OpenAI Agents SDK

The OpenAI Agents SDK enables you to build agentic AI apps in a lightweight, easy-to-use package with very few abstractions.

In combination with Python, these primitives are powerful enough to express complex relationships between tools and agents, and allow you to build real-world applications without a steep learning curve. In addition, the SDK comes with built-in **tracing** that lets you visualize and debug your agentic flows, as well as evaluate them and even fine-tune models for your application.

Why use the Agents SDK

The SDK has two driving design principles:

1. Enough features to be worth using, but few enough primitives to make it quick to learn.
2. Works great out of the box, but you can customize exactly what happens.

Here are the main features of the SDK:

- **Agent loop:** Built-in agent loop that handles calling tools, sending results to the LLM, and looping until the LLM is done.
- **Python-first:** Use built-in language features to orchestrate and chain agents, rather than needing to learn new abstractions.
- **Handoffs:** A powerful feature to coordinate and delegate between multiple agents.
- **Guardrails:** Run input validations and checks in parallel to your agents, breaking early if the checks fail.
- **Sessions:** Automatic conversation history management across agent runs, eliminating manual state handling.
- **Function tools:** Turn any Python function into a tool, with automatic schema generation and Pydantic-powered validation.
- **Tracing:** Built-in tracing that lets you visualize, debug and monitor your workflows, as well as use the OpenAI suite of evaluation, fine-tuning and distillation tools.

- ## Installation

  pip install openai-agents

  Hello world example

  Code: from agents import Agent, Runner

  This line imports two main components from the agents library:

  - Agent: Represents the AI agent.
  - Runner: Helps execute (run) the agent with a specific input prompt.

  Code: agent = Agent(name="Assistant", instructions="You are a helpful assistant")

  - This creates an **AI agent** named "Assistant".
  - It is given some **instructions** (also called a system prompt):
    "You are a helpful assistant"
  - These instructions guide how the agent should behave (e.g., polite, helpful, etc.)

  result = Runner.run_sync(agent, "Write a haiku about recursion in programming.")

  - This line **runs the agent** synchronously (i.e., blocking call) using the Runner.
  - The agent receives this **user input**:
    "Write a haiku about recursion in programming."
  - It will respond with a short poem (haiku) based on the request.

  print(result.final_output)

  - This prints the **final answer** the AI agent generated.
  - result.final_output contains the agent's full response after it finishes the task.

(*If running this, ensure you set the OPENAI_API_KEY environment variable*)

Create a project and virtual environment

```
mkdir my_project
cd my_project
python -m venv .venv
```

Activate the virtual environment

```
.venv/bin/activate
```

Install the Agents SDK

```
pip install openai-agents
```

Set an OpenAI API key

Set openai api key on .env file in vscode.

Create your first agent

Agents are defined with instructions, a name, and optional config (such as model_config)

```
from agents import Agent
agent = Agent(
  name="Math Tutor",
  instructions="You provide help with math problems. Explain your reasoning at each step
and include examples",)
```

Add a few more agents

Additional agents can be defined in the same way. handoff_descriptions provide additional context for determining handoff routing

```
from agents import Agent
history_tutor_agent = Agent(
  name="History Tutor",
  handoff_description="Specialist agent for historical questions",
  instructions="You provide assistance with historical queries. Explain important events
and context clearly.",)

math_tutor_agent = Agent(
  name="Math Tutor",
  handoff_description="Specialist agent for math questions",
  instructions="You provide help with math problems. Explain your reasoning at each step
and include examples",)
```

Define your handoffs

On each agent, you can define an inventory of outgoing handoff options that the agent can choose from to decide how to make progress on their task.

```
triage_agent = Agent(
  name="Triage Agent",
  instructions="You determine which agent to use based on the user's homework
question",
  handoffs=[history_tutor_agent, math_tutor_agent])
```

Run the agent orchestration

Let's check that the workflow runs and the triage agent correctly routes between the two specialist agents.

```
from agents import Runner
async def main():
  result = await Runner.run(triage_agent, "What is the capital of France?")
  print(result.final_output)
```

