```
"""Build Agent using Microsoft Agent Framework in Python
# Run this python script
> pip install agent-framework
> python <this-script-path>.py
"""

import asyncio
import os

from agent_framework import ChatAgent, MCPStdioTool, MCPSseTools, ToolProtocol
from agent_framework.openai import OpenAIChatClient
from openai import AsyncOpenAI

# To authenticate with the model you will need to generate a personal access token (PAT) in your GitHub settings.
# Create your PAT token by following instructions here:
https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens
openaiClient = AsyncOpenAI(
    base_url = "https://models.github.ai/inference",
    api_key = os.environ["GITHUB_TOKEN"],
    default_query = {
        "api-version": "2024-08-01-preview",
    },
)

AGENT_NAME = "ai-agent"
AGENT_INSTRUCTIONS = "Xから話題の観光地を1つ選び、Google Mapsの情報に基づき、食べ歩きをテーマとした日帰り旅行プランを提案してください。¥n¥n以下のステップに従い、論理的な検討や調査結果（理由付け）を行ってから、最終的な旅行プラン（結論）を記載してください。必ず、観光地の選定理由・プラン作成の根拠から始め、結論（旅行プラン）を最後に記載してください。¥n¥n# Steps¥n¥n1. 観光地の候補を検討し、話題性やアクセス性、食べ歩きに適した理由などを調査・記載¥n2. Google Mapsで現地の飲食店やスポットをリサーチし、魅力的な食べ歩きルート案を検討・記載¥n3. 独自の工夫（時間配分、店の順番、立ち寄りスポット等）を考慮し、日帰りで回れるプランにまとめる¥n¥n# Output Format¥n¥n下記の構造で日本語で記載してください。¥n¥n{¥n　¥"理由・調査結果¥": [観光地選定の理由、Google Mapsでの調査内容や工夫した点などを具体的に記載],¥n　¥"旅行プラン¥": {¥n　　¥"観光地名¥": [選定した観光地名],¥n　　¥"日帰り食べ歩きプラン¥": [朝・昼・夕に分けた食べ歩きルート案（1日の流れ、店名や場所、各立ち寄りポイント、交通手段）を箇条書きまたは時系列で具体的に記載]¥n　}¥n}¥n¥n# Examples¥n¥n例1:¥n{¥n　¥"理由・調査結果¥": ¥"都内からアクセスが良く、SNSで話題の『鎌倉』を選定。Google Mapsで食べ歩きスポットを事前調査し、小町通り周辺は多様なグルメ店や観光名所が集まっていたため、食べ歩きに最適と判断。¥",¥n　¥"旅行プラン¥": {¥n　　¥"観光地名¥": ¥"鎌倉¥",¥n　　¥"日帰り食べ歩きプラン¥": [¥n　　¥"09:30 鎌倉駅到着。小町通りを散策¥",¥n　　¥"10:00 『アイスキャンディー本舗』で朝のスイーツ¥",¥n　　¥"10:30 『若宮大路』でたい焼き＆お団子を食べ歩き¥",¥n　　¥"12:00 『鶴岡八幡宮』参拝¥",¥n　　¥"13:00 『しらす丼の店』で昼食¥",¥n　　¥"14:30 『カフェ鎌倉』で休憩＆スイーツ¥",¥n　　¥"16:00 江ノ電で長谷エリアへ移動、『大仏』見学¥",¥n　　¥"17:00 鎌倉駅発、帰路へ¥"¥n　]¥n　}¥n}¥n¥n（実際の出力では、理由・プラン部分がさらに詳細になるよう、Google Mapsの情報や具体的な店名、工夫点なども明記してください）¥n¥n# Notes¥n¥n- 必ず観光地選定理由・Google Maps調査結果・プラン作成時の工夫などを「理由・調査結果」欄に記載してください¥n- 「旅行プラン」欄は朝から夕方までの時系列で、店名・スポット名を含めて具体的に記載してください¥n- 出力はJSON形式で、コードブロックは使用しないでください¥n- 結論（旅行プラン）は必ず論理的な検討・根拠（理由・調査結果）の後に記載してください"

# User inputs for the conversation
USER_INPUTS = [
    "Hello",
]

def create_mcp_tools() -> list[ToolProtocol]:
    return [
        MCPStdioTool(
            name="google-maps".replace("-", "_"),
            description="MCP server for google-maps",
            command="npx",
            args=[
                "-y",
                "@modelcontextprotocol/server-google-maps",
            ],
            env={
                "GOOGLE_MAPS_API_KEY": os.environ.get("GOOGLE_MAPS_API_KEY", ""),
            }
        ),
        MCPStdioTool(
            name="filesystem".replace("-", "_"),
            description="MCP server for filesystem",
            command="npx",
            args=[
                "-y",
                "@modelcontextprotocol/server-filesystem",
                "C:¥¥Users¥¥[ユーザ名]¥¥Desktop¥¥mcp-data",
            ]
        ),
        MCPStdioTool(
            name="twitter-mcp".replace("-", "_"),
            description="MCP server for twitter-mcp",
            command="npx",
            args=[
                "-y",
                "@enescinar/twitter-mcp",
            ],
            env={
                "API_KEY": os.environ.get("API_KEY", ""),
                "API_SECRET_KEY": os.environ.get("API_SECRET_KEY", ""),
                "ACCESS_TOKEN": os.environ.get("ACCESS_TOKEN", ""),
