Context management

Context is an overloaded term. There are two main classes of context you might care about:

1. Context available locally to your code: this is data and dependencies you might need when tool functions run, during callbacks like `on_handoff`, in lifecycle hooks, etc.
2. Context available to LLMs: this is data the LLM sees when generating a response.

Local context

This is represented via the `RunContextWrapper` class and the `context` property within it. The way this works is:

1. You create any Python object you want. A common pattern is to use a dataclass or a Pydantic object.
2. You pass that object to the various run methods (e.g. `Runner.run(..., **context=whatever**))`.
3. All your tool calls, lifecycle hooks etc will be passed a wrapper object, `RunContextWrapper[T]`, where `T` represents your context object type which you can access via `wrapper.context`.

The **most important** thing to be aware of: every agent, tool function, lifecycle etc for a given agent run must use the same *type* of context.

You can use the context for things like:

- Contextual data for your run (e.g. things like a username/uid or other information about the user)
- Dependencies (e.g. logger objects, data fetchers, etc)
- Helper functions

Note

The context object is **not** sent to the LLM. It is purely a local object that you can read from, write to and call methods on it.

```
import asyncio
from dataclasses import dataclass
```

```python
from agents import Agent, RunContextWrapper, Runner, function_tool

@dataclass
class UserInfo:  ①
    name: str
    uid: int

@function_tool
async def fetch_user_age(wrapper: RunContextWrapper[UserInfo]) -> str:  ②
    return f"User {wrapper.context.name} is 47 years old"

async def main():
    user_info = UserInfo(name="John", uid=123)

    agent = Agent[UserInfo](  ③
        name="Assistant",
        tools=[fetch_user_age],
    )

    result = await Runner.run(  ④
        starting_agent=agent,
        input="What is the age of the user?",
        context=user_info,
    )

    print(result.final_output)  ⑤
    # The user John is 47 years old.

if __name__ == "__main__":
    asyncio.run(main())
```

1. This is the context object. We've used a dataclass here, but you can use any type.
2. This is a tool. You can see it takes a `RunContextWrapper[UserInfo]`. The tool implementation reads from the context.
3. We mark the agent with the generic `UserInfo`, so that the typechecker can catch errors (for example, if we tried to pass a tool that took a different context type).
4. The context is passed to the `run` function.
5. The agent correctly calls the tool and gets the age.

Agent/LLM context

When an LLM is called, the **only** data it can see is from the conversation history. This means that if you want to make some new data available to the LLM, you must do it in a way that makes it available in that history. There are a few ways to do this:

1. You can add it to the Agent `instructions`. This is also known as a "system prompt" or "developer message". System prompts can be static strings, or they can be dynamic functions that receive the context and output a string. This is a common tactic for information that is always useful (for example, the user's name or the current date).
2. Add it to the `input` when calling the `Runner.run` functions. This is similar to the `instructions` tactic, but allows you to have messages that are lower in the chain of command.
3. Expose it via function tools. This is useful for *on-demand* context - the LLM decides when it needs some data, and can call the tool to fetch that data.
4. Use retrieval or web search. These are special tools that are able to fetch relevant data from files or databases (retrieval), or from the web (web search). This is useful for "grounding" the response in relevant contextual data.