Add a guardrail

You can define custom guardrails to run on the input or output.

```
from agents import GuardrailFunctionOutput, Agent, Runner
from pydantic import BaseModel
class HomeworkOutput(BaseModel):
  is_homework: bool
  reasoning: str
guardrail_agent = Agent(
  name="Guardrail check",
  instructions="Check if the user is asking about homework.",
  output_type=HomeworkOutput,)
async def homework_guardrail(ctx, agent, input_data):
  result = await Runner.run(guardrail_agent, input_data, context=ctx.context)
  final_output = result.final_output_as(HomeworkOutput)
  return GuardrailFunctionOutput(
    output_info=final_output,
    tripwire_triggered=not final_output.is_homework,   )
```

Put it all together

Let's put it all together and run the entire workflow, using handoffs and the input guardrail.
Link code
https://openai.github.io/openai-agents-python/quickstart/

Imports

```
from agents import Agent, InputGuardrail, GuardrailFunctionOutput, Runner
from agents.exceptions import InputGuardrailTripwireTriggered
from pydantic import BaseModel
import asyncio
```

These import necessary classes for agents, error handling, data validation (pydantic.BaseModel), and async execution.

Model to validate guardrail output

```
class HomeworkOutput(BaseModel):
  is_homework: bool
  reasoning: str
```

- Defines the structure of the output from the guardrail check.
- is_homework: Tells if the question is actually about homework.
- reasoning: Explanation for the decision.

Guardrail Agent

```
guardrail_agent = Agent(
name="Guardrail check",
instructions="Check if the user is asking about homework.",
output_type=HomeworkOutput,)
```

- This agent checks if the user input is a *homework question* or not.
- Returns structured output (HomeworkOutput).

Specialist Agents

```
math_tutor_agent = Agent(...)
history_tutor_agent = Agent(...)
```

math_tutor_agent: Helps with math questions.
history_tutor_agent: Helps with history questions.
Both explain their answers clearly.

Guardrail Function

```
async def homework_guardrail(ctx, agent, input_data):
  result = await Runner.run(guardrail_agent, input_data, context=ctx.context)
  final_output = result.final_output_as(HomeworkOutput)
  return GuardrailFunctionOutput(
    output_info=final_output,
    tripwire_triggered=not final_output.is_homework,   )
```

This is the actual **guardrail logic**.
If is_homework is False, the guardrail **blocks** the input using a "tripwire."

🔍 What is a "Tripwire"?

A **tripwire** in programming (especially in AI guardrails) is like a **trap** or **filter** that says:

❗ *"If this condition is met, stop everything and don't let the user's input proceed."*

Triage Agent with Guardrail

```
triage_agent = Agent(
  name="Triage Agent",
  instructions="You determine which agent to use...",
  handoffs=[history_tutor_agent, math_tutor_agent],
  input_guardrails=[InputGuardrail(guardrail_function=homework_guardrail)],)
```

- This agent routes the question to the appropriate tutor (math/history).
- But it first checks input using the guardrail.
- If the input is *not* a homework question, it raises an exception and blocks the request.

Main Function

```
async def main():
  # Example 1: History question
  try:
    result = await Runner.run(triage_agent, "who was the first president of the united states?")
    print(result.final_output)
  except InputGuardrailTripwireTriggered as e:
    print("Guardrail blocked this input:", e)
```

This **is** a homework-style question → **allowed** and handled by the history tutor.

```
  # Example 2: General/philosophical question
  try:
    result = await Runner.run(triage_agent, "What is the meaning of life?")
    print(result.final_output)
  except InputGuardrailTripwireTriggered as e:
    print("Guardrail blocked this input:", e)
if __name__ == "__main__":
  asyncio.run(main())
```

This is **not a homework question** → the guardrail **blocks** it and raises InputGuardrailTripwireTriggered

Agents

Agents are the core building block in your apps. An agent is a large language model (LLM), configured with instructions and tools.

Basic configuration
The most common properties of an agent you'll configure are:

- name: A required string that identifies your agent.
- instructions: also known as a developer message or system prompt.
- model: which LLM to use, and optional model_settings to configure model tuning parameters like temperature, top_p, etc.
- tools: Tools that the agent can use to achieve its tasks.

```
from agents import Agent, ModelSettings, function_tool

@function_tool
def get_weather(city: str) -> str:
     """returns weather info for the specified city."""
    return f"The weather in {city} is sunny"

agent = Agent(
    name="Haiku agent",
    instructions="Always respond in haiku form",
    model="o3-mini",
    tools=[get_weather],)
```

✅ **1. Imports**
from agents import Agent, ModelSettings, function_tool

Agent: the core object that represents an AI agent.
ModelSettings: likely used to configure the model (though **not used** in your current code).
function_tool: a decorator used to turn Python functions into **tools** that an agent can call.

