**Purpose Learning OpenAI Agents SDK(Agents)**
**Created by Samad https://github.com/samade747**
**https://github.com/samade747/openai-sdk-decoded/tree/main/Basic%20Learning**

# Learning the OpenAI Agents SDK: A Beginner's Guide

The OpenAI Agents SDK is a tool that lets you create smart AI assistants, called agents, which can perform tasks, answer questions, and even work together. These agents use powerful language models, like those from OpenAI, to understand and respond to user inputs. If you're starting with no prior knowledge, this guide will walk you through setting up, building, and running agents step by step, with simple explanations and code examples. While the SDK is designed to be user-friendly, some concepts like asynchronous programming might feel tricky at first, but we'll break them down clearly.

- **Key Points**:
  - The OpenAI Agents SDK helps build AI agents that can use tools and delegate tasks.
  - It's beginner-friendly but requires basic Python setup and an API key.
  - Agents are configured with instructions, models, and tools to handle tasks.
  - You can run agents to process inputs and get outputs, like answers or actions.
  - Advanced features like handoffs and guardrails add flexibility but may need practice.

### What You'll Need

You'll need a computer with internet access, and we'll guide you through installing Python and the SDK. No prior coding experience is assumed, but following along with the examples will help you learn.

### How to Get Started

First, you'll set up your environment, then create a simple agent, and finally run it to see how it works. We'll also cover advanced features like tools and handoffs to give you a full picture.

### What to Expect

By the end, you'll have a working agent that can answer questions or perform tasks, and you'll understand how to expand it with more features. The process is straightforward, but setting up the environment correctly is crucial.

# Comprehensive Guide to Learning the OpenAI Agents SDK

This guide is designed for someone with no prior knowledge, providing a complete, step-by-step path to understanding and using the OpenAI Agents SDK. It includes detailed explanations, commented code examples, and covers everything from setup to advanced features, based on the official documentation and additional research.

## 1. Introduction to the OpenAI Agents SDK

The OpenAI Agents SDK is a Python library that enables developers to create intelligent AI agents powered by large language models (LLMs) like GPT-4o. These agents can process user inputs, use tools (like functions or APIs), delegate tasks to other agents, and validate inputs/outputs with guardrails. The SDK is lightweight, making it accessible for beginners, yet powerful enough for complex applications.

### Why Use the SDK?

- **Agents**: Act as smart assistants that can follow instructions and use tools.
- **Handoffs**: Allow agents to pass tasks to specialized agents for efficiency.
- **Guardrails**: Ensure inputs and outputs meet specific criteria, like safety rules.
- **Tracing**: Helps debug and visualize how agents process tasks.

**2. Setting Up Your Environment**

Before coding, you need to set up your computer with Python, the SDK, and an OpenAI API key.

**Step 2.1: Install Python**

- Download Python 3.10 or later from python.org.
- Follow the installation instructions for your operating system (Windows, macOS, or Linux).
- Verify installation by opening a terminal (Command Prompt on Windows, Terminal on macOS/Linux) and typing:
  bash
  CollapseWrapRun
  Copy
  ```
  python --version
  ```
  You should see the Python version (e.g., Python 3.11.0).

**Step 2.2: Create a Project and Virtual Environment**

A virtual environment keeps your project's dependencies separate, avoiding conflicts.

Open a terminal and create a project directory:
bash
CollapseWrapRun
Copy
```
mkdir my_project
```

1. `cd my_project`

2. Create a virtual environment:
   bash
   CollapseWrapRun
   Copy
   ```
   python -m venv .venv
   ```

3. Activate the virtual environment:
   - On Windows:
     bash
     CollapseWrapRun
     Copy
     ```
     .venv\Scripts\activate
     ```

- On macOS/Linux:
  bash
  CollapseWrapRun
  Copy
  ```
  source .venv/bin/activate
  ```

4. Your terminal prompt should change, indicating the virtual environment is active.

**Step 2.3: Install the OpenAI Agents SDK**

Install the SDK using pip:

bash
CollapseWrapRun
Copy
```
pip install openai-agents
```

This command downloads and installs the SDK and its dependencies.

**Step 2.4: Set Up Your OpenAI API Key**

The SDK requires an API key to access OpenAI's models.

1. Create an API key at [OpenAI's platform](#).
2. Set the API key as an environment variable:
   bash
   CollapseWrapRun
   Copy
   ```
   export OPENAI_API_KEY=sk-...
   ```
   Replace sk-... with your actual API key. On Windows, use:
   bash
   CollapseWrapRun
   Copy
   ```
   set OPENAI_API_KEY=sk-...
   ```

## 3. Understanding Agents

Agents are the core of the SDK. They're like virtual assistants powered by an LLM, configured to perform specific tasks.

### What Are Agents?

- An **agent** is an LLM instance with:
  - **Instructions**: A prompt defining its purpose (e.g., "Answer math questions").
  - **Model**: The LLM used (e.g., "gpt-4o" or "o3-mini").
  - **Tools**: Functions or services the agent can call (e.g., a weather API).
- Agents process user inputs, decide actions (like calling tools), and produce outputs.

