🧠 What Are Generics in Python (and the SDK)?

In programming, **Generics** mean:

> "I want to work with some data, but I don't want to lock it into a single type (like just str or int). I want it to be flexible — but still type-safe."

In **Python**, generics are enabled via the `typing` module:

python
CopyEdit
```
from typing import TypeVar, Generic
```

In **OpenAI Agents SDK**, they help you define agent-related components (like tools, memory, etc.) that can work with **different types of data**, **without losing type safety**.

🧱 Real Example Analogy:

Imagine a **"Box"**:

- A `Box[str]` holds a string
- A `Box[int]` holds a number
- A generic `Box[T]` can hold *anything*, while still tracking what type it's holding

Same goes for an **Agent**, a **Tool**, or even **Memory** — you want to allow agents to use **different tools**, or store **different memory types**, but still be **type-aware**.

📦 In OpenAI Agents SDK — Where Generics Are Used

✅ 1. Tools

Each tool may return **different output types** (e.g., string, number, dict, etc.)

python
CopyEdit
```
from typing import TypeVar, Generic
```

```
T = TypeVar("T")  # generic return type

class Tool(Generic[T]):
    def __call__(self, *args, **kwargs) -> T:
        ...
```

🧠 This allows you to build tools like:

python
CopyEdit
```
class WeatherTool(Tool[str]):
    def __call__(self, location: str) -> str:
        return "It's sunny in Karachi"

class PriceTool(Tool[float]):
    def __call__(self, product: str) -> float:
        return 999.99
```

🔍 So now your agent knows:

- `WeatherTool` will return a `str`
- `PriceTool` will return a `float`

➡️ Helps with **auto-completion**, **error detection**, and **function chaining** in complex agent logic.

✅ 2. Memory Systems

Memory can also be generic:

python
CopyEdit
```
MemoryType = TypeVar("MemoryType")

class Memory(Generic[MemoryType]):
    def load(self) -> MemoryType:
        ...
```

This helps when your memory returns:

- A **string of chat history**
- A **dictionary** of structured user data
- A **custom class** like `ConversationSummary`

Each use case stays type-aware ✅

✅ 3. Agent Inputs & Outputs

When agents or functions accept **dynamic types of context**, generics help preserve flexibility **without losing clarity**.

python
CopyEdit
```
InputType = TypeVar("InputType")
OutputType = TypeVar("OutputType")

class Agent(Generic[InputType, OutputType]):
    def __call__(self, input: InputType) -> OutputType:
        ...
```

So you can build:

- `Agent[str, str]` –> Takes a string, returns a string
- `Agent[dict, str]` –> Takes JSON context, returns a message
- `Agent[Image, str]` –> Takes image, returns caption

➡️ Agents become **adaptable to different tasks**, but still type-safe!

✨ Benefits of Using Generics in SDK

Feature	Benefit
✅ Type Safety	Detect errors before runtime (e.g., wrong return type)
🧠 Intellisense	Better auto-complete and type hints in VSCode, PyCharm

🔀 Flexibility	Same `Tool` or `Agent` base class supports many kinds of logic
🚫 Fewer Bugs	Prevent mismatched types or misused outputs
📚 Better Docs	Types act as implicit documentation for your classes
🤖 AI Assistant Design	Supports multi-modal agents, different memory formats, tools, etc.

📌 Example: Building Your Own Generic Tool

python
CopyEdit
```
from typing import TypeVar, Generic

T = TypeVar("T")

class MyTool(Generic[T]):
    def __call__(self, query: str) -> T:
        ...
```

Use Case:

python
CopyEdit
```
class SummaryTool(MyTool[str]):
    def __call__(self, query: str) -> str:
        return "Here's your summary."
```

Python knows `MyTool[str]` returns a `str` — not anything else!

