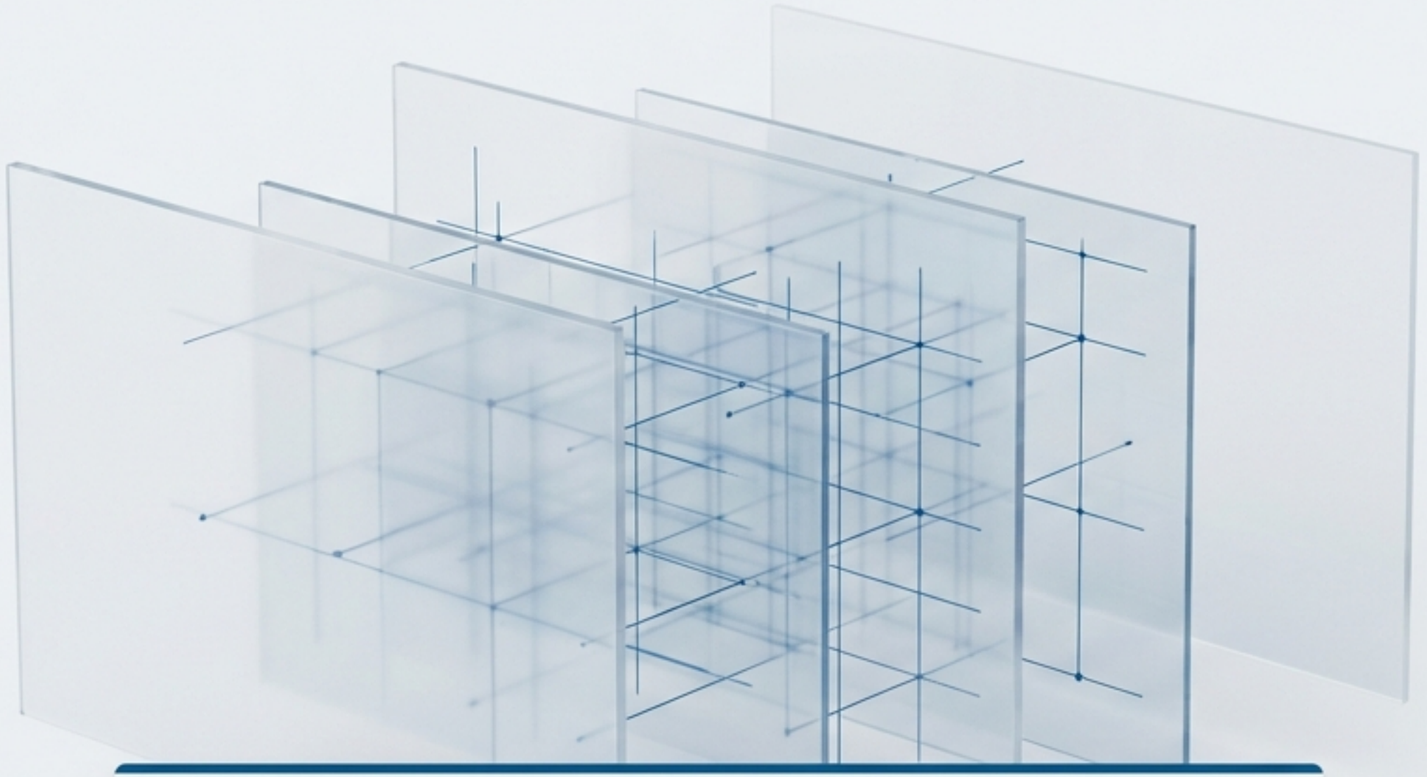
Your Code's X-Ray Machine.

See exactly what your Python code is doing at runtime with `Trace_Call`.



Tool: `Trace_Call`
Version: v3.59.2
Repository: OSBot-Utils (github.com/owasp-sbot/OSBot-Utils)
Installation: `pip install osbot-utils`

The Code Is a Black Box. We've All Been There.

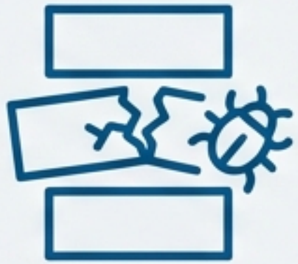
You write and inherit Python code every day. But how often do you face these fundamental questions without good answers?

"What is this code *actually* doing?"

Trying to understand a third-party library or a complex internal service without reading thousands of lines of source.

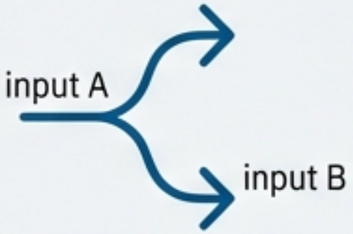
"Where is the time going?"

