Changes the state of the light",
    )
    def change_state(
        self,
        id: int,
        is_on: bool,
    ) -> str:
        """Changes the state of the light."""
        for light in self.lights:
            if light["id"] == id:
                light["is_on"] = is_on
```

```
                return light
        return None
```

Add the plugin to the kernel

Once you've created your plugin, you can add it to the kernel so the AI agent can access it. In the sample, we added the `LightsPlugin` class to the kernel.

Python

```
# Add the plugin to the kernel
kernel.add_plugin(
    LightsPlugin(),
    plugin_name="Lights",
)
```

9) Planning

Semantic Kernel leverages function calling–a native feature of most LLMs–to provide planning. With function calling, LLMs can request (or call) a particular function to satisfy a user's request. Semantic Kernel then marshals the request to the appropriate function in your codebase and returns the results back to the LLM so the AI agent can generate a final response.

To enable automatic function calling, we first need to create the appropriate execution settings so that Semantic Kernel knows to automatically invoke the functions in the kernel when the AI agent requests them.

Python

```
execution_settings = AzureChatPromptExecutionSettings()
execution_settings.function_choice_behavior = FunctionChoiceBehavior.Auto()
```

10) Invoke

Finally, we invoke the AI agent with the plugin. The sample code demonstrates how to generate a non-streaming response, but you can also generate a streaming response by using the `GetStreamingChatMessageContentAsync` method.

Python

```
# Create a history of the conversation
history = ChatHistory()

# Get the response from the AI
result = (await chat_completion.get_chat_message_contents(
    chat_history=history,
    settings=execution_settings,
    kernel=kernel,
    arguments=KernelArguments(),
))[0]
```

Next steps

In this guide, you learned how to quickly get started with Semantic Kernel by building a simple AI agent that can interact with an AI service and run your code. To see more examples and learn how to build more complex AI agents, check out our in-depth samples.