Effortless Performance Profiling for Python

An introduction to the `timestamp_capture` utility.

Simple: Add a single decorator.

Insightful: Pinpoint exact bottlenecks.

Production-Safe: Negligible overhead when inactive.

From `osbot_utils` | `pip install osbot-utils`

Find a Bottleneck in Under 60 Seconds

1. Decorate

```
from osbot_utils.helpers.timestamp_capture
import Timestamp_Collector, timestamp

@timestamp
def slow_function():
    # ... complex logic ...

@timestamp
def main_process():
    slow_function()
```

Add `@timestamp` to the methods you want to measure.

2. Capture

```
# In your main script...
_timestamp_collector_ = Timestamp_Collector()
main_process()
_timestamp_collector_.print_report()
```

Create the collector with a specific variable name. That's it.

3. Analyse

```
-=[ Timing Report ]=-
Target          | Total Time
----------------|-----------
main_process    | 150.23ms
slow_function   | 145.12ms
```

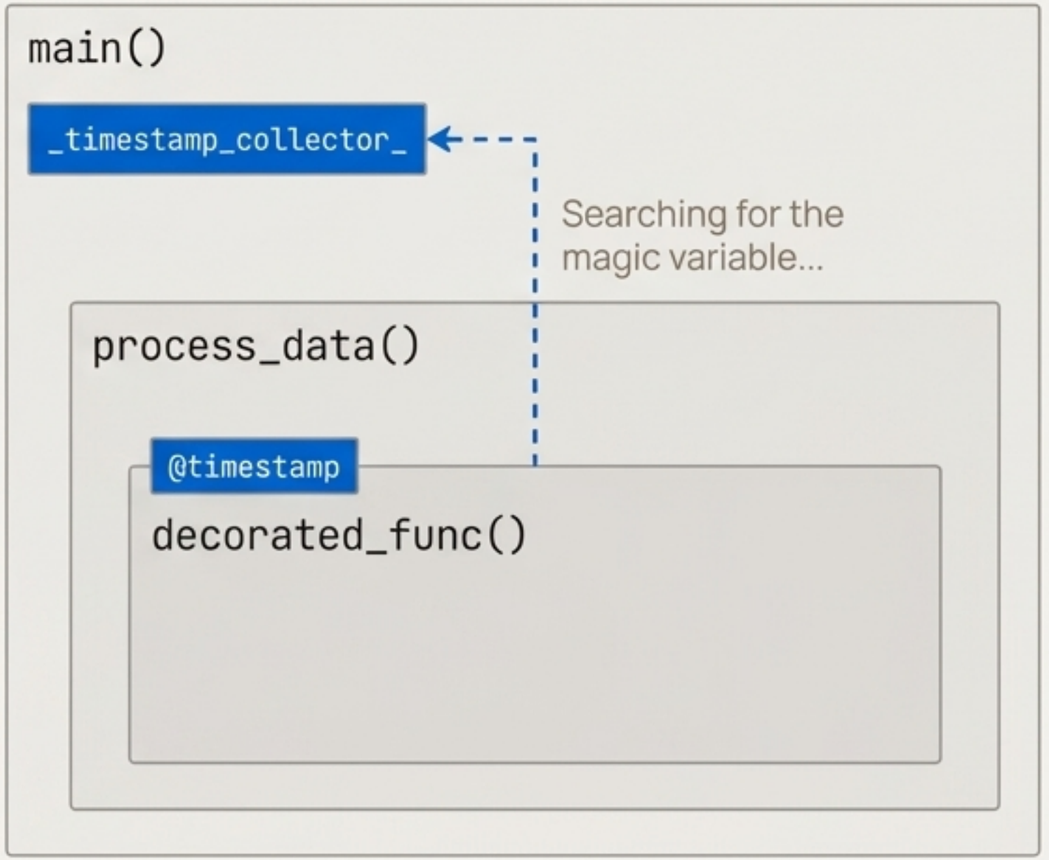
How It Works: The Magic is in the Name

The `@timestamp` decorator doesn't need to be told where the collector is. It finds it automatically.