Your application is slow. Standard profilers give you a flat, confusing list of functions, not the critical execution path.



"What was the state when this broke?"

A bug happens deep in a call stack. You need to know the variable values at each step leading to the failure.



"How do different inputs change execution?"

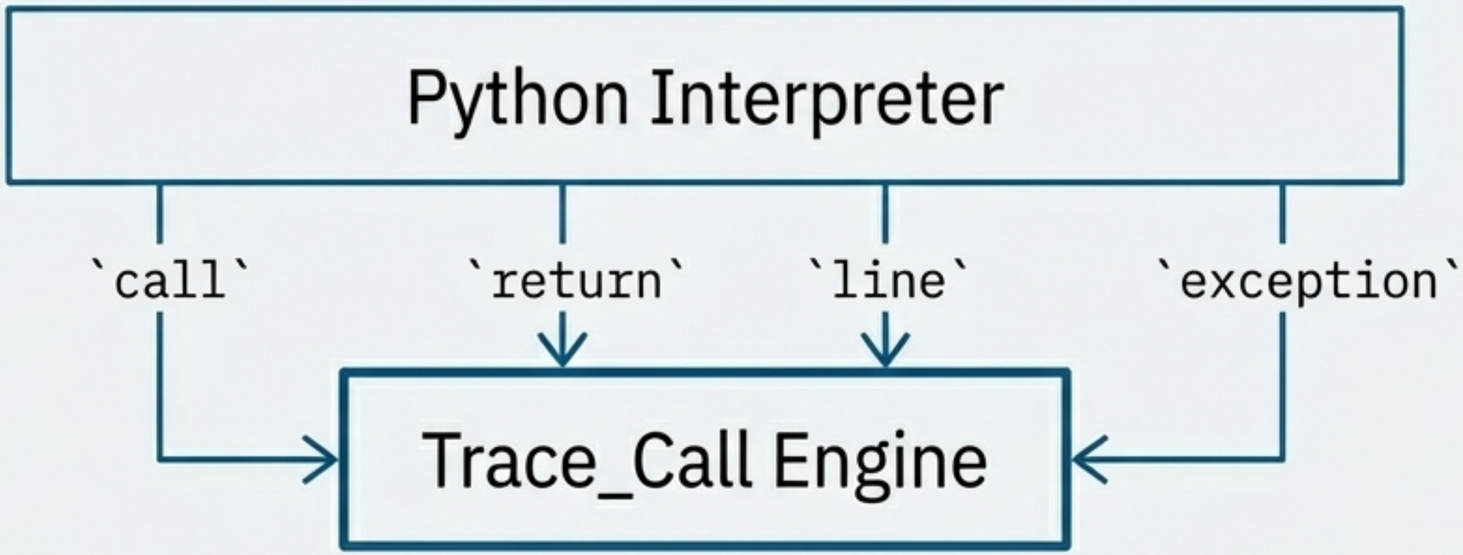
Comparing code paths without littering your codebase with temporary `print` statements.

"But I can't modify this code"

The code you need to inspect is in production, a compiled module, or a package you don't own.

The Solution: Zero-Instrumentation Observability.

Trace_Call is a powerful runtime introspection system that reveals the complete call hierarchy, timing, and variable states of any Python code—**without modifying a single line.**



By leveraging Python's built-in `sys.settrace()` hook, `Trace_Call` sees every operation at the interpreter level. This is why it works on *any* code, from your own scripts to third-party libraries.

See the Whole Story, Not Just a Flat List.

Traditional profilers show you *what* ran. **Trace_Call** shows you *why* and *how*.

Typical Profiler Output

```
process_request ... 0.85s
db_query        ... 0.72s
execute_sql     ... 0.70s
validate_input  ... 0.10s
```

Where is the bottleneck, really? `process_request` is slow, but what's the cause?