                "ACCESS_TOKEN_SECRET": os.environ.get("ACCESS_TOKEN_SECRET", ""),
            },
```

```
            env={
                "API_KEY": os.environ.get("API_KEY", ""),
                "API_SECRET_KEY": os.environ.get("API_SECRET_KEY", ""),
                "ACCESS_TOKEN": os.environ.get("ACCESS_TOKEN", ""),
                "ACCESS_TOKEN_SECRET": os.environ.get("ACCESS_TOKEN_SECRET", ""),
            },
            env={
                "API_KEY": os.environ.get("API_KEY", ""),
                "API_SECRET_KEY": os.environ.get("API_SECRET_KEY", ""),
                "ACCESS_TOKEN": os.environ.get("ACCESS_TOKEN", ""),
                "ACCESS_TOKEN_SECRET": os.environ.get("ACCESS_TOKEN_SECRET", ""),
            },
            env={
                "API_KEY": os.environ.get("API_KEY", ""),
                "API_SECRET_KEY": os.environ.get("API_SECRET_KEY", ""),
                "ACCESS_TOKEN": os.environ.get("ACCESS_TOKEN", ""),
                "ACCESS_TOKEN_SECRET": os.environ.get("ACCESS_TOKEN_SECRET", ""),
            }
        ),
    ]

async def main() -> None:
    async with (
        ChatAgent(
            chat_client=OpenAIChatClient(
                async_client=openaiClient,
                ai_model_id="openai/gpt-4.1"
            ),
            instructions=AGENT_INSTRUCTIONS,
            temperature=1,
            top_p=1,
            tools=create_mcp_tools(),
        ) as agent
    ):
        # Create a new thread that will be reused
        thread = agent.get_new_thread()

        # Process user messages
        for user_input in USER_INPUTS:
            print(f"¥n# User: '{user_input}'")
            async for chunk in agent.run_stream([user_input], thread=thread):
                if chunk.text:
                    print(chunk.text, end="")
                elif (
                    # log tool calls if any
                    chunk.raw_representation
                    and chunk.raw_representation.raw_representation
                    and hasattr(chunk.raw_representation.raw_representation, "choices")
                    and chunk.raw_representation.raw_representation.choices is not None
                    and len(chunk.raw_representation.raw_representation.choices) > 0
                    and hasattr(chunk.raw_representation.raw_representation.choices[0], "delta")
                    and hasattr(chunk.raw_representation.raw_representation.choices[0].delta, "tool_calls")
                    and chunk.raw_representation.raw_representation.choices[0].delta.tool_calls is not None
                    and len(chunk.raw_representation.raw_representation.choices[0].delta.tool_calls) > 0
                ):
                    toolCalls = list(filter(lambda call: call.function.name != None,
chunk.raw_representation.raw_representation.choices[0].delta.tool_calls))
                    if len(toolCalls) > 0:
                        print("")
                        print("Tool calls:", list(map(lambda call: call.function.name, toolCalls)))
            print("")

        print("¥n--- All tasks completed successfully ---")

        # Give additional time for all async cleanup to complete
        await asyncio.sleep(1.0)

if __name__ == "__main__":
    try:
        asyncio.run(main())
    except KeyboardInterrupt:
        print("¥nProgram interrupted by user")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
        import traceback
        traceback.print_exc()
    finally:
        print("Program finished.")
```