Stack-Walking: The decorator walks up the call stack, frame by frame.

The Magic Variable: It actively looks for a local variable named *exactly* `_timestamp_collector_`.

Result: This decouples your business logic from your instrumentation code. No need to pass a `collector` object through every function signature.



Designed to Be Left in Production Code

The decorator is dormant if no collector is found, with minimal overhead.

Scenario	Overhead per Call	Production Safe?
`@timestamp (Inactive)` No `_timestamp_collector_` found in stack.	`~3 µs`	**Yes.** Negligible impact.
`@timestamp (Active)` Collector is present and capturing data.	`~8 µs`	**Yes.** Ideal for methods >100µs.
Nested Decorators N levels of decorated calls.	`N × overhead`	**Yes.** Overhead scales linearly and predictably.

Rule of Thumb: If your method takes over 100µs to execute, the active capture overhead is less than 8%. This is acceptable for most production profiling scenarios.

Profiling More Than Just Methods

Sometimes the code you need to measure isn't a single function. Use a context manager to profile arbitrary blocks of code.

For Methods

```
# Use the decorator for entire methods
@timestamp
def data_processing_pipeline():
    # ...
```

For Code Blocks

```
# Use the block for specific phases
def data_processing_pipeline():
    # ... setup code (not timed)
    with timestamp_block("data.extraction"):
        # ... extraction logic
    with timestamp_block("data.transformation"):
        # ... transformation logic
    # ... teardown code (not timed)
```

Use Cases:

- Timing specific phases within a large function.
- Profiling code inside a loop without decorating the looped function.
- Measuring I/O or network operations that are not in their own methods.

Go Beyond Function Names for Clearer Reports

Default method names (`__qualname__`) can be long or uninformative. For complex systems, you need better organisation.

Use the **`name`** parameter to define a **custom, hierarchical metric name**.

Decorator Example:

```
# Instead of the default 'process_user_data'
@timestamp(name="pipeline.stage1.extract")
def process_user_data(user_id):
    # ...
```

Block Example:

```
with timestamp_block(name="io.read.open_file"):
    # ...
```

Benefit: This allows you to group related operations, version algorithms, and build reports that mirror your system's architecture, not just its code structure.

A Practical Guide to Naming Your Timestamps

A consistent naming convention is key to building insightful reports. Use a dot-separated format to create a logical hierarchy.

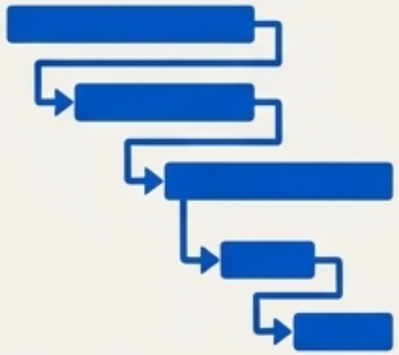
Pattern	Example	Typical Use Case
`domain.operation`	`parser.parse_xml`	Simple, effective categorisation.
`component.stage.action`	`pipeline.stage1.extract`	Tracking stages in a data pipeline.
`feature.version.method`	`algorithm.v2.sort`	A/B testing or comparing implementations.
`io.category.operation`	`io.read.open_file`	Grouping related operations like I/O.

“Key Insight: Treat your metric names as a form of documentation. They should clearly communicate the purpose and context of the code being measured.”

Turning Raw Data into Actionable Insight

`timestamp_capture` provides three distinct views into your performance data, each designed to answer a different question.

The Timeline



What is the sequence of events?

`print_timeline()`

Shows the chronological flow and call hierarchy.

The Hotspots

Where is the CPU actually spending its time?

`print_hotspots()`

Ranks methods by the work done *inside* them.