`Trace_Call's Hierarchical Tree

```
process_request
├── validate_input
└── db_query
    └── execute_sql
```

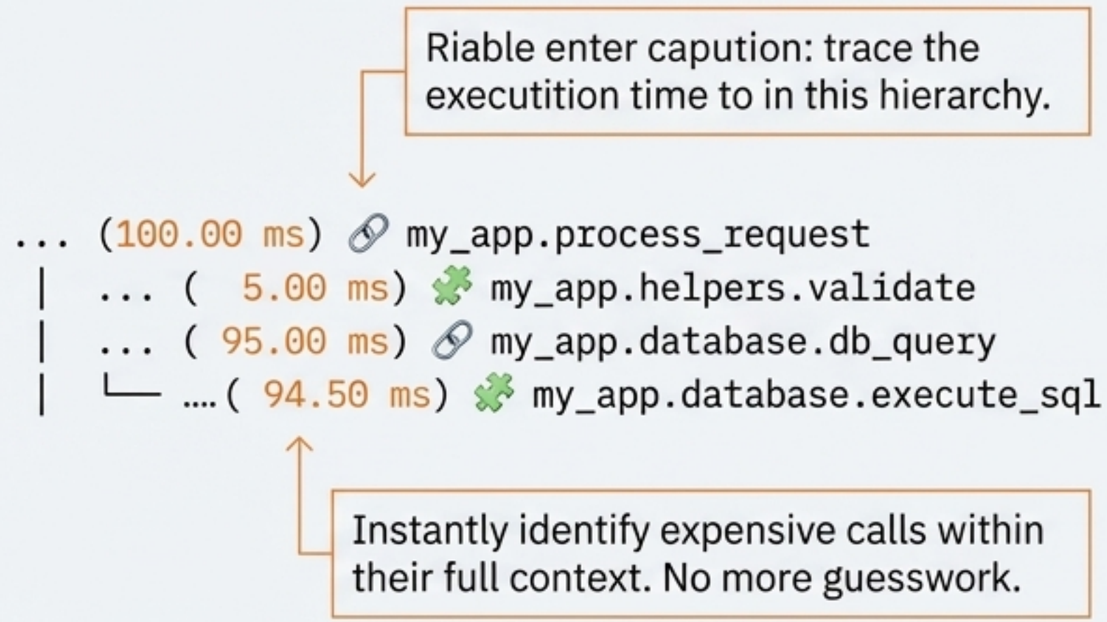
Clarity. You can now *see* that `execute_sql` inside `db_query` is the true source of the delay.

Find the Real Bottleneck with Precision Timing.

Enable `capture_duration` to record the execution time of every single call in the hierarchy. Then, filter out the noise to focus only on what matters.

```
# Find all calls taking longer than 10 milliseconds
with Trace_Call(capture_duration=True,
           with_duration_bigger_than=0.01) as trace:
    slow_function()

trace.print()
```



“Time-Travel” to See the State of Any Function Call.

When a bug occurs, you need to know *what the variables were* at each step. `capture_locals` automatically snapshots them for you.

> *“It’s like having a debugger breakpoint at every function call, captured automatically and reviewable after the code has finished running.”*

```
🔗 my_module.process_user(user_id=123, role='admin')
|   locals: {'user_id': 123, 'role': 'admin', 'user_obj': <User...>}
|
└── 🧩 my_module.check_permissions(user=<User...>, required='admin')
       locals: {'user': <User...>, 'required': 'admin', 'has_perm': True}
```

Use `deep_copy_locals=True` for an accurate snapshot if variables are mutated after the call.

Focus Your View: Surgical Filtering for a Clear Signal.

A single function call can trigger hundreds of internal calls in frameworks, loggers, and libraries. Tracing everything is overwhelming. `Trace_Call` gives you the tools to cut through the noise.

Capture by Prefix

Target specific modules.

```
trace_capture_start_with=['my_app', 'requests']
```

Capture by Content

Find functions or modules by name.

```
trace_capture_contains=['process_data', 'utils']
```

Ignore by Prefix

Exclude noisy libraries.

```
trace_ignore_start_with=['logging', 'osbot_utils.helpers']
```

Control the Depth

Get a high-level overview without getting lost.

```
trace_up_to_depth=4
```

Combine these filters to precisely define your investigation’s scope.

Two Elegant Ways to Activate Tracing.

Integrating `Trace_Call` is clean, Pythonic, and requires no code refactoring.

The Context Manager

For tracing specific blocks of code or third-party calls.

```
from osbot_utils.helpers.trace.Trace_Call
import Trace_Call

with Trace_Call(config, ...) as trace:
    # Code to be traced goes here
    result = complex_library.run()

# Trace data is now in trace.view_data()
```

The Decorator

For tracing a specific function every time it's called.

```
from osbot_utils.helpers.trace.Trace_Call
import trace_calls

@trace_calls(config, ...)
def my_function_to_trace(a, b):
    # ...
```

The tracer is production-safe. When `trace_enabled=False', the overhead is negligible (~microseconds), so decorators can be left in place.

Your Control Panel: Fine-Tuning the Trace.

A selection of key configuration parameters to tailor the tracer to your exact needs.

