c:

Understanding Generics in Python and Their Use with `TContext`

What Are Generics in Python?

Generics are a feature in Python's `typing` module that enable you to write code that can handle multiple data types in a type-safe way. Instead of duplicating the same logic for different types (e.g., `str`, `dict`, or a custom class), generics let you define flexible and reusable classes or functions using type placeholders like `TypeVar`.

Why Use Generics?

Using generics allows:

- Code reuse without sacrificing type safety.
- Flexibility to work with various data types.
- Better type-checking during development using tools like **MyPy** or **Pyright**.

For example, a generic `Box` class can store items of any type:

```
from typing import TypeVar, Generic

T = TypeVar('T')

class Box(Generic[T]):
    def __init__(self, item: T):
        self.item = item

# Usage
box1 = Box[str]("Hello")
box2 = Box
```

This class can be reused with different data types—no need to write separate versions for each.

Generics in the OpenAI Agents SDK: The Role of `TContext`

In the OpenAI Agents SDK, `TContext` is a generic type variable. It serves as a placeholder for the agent's **context**, which can vary depending on the application. The context could be anything that stores relevant information about the user, session, environment, or task.

Examples of Possible `TContext` Values:

- A simple string: `"User is a student"`
- A dictionary: `{"user": "Ali", "budget": 500}`
- A custom class: `UserProfile(name="Ali", age=30)`

By defining `Agent` as a generic class, developers can pass in any context structure they need while still getting the benefits of static typing.

Code Example: Using Generics with `TContext`

```
from typing import TypeVar, Generic

TContext = TypeVar("TContext")

class Agent(Generic[TContext]):
    def __init__(self, instructions: str, context: TContext):
        self.instructions = instructions
        self.context = context

# Example 1: String context
agent1 = Agent[str](
    instructions="Act as a teacher",
    context="User is a student"
)
print(agent1.context)

# Example 2: Dictionary context
agent2 = Agent[dict](
    instructions="Create a travel plan",
    context={"user": "Ali", "budget": 500}
)
print(agent2.context)
```

Output:

```
User is a student
{'user': 'Ali', 'budget': 500}
```

Analogy to Understand Generics

Think of **generics like a universal remote control**. Instead of needing different remotes for your TV, AC, or sound system, you have one remote that can be configured for any device. In the same way, a generic type like `TContext` lets one class (`Agent`) work with any kind of context (data type).

Benefits of Using Generics for `TContext`

- **Flexibility:** Developers can define the context type based on their application's needs.
- **Type Safety:** Tools can catch errors at development time if context types are misused.
- **Reusability:** One implementation of `Agent` can be reused across many projects with different data structures.

Conclusion

Generics in Python—especially with the use of `TContext` in the OpenAI Agents SDK—make your code more adaptable and safer. They are particularly helpful when building frameworks that must support a variety of input types without rewriting core logic for each case.

Author

Written by Humaiza naz