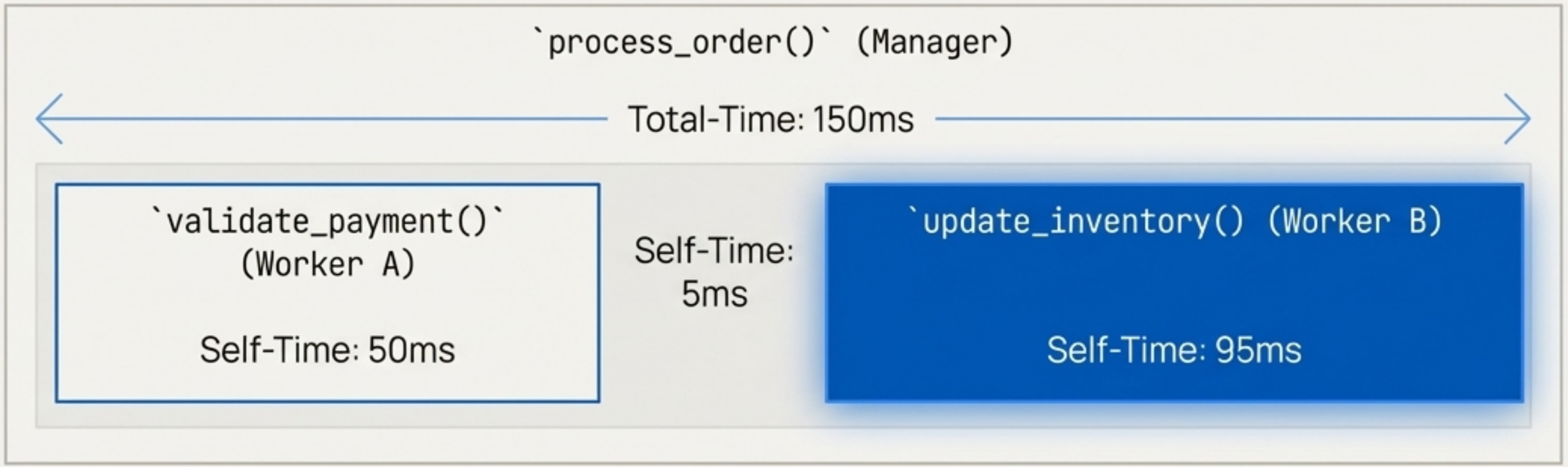
The Full Report

What are the complete timing details?

`print_report()`

A comprehensive table of all captured metrics.

The Most Important Concept: Self-Time vs. Total-Time



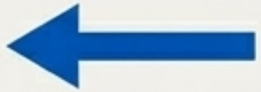
Total-Time: The entire time spent in a method, *including* all the time spent in other methods it calls (its children). A 'busy manager'.

Self-Time: The time spent executing code *directly within* a method, *excluding* time spent in its children. This is the actual work done. **This is what you need to optimise.**

Find the Real Bottleneck with the Hotspots Report

Don’t be misled by high Total-Time. The `print_hotspots()` report sorts methods by **Self-Time** to show you where your program is actually spending its clock cycles.

```
-=[ Hotspots Report (by self time) ]=-
Target              | Self Time   | Total Time  | ...
--------------------|-------------|-------------|------
update_inventory    | 95.0ms      |  95.0ms     | ...
validate_payment    | 50.0ms      |  50.0ms     | ...
process_order       | 5.0ms       | 150.0ms     | ...
```



Interpretation

In this example, `process_order` has the highest Total-Time (150ms), but it’s a red herring. The real work, and the place to start optimising, is `update_inventory`, which has the highest Self-Time.