Filtering

`trace_capture_all`
Capture everything (use with `trace_up_to_depth`).

`trace_capture_start_with`
Target modules by prefix.

`trace_ignore_start_with`
Exclude noisy modules.

`trace_up_to_depth`
Limit recursion depth.

`trace_show_internals`
Include _private methods.

Performance Analysis

`capture_duration`
Record execution time.

`print_duration`
Show timings in the output.

`with_duration_bigger_than`
Filter by execution time.

State Debugging

`capture_locals`
Capture local variables.

`print_locals`
Show locals in the output.

`deep_copy_locals`
Snapshot locals to prevent mutation issues.

Detailed Tracing

`trace_capture_lines`
Capture line-by-line execution.

`trace_capture_source_code`
Capture the source code line itself.

Common Scenarios, Solved.

Here is how to apply `Trace_Call` to solve your most frequent development challenges.

Your Goal	The `Trace_Call` Approach
Understand unfamiliar code	`trace_capture_start_with=['unknown_lib'], trace_up_to_depth=4`
Find performance bottlenecks	`capture_duration=True, with_duration_bigger_than=0.01`
Debug complex variable state	`capture_locals=True, deep_copy_locals=True`
Understand a test failure	Use the `@trace_calls` decorator on the failing test method.
Compare execution paths	Run twice with different inputs, then programmatically compare `view_data()` outputs.
Reverse-engineer an API	Trace the target library while making a series of API calls.

Interpreting the X-Ray: Reading the Output.

The Call Tree (`print_traces`)

Visual Key

📦 **Root node:** The start of the trace session.

🔗 **Node with children:** A function that called other traced functions.

🧩 **Leaf node:** A function that made no other traced calls.

Structure Key

```
│    Standard tree characters showing
│    parent—child hierarchy and siblings.
└── └── [redacted]
```

Line-by-Line Execution (`print_lines`)

For the deepest debugging, this view shows the precise execution path, including which branches were taken.

```
# | Line | Source code | Method | Depth
---|-----|-------------|--------|------
```

The indentation of the `Source code` column visually represents the call stack depth.

Best Practices for Effective Tracing.

DO

- ✅ **Use Specific Filters:** Start with `trace_capture_start_with`. Avoid `trace_capture_all` unless you also use a depth limit.
- ✅ **Limit Depth for Exploration:** Use `trace_up_to_depth` to get a high-level map before diving deep.
- ✅ **Use Duration Filtering for Performance:** `with_duration_bigger_than` is your best tool for isolating slow code.
- ✅ **Ignore Noisy Modules:** Proactively add common modules like `logging` or `caching` to `trace_ignore_start_with`.

DON'T

- ❌ **Leave Tracing Enabled in Production:** While the disabled-state overhead is tiny, active tracing is for development and debugging, not production monitoring.
- ❌ **Capture Locals in Performance-Sensitive Code:** Capturing locals adds overhead. Use it for debugging state, not for performance profiling.
- ❌ **Run Alongside a Debugger:** Both tools use `sys.settrace()`. They cannot run at the same time.

Troubleshooting Common Issues.

Problem: No traces are captured.

- Check your `trace_capture_` filters for typos.
- Ensure `trace_enabled` is `True`.
- Check your `trace_ignore_` rules aren't too broad.

Problem: The output is overwhelming.

- Add a `trace_up_to_depth` limit.
- Be more specific with `trace_capture_start_with`.
- Add `trace_ignore_` patterns.
- Use `with_duration_bigger_than` to hide fast functions.

Problem: Locals show incorrect (mutated) values.

- The default capture is by reference. For an accurate point-in-time snapshot, set `deep_copy_locals=True`.

Problem: Private methods (e.g., `_my_method`) are missing.

- By default, internal methods are hidden for clarity. Set `trace_show_internals=True` to include them.

Your Pre-Flight Checklist for Code Investigation.

Stop guessing. Start seeing. Before your next deep dive into a complex codebase, run through this checklist.

- [ ] **Choose Your Strategy**
  - `trace_capture_start_with` for modules?
  - `trace_capture_all` + `trace_up_to_depth` for exploration?
- [ ] Add **Ignore Patterns** for noisy libraries.
- [ ] Set **Tracing Goal**
  - **Performance**? Enable `capture_duration` and `with_duration_bigger_than`.
  - **Debugging State**? Enable `capture_locals` and `deep_copy_locals`.
  - **Deep Analysis**? Enable `trace_capture_lines`.
- [ ] Apply with **Context Manager** or **Decorator**.
- [ ] **Analyse** the **output**.

Turn the lights on in your code's black box.

```
pip install osbot-utils
```