Temporal Training Session 2: Workflows in Python

Table of Contents

Welcome and Recap

Welcome to Day 2 of our Temporal training. Yesterday, we established the "why" behind Temporal—the evolution from monolithic systems to microservices and the critical need for orchestration. We introduced Temporal's core concepts: the server, workers, task queues, and the fundamental distinction between workflows and activities. Today, we dive deeper into the heart of Temporal: the Workflow itself, with a practical focus on its implementation in Python.

Recap: What is a Workflow in Temporal?

From Day 1, we learned that a Workflow is a sequence of business logic defined as code, orchestrating the execution of tasks or activities. But let's refine that definition. A Temporal Workflow is a **durable, reliable, and replayable function execution**.

- **Durable:** The state of your workflow is preserved, allowing it to run for seconds, days, or even years, unaffected by process or server crashes
- **Reliable:** Temporal guarantees that your workflow logic will run to completion, managing retries and failures behind the scenes
- **Replayable:** This is a cornerstone concept we will explore today. Every workflow execution has a complete event history, which Temporal uses to recover the state after a failure

Think of a workflow not as a static script, but as a durable entity that orchestrates tasks, reacts to events, and whose state is constantly persisted.

Workflow Functions vs. Activities

This is one of the most critical distinctions to master when working with Temporal. While both are implemented as functions (or methods) in your code, they serve fundamentally different purposes and have different rules.

Aspect	Workflow Functions	Activities
Purpose	To **orchestrate** the business logic. They call Activities, handle logic, manage state, and react to timers or signals.	To **execute** a single, well-defined task. This is where you interact with the outside world.
Execution	Executed by a Worker, but their state is managed and replayed by the Temporal system. They must be **deterministic**.	Executed by a Worker. They are the "side effects" of your system. They can be **non-deterministic**.
Interaction	**Cannot** directly call external APIs, access filesystems, or use libraries that produce side effects (like network calls or random number generation).	**Can and should** perform all I/O, interact with databases, call third-party APIs, and process data.
State	State is implicitly and durably managed by Temporal through its event history.	Are considered stateless from the workflow's perspective. Any state they need must be passed in as input.
Example	A `LoanApprovalWorkflow` function that calls a `creditCheck` activity, then a `verifyDocuments` activity, and based on the results, an `approveLoan` activity.	A `creditCheck` function that makes an API call to a credit bureau.

The golden rule is: **Workflows orchestrate, Activities execute.** Your workflow is the conductor of an orchestra; it tells the musicians (Activities) what to play and when, but it doesn't play the instruments itself.

Determinism: Why It Matters and Common Pitfalls

The concept of "replay" is central to how Temporal achieves fault tolerance. When a Worker running a workflow crashes, a new Worker picks up the task. To restore the workflow's state, Temporal replays the event history recorded by the History Service. It feeds the historical events (e.g., "Activity X completed and returned result Y," "Timer Z fired") back into the workflow function.

For this replay to work correctly, the workflow code **must be deterministic**. This means that given the same sequence of input events, the workflow must always produce the exact same sequence of commands.

If your workflow code behaves differently on replay (e.g., by generating a new random number or checking the current system time), the sequence of commands it produces will no longer match the history, leading to a **non-deterministic error**, and the workflow execution will fail.

Common Pitfalls of Determinism in Python:

- **Using `random`:** Never use Python's standard `random` library inside a workflow

    - **Incorrect:** `random.randint(1, 100)`
    - **Correct:** Use Temporal's APIs for deterministic random value generation
- **Using `datetime.now()` or `time.time()`:** The current time is different every time you run the code
    - **Incorrect:** `print(f"Starting process at {datetime.now()}")`
    - **Correct:** Use Temporal's APIs for operations like time to get the time from the workflow history
- **Direct I/O:** Reading files, making network calls (`requests.get`), or accessing databases directly within a workflow is forbidden. All such operations must be moved into Activities
- **Relying on Global State or Environment Variables:** These can change between the original execution and the replay
- **Using non-deterministic dictionary iteration (in Python < 3.7):** While modern Python versions have deterministic dictionary iteration, it's a good practice to be aware of this

Think of it this way: A workflow function has already run. When it's being replayed, Temporal is just fast-forwarding through the history to rebuild its state. Any new decisions made during this replay will break the historical record.

Workflow Lifecycle and History Replay

A workflow execution, often called a "run," progresses through several stages, all meticulously tracked in its Event History.