Editorial Blueprint

Visualise Your Code's Execution Journey

The `print_timeline()` report provides a chronological, indented view of your program's execution, making it easy to understand complex interactions.

```
▶ root.process_order          (t=0.00ms)
  ▶ a.validate_payment        (t=2.05ms)
  ◀ a.validate_payment        (t=52.15ms)
  ▶ b.update_inventory        (t=52.20ms)
    ▶ c.db_connection         (t=53.10ms)
    ◀ c.db_connection         (t=73.50ms)
  ◀ b.update_inventory        (t=147.30ms)
◀ root.process_order          (t=150.00ms)
```

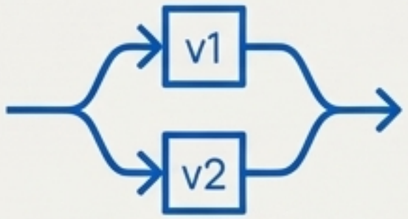
Use this report to:

- Understand execution flow and method call hierarchy at a glance.
- Identify unexpected or recursive call sequences.
- Spot methods that are called far more times than you expect.
- Visualise where time is spent in deeply nested operations.

Putting It All Together: Common Scenarios

Profiling a Data Pipeline

- **Goal:** Find the slowest stage in a multi-step data conversion process.
- **Method:** Use `@timestamp(name="pipeline.stage.X")` on each major function. Use `print_hotspots()` to immediately identify the bottleneck stage by its high Self-Time.



Comparing Implementations

- **Goal:** A/B test a new algorithm against an old one.
- **Method:** Decorate both with `name="alg.v1.process"` and `name="alg.v2.process"`. Run both and compare their average execution times in the final `print_report()`.

Continuous Performance Monitoring

- **Goal:** Track performance over time and catch regressions.
- **Method:** Leave decorators in production code. Use `get_method_timings()` to programmatically access the data and send it to a monitoring service (e.g., StatsD, Prometheus).

Best Practices for Effective Profiling

DO ☑️

- **Decorate entry points and key methods**: Start at a high level to understand the overall flow.
- **Use descriptive collector names**: `_timestamp_collector_ = Timestamp_Collector("Process X")`.
- **Use `timestamp_block` for phases**: Isolate and time important logic within larger methods.
- **Use `print_hotspots()` first**: It's the fastest way to find actionable bottlenecks.

DON'T ❌

- **Decorate micro-functions in hot loops**: The overhead can become significant and create noise. Profile the loop itself instead.
- **Forget the magic variable name**: It must be exactly `_timestamp_collector_`.
- **Misinterpret Total-Time**: Don't optimise a method with high Total-Time but low Self-Time.

Troubleshooting Common Issues

Problem: Decorator isn't capturing any data.

Likely Cause: The collector variable is not named exactly `_timestamp_collector_` or is out of scope.

Solution: Check for typos. Ensure the collector is defined in a frame that is an ancestor of the decorated function call.

Problem: A method shows 0ms Self-Time.

Likely Cause: This is expected if the method only calls other decorated methods and performs no other significant work itself. It's a 'manager' method.

Solution: This isn't an error. Look at the methods it calls to find the real work.

Problem: The report is too noisy, with too many entries.

Likely Cause: You have decorated a small utility function that is called thousands of times inside a loop.

Solution: Remove the decorator from the inner function. Instead, wrap the entire loop in a `timestamp_block` to measure its total impact.

Your Checklist for Codebase Instrumentation

Use these steps to integrate `timestamp_capture` into your project today.

- [ ] **Import** `@timestamp` and `Timestamp_Collector`.
- [ ] **Add** `@timestamp` to key methods (entry points, major processing steps).
- [ ] **Use** `name="..."` for hierarchical naming where clarity is needed.
- [ ] **Use** `timestamp_block` for code phases not in dedicated methods.
- [ ] **Create the collector** and name the variable exactly `_timestamp_collector_`.
- [ ] **Run your code** and generate reports.
- [ ] **Start** with `print_hotspots()` to find the real bottlenecks (Self-Time).
- [ ] **Use** `print_timeline()` to understand the execution flow.

Remember: The decorators are production-safe and can be committed to your codebase.