### How Agents Work

1. You define an agent with instructions, a model, and optional tools.
2. You provide an input (e.g., "What's the weather in London?").
3. The agent uses the LLM to interpret the input, possibly calling tools or delegating tasks.
4. It produces a final output or loops until the task is complete.

### Key Components

| Component | Description |
|---|---|
| Instructions | Defines the agent's behavior (e.g., "Respond in haiku form"). |
| Model | The LLM powering the agent (e.g., "gpt-4o"). |
| Tools | Functions the agent can call (e.g., a calculator or API). |
| Context | Extra data passed to the agent (e.g., user info). |
| Output Types | Format of the agent's output (e.g., string, list, or custom structure). |
| Handoffs | Allows delegation to other agents. |
| Guardrails | Rules to validate inputs/outputs (e.g., block offensive content). |

## 4. Building Your First Agent

Let’s create a simple agent that responds in haiku form and uses a tool to get weather information.

Simple_agent.py

```
from agents import Agent, function_tool

# Define a tool to get weather information
@function_tool
def get_weather(city: str) -> str:
    # Simulate an API call; in real scenarios, this would fetch real
data
    return f"The weather in {city} is sunny"

# Create an agent that responds in haiku form
agent = Agent(
    name="Haiku Agent",
    instructions="Always respond in haiku form",
    model="o3-mini",  # Use "gpt-4o" if available
    tools=[get_weather]
)
```

**Code Explanation**:

- function_tool: A decorator that marks a function as a tool the agent can use.
- get_weather: A simple function that takes a city name and returns a weather description.
- Agent: Creates an agent with a name, instructions, model, and tools.

**5. Running Agents**

To run an agent, you use the Runner class, which handles the agent's execution.

**Running a Single Agent**

Here's how to run the agent from the previous step:

Run_agent.py

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

# Define the weather tool
@function_tool
def get_weather(city: str) -> str:
    return f"The weather in {city} is sunny"

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

# Define the main function to run the agent
async def main():
    result = await Runner.run(agent, "What's the weather in New York?")
    print(result.final_output)  # Prints the agent's response

# Run the async function
```

```
asyncio.run(main())
```

**Code Explanation**:

- Runner.run: An asynchronous method that runs the agent with a given input.
- asyncio.run: Executes the asynchronous main function.
- The agent processes the input, calls the get_weather tool, and responds in haiku form.

**Handling Conversations**

For multi-turn conversations, you can maintain a conversation history:

conversation.py

```
from agents import Agent, Runner
import asyncio

# Create a concise agent
agent = Agent(name="Assistant", instructions="Reply very concisely.")

# Define the main function for a conversation
async def main():
    # First question
    result = await Runner.run(agent, "What city is the Golden Gate
Bridge in?")
    print(result.final_output)  # Expected: San Francisco

    # Continue the conversation
    new_input = result.to_input_list() + [{"role": "user", "content":
"What state is it in?"}]
    result = await Runner.run(agent, new_input)
    print(result.final_output)  # Expected: California

# Run the async function
asyncio.run(main())
```

**Code Explanation**:

- result.to_input_list(): Converts the previous run’s result into a list for conversation history.
- The agent maintains context by processing the updated input list.

## 6. Advanced Features

Once you're comfortable with basic agents, you can explore advanced features.

### Handoffs

Handoffs allow an agent to delegate tasks to other agents. For example, a triage agent can pass tasks to specialized agents:

handoffs.py

```python
from agents import Agent

# Define specialized agents
booking_agent = Agent(name="Booking Agent", instructions="Help users book tickets")
refund_agent = Agent(name="Refund Agent", instructions="Help users with refunds")

# Create a triage agent that delegates tasks
triage_agent = Agent(
    name="Triage Agent",
    instructions="Determine if the user wants to book or refund a ticket and hand off accordingly",
    handoffs=[booking_agent, refund_agent]
)
```

**Code Explanation**:

- handoffs: A list of sub-agents the triage agent can delegate to.
- The triage agent decides which sub-agent to use based on the input.

**Guardrails**

Guardrails validate inputs and outputs to ensure they meet specific criteria:

guardrails.py

```
from agents import Agent, Guardrail

# Define a guardrail to check for offensive language
def no_offensive_language(input: str) -> bool:
    # In a real scenario, implement logic to detect offensive content
    return True  # Placeholder: assume input is safe

# Create a guardrail
guardrail = Guardrail(input_checks=[no_offensive_language])

# Create an agent with guardrails
agent = Agent(
    name="Safe Agent",
    instructions="Answer questions safely",
    guardrails=guardrail
)
```

```
from agents import Agent, Guardrail

# Define a guardrail to check for offensive language
def no_offensive_language(input: str) -> bool:
    # In a real scenario, implement logic to detect offensive content
    return True  # Placeholder: assume input is safe

# Create a guardrail
guardrail = Guardrail(input_checks=[no_offensive_language])

# Create an agent with guardrails
agent = Agent(
    name="Safe Agent",
    instructions="Answer questions safely",
    guardrails=guardrail
)
```

**Code Explanation**:

- Guardrail: Defines rules to validate inputs.
- no_offensive_language: A placeholder function to check input safety.

## Dynamic Instructions

You can provide instructions that change based on context:

Dynamic_instructions.py

```
from agents import Agent, RunContextWrapper
from dataclasses import dataclass

# Define a context class
@dataclass
class UserContext:
    name: str

# Define dynamic instructions
def dynamic_instructions(context: RunContextWrapper[UserContext],
agent: Agent[UserContext]) -> str:
    return f"The user's name is {context.context.name}. Help them..."