1. **Start:** A client initiates a workflow execution. This creates the first event in the history, `WorkflowExecutionStarted`, which contains the initial input
2. **Execution & Task Scheduling:** The workflow function begins to execute. When it encounters a command, like calling an Activity or starting a timer, it doesn't execute it directly. Instead, it yields a command to the Temporal SDK. The SDK sends this command to the Temporal Server
    - *Example:* When you call `await activities.my_activity()`, your workflow pauses, and a command to schedule `my_activity` is sent to the server
3. **State Persistence:** The Temporal Server receives the command and writes an event to the history (e.g., `ActivityTaskScheduled`)
4. **Activity Execution:** The Matching Service places the activity task on the appropriate Task Queue. A Worker polls this queue, executes the activity, and reports the result back to the server
5. **State Update & Resumption:** The server receives the activity result and writes another event to the history (e.g., `ActivityTaskCompleted`). It then schedules a workflow task to notify the workflow of this completion. A Worker picks up this workflow task, and Temporal resumes the workflow function from where it left off, feeding it the result of the activity
6. **Completion:** This cycle continues until the workflow function returns or throws an exception. The final event, `WorkflowExecutionCompleted` or `WorkflowExecutionFailed`, is recorded

The Magic of History Replay

Now, imagine the Worker running the workflow crashes right after scheduling an activity but before it completes.

1. The workflow execution times out on that Worker
2. The Temporal Server schedules a new workflow task
3. A different Worker (or the same one after a restart) picks up this task

4. This new Worker has no in-memory state for the workflow. So, it fetches the full event history from the server
5. The Temporal SDK then **replays** the workflow function. It executes the code from the beginning, but it doesn't re-issue commands that are already in the history
   - When it reaches the line `await activities.my_activity()`, the SDK sees the `ActivityTaskScheduled` event in the replayed history. It knows this command was already sent
   - If the history also contains the `ActivityTaskCompleted` event, the SDK will skip scheduling the activity altogether and immediately return its historical result to the workflow function
   - If the activity hasn't completed yet, the SDK simply notes that the command was issued and the workflow remains paused, waiting for the completion event
6. This replay process restores the workflow to its exact state before the crash, allowing it to continue reliably

---

Creating and Running a Python Workflow (Conceptual)

Let's walk through the structure of a simple workflow in Python.

1. Defining an Activity

Activities are often defined in a separate file (e.g., `activities.py`). They are simple Python functions decorated with `@activity.defn`.

```python
# activities.py
from temporalio import activity

@activity.defn
async def compose_greeting(name: str) -> str:
    activity.logger.info(f"Composing greeting for {name}...")
    return f"Hello, {name}!"
```

2. Defining the Workflow

Workflows are classes that inherit from Workflow and have a main entry point method decorated with `@workflow.run`.

```python
# workflows.py
from datetime import timedelta
from temporalio import workflow
# Import activity definitions
from .activities import compose_greeting

@workflow.defn
class GreetingWorkflow:
    @workflow.run
    async def run(self, name: str) -> str:
        workflow.logger.info(f"Workflow started for name: {name}")
        result = await workflow.execute_activity(
            compose_greeting,
            name,
            start_to_close_timeout=timedelta(seconds=10),
        )
        return result
```

Key elements:

- `@workflow.defn`: Marks the class as a workflow definition
- `@workflow.run`: Designates the entry point for the workflow
- `workflow.execute_activity()`: The command to schedule and execute an activity. Note that we pass the activity function itself, not a string name

---

Using await, Workflow.sleep, and Handling I/O

The `async`/`await` syntax in Python is central to how Temporal orchestrates tasks.

await in Workflows

When you `await` something in a Temporal Workflow, you are not just waiting for an I/O operation like in typical asyncio code. You are yielding control back to the Temporal SDK and waiting for an event from the Temporal Server.

- `await workflow.execute_activity(...)`: Pauses the workflow until the activity completes, fails, or times out
- `await workflow.sleep(...)`: Pauses the workflow for a durable, specified amount of time
- `await workflow.wait_for_condition(...)`: Pauses the workflow until a certain condition becomes true

Workflow.sleep

Never use `time.sleep()` in a workflow. It blocks the entire worker thread and is not durable. A crash would forget the sleep.

`Workflow.sleep(duration)` is the deterministic and durable alternative. It creates a timer on the Temporal Server. The workflow execution is safely suspended and will be resumed by the server only after the timer fires, even if the worker restarts in the meantime.

```
# workflows.py
from datetime import timedelta
from temporalio import workflow
from .activities import compose_greeting

@workflow.defn
class GreetingWorkflow:
    @workflow.run
    async def run(self, name: str) -> str:
        workflow.logger.info("Waiting for 10 seconds before starting...")
        # Durable sleep for 10 seconds
        await workflow.sleep(10)
        workflow.logger.info("10 seconds have passed. Executing activity.")
        result = await workflow.execute_activity(
            compose_greeting,
            name,
            start_to_close_timeout=timedelta(seconds=10),
        )
        return result
```

Handling Workflow Input/Output

Input: Workflow input is passed as arguments to the `@workflow.run` method when the workflow is started by the client. These arguments are serialized and stored in the `WorkflowExecutionStarted` event.

Output: The return value of the `@workflow.run` method is the result of the workflow. It is serialized and recorded in the `WorkflowExecutionCompleted` event. The client that started the workflow can then fetch this result.

This structured approach to I/O ensures that all data entering and leaving the workflow is captured in the event history, making the entire execution fully auditable and replayable.