@function_tool: turns the function into a callable tool that the agent can use when needed.
get_weather(city: str) -> str: a simple function that takes a city name and returns a string describing the weather. In this mockup, it always returns "The weather in {city} is sunny" regardless of real data.

name: "Haiku agent" – a human-friendly name for the agent.
instructions: "Always respond in haiku form" – tells the agent to answer using **haikus** (a 3-line poem: 5-7-5 syllables).
model: "o3-mini" – specifies which model to use.
tools: [get_weather] – allows the agent to **call this function** when it needs weather info.

Why Use **Haiku** for an AI?

- It's poetic and creative.
- It forces the AI to think within a structure.
- It's fun and surprising for users!

Context

Agents are generic on their context type. Context is a dependency-injection tool: it's an object you create and pass to Runner.run(), that is passed to every agent, tool, handoff etc, and it serves as a grab bag of dependencies and state for the agent run. You can provide any Python object as the context.

In the OpenAI Agents SDK, context refers to the data or memory that is passed along the agent's loop as it interacts with users and tools. This helps the agent remember important facts, share state, and customize behavior during the workflow.

Why is context important?
Context allows you to:

- Keep track of what's already been done.
- Pass information between tools and the agent.
- Maintain continuity in multi-step tasks.

Let's look at a basic example where an agent performs two tasks:

1. **Gets** the user's name.
2. **Greets** the user by name

```
from agents import Agent, function_tool, RunConfig, Runner

# Tool that saves user name in context
@function_tool
def save_name(name: str, context) -> str:
   context["user_name"] = name
   return f"Nice to meet you, {name}!"

# Tool that greets using the name from context
@function_tool
def greet_user(context) -> str:
   name = context.get("user_name", "friend")
   return f"Hello again, {name}!"

# Define the agent
agent = Agent(
   name="GreetingAgent",
  instructions="You're a friendly assistant who remembers the user's name.",
  tools=[save_name, greet_user],)

# Run the agent in a conversation
result = Runner.run_sync(agent, "My name is Ali. Say hi to me later.")
print(result.final_output)
```

Context is a dictionary (context: dict) that persists during the agent's reasoning cycle.

The tool save_name extracts the name and stores it in context.

Later, greet_user accesses that same context to give a personalized greeting.

Output types

By default, agents produce plain text (i.e. str) outputs. If you want the agent to produce a particular type of output, you can use the output_type parameter.

In the **OpenAI Agents SDK**, **Output Types** define how the agent responds at the end of its run — whether it's returning a **text string**, **JSON object**, **tool call result**, or other structured output.

Why Output Types Matter
Output types allow your agent to:

- Return human-readable text (like a chatbot).
- Return structured data (like a JSON object for API use).
- Return intermediate tool outputs.

Output Type	**Description**	**Example Use Case**
str (Text)	Default. Returns a final string response	Chatbot greeting, summary, etc.
dict / JSON	Structured response using dictionaries	Return data like weather info
ToolResult	Tool return values	When you want the raw tool output
Custom	Custom object types you define	Advanced workflows with structure

```
from agents import Agent, function_tool, Runner

@function_tool
def get_fact() -> str:
   return "The Earth revolves around the Sun."

agent = Agent(
   name="SimpleAgent",
   instructions="Provide scientific facts.",
   tools=[get_fact],)

result = Runner.run_sync(agent, "Tell me a fact")
print(result.final_output)  # Output: The Earth revolves around the Sun.
```

Example 2: JSON/Dict Output

```
from agents import Agent, function_tool, Runner

@function_tool
def get_weather(city: str) -> dict:
   return {"city": city, "temperature": "35°C", "condition": "Sunny"}

agent = Agent(
   name="WeatherAgent",
   instructions="Give structured weather data.",
   tools=[get_weather],
)

result = Runner.run_sync(agent, "What's the weather in Karachi?")
print(result.final_output)
# Output: {'city': 'Karachi', 'temperature': '35°C', 'condition': 'Sunny'}
```