# Create an agent with dynamic instructions
agent = Agent(
    name="Triage Agent",
    instructions=dynamic_instructions
)
```

**Code Explanation**:

- dynamic_instructions: A function that generates instructions based on context.
- UserContext: A dataclass to hold user-specific data.

**Cloning Agents**

You can clone an agent to create a new one with modified properties:

clone_agent.py

```
from agents import Agent

# Create a base agent
pirate_agent = Agent(name="Pirate", instructions="Write like a pirate",
model="o3-mini")

# Clone the agent with new properties
robot_agent = pirate_agent.clone(name="Robot", instructions="Write like
a robot")
```

**Code Explanation**:

- clone: Creates a new agent based on an existing one, allowing you to modify specific properties.

## 7. Debugging and Tracing

The SDK includes built-in tracing to help you debug and visualize agent workflows.

tracing.py

```
from agents import Agent, Runner, trace
import asyncio

# Create an agent
agent = Agent(name="Assistant", instructions="Reply very concisely.")

# Run the agent with tracing
async def main():
    with trace(workflow_name="Conversation", group_id="thread_123"):
        result = await Runner.run(agent, "What city is the Golden Gate
Bridge in?")
        print(result.final_output)  # Expected: San Francisco

# Run the async function
asyncio.run(main())
```

**Code Explanation**:

- trace: A context manager that logs the agent's workflow.
- View traces in the OpenAI Dashboard.

## 8. Complete Example: Customer Support System

Let’s build a customer support system with a triage agent that delegates to booking or refund agents.

customer_support.py
```python
from agents import Agent, function_tool, Runner
from dataclasses import dataclass
import asyncio

# Define a context class
@dataclass
class UserContext:
    uid: str
    is_pro_user: bool

# Define tools
@function_tool
def book_ticket(user_id: str) -> str:
    return f"Ticket booked for user {user_id}"

@function_tool
def refund_ticket(user_id: str) -> str:
    return f"Refund processed for user {user_id}"

# Create specialized agents
booking_agent = Agent(
    name="Booking Agent",
    instructions="Help users book tickets",
    tools=[book_ticket]
)
refund_agent = Agent(
    name="Refund Agent",
    instructions="Help users with refunds",
    tools=[refund_ticket]
)

# Create a triage agent
triage_agent = Agent(
    name="Triage Agent",
    instructions="Determine if the user wants to book or refund a
ticket and hand off accordingly",
    handoffs=[booking_agent, refund_agent]
)

# Run the system
async def main():
    user_input = "I want to book a ticket."
    context = UserContext(uid="123", is_pro_user=True)
    result = await Runner.run(triage_agent, user_input,
context=context)
    print(result.final_output)

# Execute the async function
asyncio.run(main())
```

**Code Explanation**:

- **Context**: UserContext holds user data like ID and status.
- **Tools**: book_ticket and refund_ticket simulate booking and refund actions.
- **Agents**: booking_agent and refund_agent handle specific tasks, while triage_agent delegates based on input.
- **Runner**: Executes the triage agent, which may hand off to a sub-agent.

## 9. Best Practices and Tips

- **Start Simple**: Begin with a single agent and one tool to understand the basics.
- **Test Incrementally**: Add features like handoffs or guardrails one at a time.
- **Use Tracing**: Always enable tracing for debugging complex workflows.
- **Experiment**: Try different instructions and tools to see how agents behave.
- **Check Documentation**: Refer to the OpenAI Agents SDK Documentation for detailed guidance.

## 10. Conclusion

This guide has provided a complete path to learning the OpenAI Agents SDK, from setting up your environment to building and running complex agent systems. By following the steps and experimenting with the code examples, you can create your own AI agents for various applications. The SDK's flexibility allows you to start simple and gradually explore advanced features like handoffs, guardrails, and dynamic instructions.

**Key Citations**:

- OpenAI Agents SDK Official Documentation
- OpenAI Agents SDK Quickstart Guide
- OpenAI Platform Developer Resources
- GitHub Repository for OpenAI Agents SDK
- Building AI Agents with OpenAI Agents SDK
- OpenAI Agents SDK Tutorial by DataCamp
- How to Use the OpenAI Agents SDK
- Running OpenAI Agents SDK Locally
- OpenAI Agents SDK with Local LLM