🧪 Exam-Style Summary

Term	Definition	Example
`Generic[T]`	A class or function that works with any type `T`	`Tool[str]`, `Agent[dict, str]`
`TypeVar`	Declares a type variable for generics	`T = TypeVar("T")`

Agent/Tool/Memory with generics	Flexible but type-checked implementations	Supports multimodal or varied logic

---

🧭 When to Use Generics in Your Own Agent Code?

Use when:

- You're building tools with different kinds of outputs
- You want your memory to be flexible (e.g., str vs dict)
- Your agent takes complex context or dynamic inputs
- You want robust type-checking while maintaining flexibility

🧠 What is `TContext`?

➤ `TContext` is a type variable that represents the "context" data passed to or used by an agent — but in a generic way.

python
CopyEdit
```
from typing import TypeVar

TContext = TypeVar("TContext")
```

This means:

- You don't lock context to just a `dict`, `str`, or `None`.
- You say: "Context can be **any type**, and I'll track what type it is using generics."

📦 Why Is Context Important in Agents?

Agents often rely on **context** to make better decisions.
Context might include:

- User session data

- Chat history
- Current topic/task
- External memory
- Preferences or locale

But context isn't always the same type! Sometimes it's:

- `dict` for structured metadata
- `str` for simple summaries
- `CustomContext` class for rich behavior

That's where `TContext` shines 🔍

💡 Real-Life Analogy: `TContext` = Personal Info for AI

Imagine you're building a **custom assistant**.

- For a **shopping assistant**, context = cart, budget, location
- For a **coding assistant**, context = programming language, user skill level
- For a **travel planner**, context = country, preferences, visa info

You don't want to build **separate agent classes** for each — just use **TContext**!

✅ Example: Using `TContext` in a Generic Agent

python
CopyEdit
```
from typing import TypeVar, Generic, Callable

TContext = TypeVar("TContext")

class Agent(Generic[TContext]):
```

```
    def __init__(self, instructions: Callable[[TContext], str]):
        self.instructions = instructions

    def get_prompt(self, context: TContext) -> str:
        return self.instructions(context)
```

🔁 Now define a specific context:

python
CopyEdit

```
@dataclass
class UserContext:
    name: str
    topic: str

def context_based_prompt(ctx: UserContext) -> str:
    return f"You're teaching {ctx.name} about {ctx.topic}. Be
friendly."

my_agent = Agent[UserContext](instructions=context_based_prompt)

ctx = UserContext(name="Hafsa", topic="Generics in Python")
print(my_agent.get_prompt(ctx))
```

🔽 Output:

vbnet
CopyEdit

```
You're teaching Hafsa about Generics in Python. Be friendly.
```

Perfect dynamic system prompt 🎯
With **full type safety**, your IDE knows exactly what `ctx` is.

📌 Why `TContext` Matters

Feature	Benefit
✅ Flexibility	You can pass any shape/context of data

🔒 Type Safety	Your IDE ensures that the `context` matches your expected structure
🧠 Smart Prompts	Enables personalized or context-aware agents
🚀 Reusability	One agent class works with many types of contextual logic

---

🧪 Advanced Usage with Tools or Memory

python
CopyEdit
```
from typing import TypeVar, Generic, Protocol

TContext = TypeVar("TContext")

class ContextAwareTool(Protocol[TContext]):
    def use(self, context: TContext) -> str: ...
```

Now you can plug in **different tools** depending on what context your agent uses.

🧭 When Should You Use `TContext`?

Use it when:

- You want your agent to be context-aware
- Context structure might vary (user data, session info, etc.)
- You want to keep things **flexible but strongly typed**
- You're building reusable, composable agent classes and tools

---

🎯 Exam-Friendly Recap:

Concept	Summary
`TContext`	A generic type variable representing any context type

Purpose	Allows agents/tools to accept different structured contexts
Benefit	Type-safe, flexible, reusable, IDE-supported design
Example	`Agent[UserContext]`, `Tool[SessionContext]`
Real Use Case	Personalized agents based on user data or memory