Questions to Start Thinking….

1. Why is the `Agent` class defined as a `dataclass` in the OpenAI Agents SDK?

When you see this:

@dataclass

class Agent:

    tools: List[Tool]

    name: Optional[str] = None

    system_message: Optional[str] = None

    …

Think of it like this:

The `Agent` class is mainly holding data — like:

- What tools the agent uses
- What name it has
- What system message to follow

It doesn't do much by itself — it just stores information.

What `@dataclass` does:

It helps you create such data-holding classes quickly and cleanly, without writing extra code.

If they didn't use `@dataclass`, they'd need to write this:

```
class Agent:
    def __init__(self, tools, name=None, system_message=None):
        self.tools = tools
        self.name = name
        self.system_message = system_message
```

That's more code for the same thing.
With `@dataclass`, Python automatically creates this constructor (`__init__`) and other useful methods for you.

Benefits:

What you get	Why it's useful
`__init__()` method	Automatically created to store data
`__repr__()` method	So when you print the object, it looks nice
Less boilerplate	You write less code
Cleaner structure	You see all the fields at the top, clearly listed

Real-life Example:

Imagine you want to build a chatbot agent with tools like "search", "math", or "code".

You can just write:

```
agent = Agent(
    tools=["search", "code"],
    name="HelperBot",
    system_message="You're a helpful assistant"
)
```

This is simple and clean — and it works because of `@dataclass`.

2a. The system prompt is contained in the Agent class as instructions? Why you can also set it as callable?

The system prompt is like a set of instructions or rules given to the agent.
It helps shape how the agent behaves.

Two ways to give this instruction:

1. As plain text:

```
agent = Agent(
    system_message="You are a helpful assistant"
)
```

Simple, fixed instruction.

2. As a callable (function):

```
def dynamic_prompt(tools, memory):
    return "You are a helpful assistant with access to special tools."
```

agent = Agent(

    system_message=dynamic_prompt

)

This means:
 You can create the system message using logic — like based on the tools the agent has or what it remembers.

Why allow it as a function?

Because sometimes you want the system message to change depending on:

- What tools the agent has
- The current memory
- The context

So, by making it callable, you can generate smarter instructions at runtime.

2b. But the user prompt is passed as parameter in the run method of Runner and the method is a classmethod

- The user prompt is the actual question or command you give to the agent.
- Example: `"What's the weather today?"`

This changes every time the agent runs. So it makes sense to pass it as a parameter:

Runner.run("What's the weather today?")

Why is `run()` a `classmethod`?

Because you don't need to make an object of `Runner` to use it.
You can just call it like:

Runner.run(prompt="Translate this to French")

This design makes it:

- Simple to use
- Stateless (you don't store info in the Runner, it just runs with what you give)

Summary:

Concept	What it is	Why it works like that
`system_message`	Agent's permanent instructions	Can be fixed text or a smart function (callable)
`user_prompt`	The question you ask	Changes every time — passed to `.run()`
`Runner.run()`	A class method to activate the agent	Easy to call without creating a Runner object

3.) What is the purpose of the `Runner` class?

The `Runner` class is like the engine that runs the agent.

- You give it an `Agent`
- You give it the user's message
- Then it does all the work to process the tools, messages, memory, and return the response.

Let's say you have an agent like this:

agent = Agent(tools=[...], system_message="You are a helpful assistant")

Then you want to run this agent on a user prompt like:

Runner.run(agent, "What's 5 + 5?")

This will:

- Use the agent's system message
- Use the tools
- Think
- And return an answer like "5 + 5 is 10."

So in short:

`Runner` is the class that makes the agent actually work when you give it a user message.

4.) What are generics in Python? Why do we use it for `TContext`?

Generics are like placeholders for data types.

You use them when you want to say:

> “This function or class can work with any type of data, and I’ll decide what type later.”

What is `TContext` then?

In the SDK, `TContext` is a generic type that represents a "context object".

We don’t know what kind of data this context holds — it could be:

- User profile
- App settings
- History
- Anything!

So we write:

from typing import TypeVar

TContext = TypeVar("TContext")

This means:

> “TContext can be any type, and we’ll decide that when using the class or function.”

Why use generics?

1. Flexibility: You can reuse the same code with different types.
2. Type safety: Python can help you catch errors if you use the wrong type.
3. Clarity: It makes your code cleaner and easier to understand.

example with generics:

```
from typing import TypeVar, Generic

T = TypeVar("T")

class Box(Generic[T]):

    def __init__(self, item: T):

        self.item = item
```

Now this works:

```
b1 = Box         # A box that holds an int

b2 = Box[str]("Hello")    # A box that holds a string
```

Summary:

Concept	Meaning
`Runner`	The engine that runs the agent on a prompt
`Generics`	Flexible types you define later (like a blank box)
`TContext`	A placeholder for “context type” that can change depending on what you're building