`!profiling.tracing` --- Deterministic profiler

- What is deterministic profiling?
- Command-line interface
- Programmatic usage examples
  - Basic profiling
  - Using the !Profile class
- Module reference
- Using a custom timer
- Limitations

profiling.tracing

cProfile

3.15

Source code: `Lib/profiling/tracing/`

The `!profiling.tracing` module provides deterministic profiling of Python programs. It monitors every function call, function return, and exception event, recording precise timing for each. This approach provides exact call counts and complete visibility into program execution, making it ideal for development and testing scenarios.

> This module is also available as `cProfile` for backward compatibility. The `cProfile` name will continue to work in all future Python versions. Use whichever import style suits your codebase:

```
# Preferred (new style)
import profiling.tracing
profiling.tracing.run('my_function()')

# Also works (backward compatible)
import cProfile
cProfile.run('my_function()')
```

What is deterministic profiling?

`Deterministic profiling` captures every function call, function return, and exception event during program execution. The profiler measures the precise time intervals between these events, providing exact statistics about how the program behaves.

In contrast to `statistical profiling <profiling-sampling>`, which samples the call stack periodically to estimate where time is spent, deterministic profiling records every event. This means you get exact call counts rather than statistical approximations. The trade-off is that instrumenting every event introduces overhead that can slow down program execution.

Python's interpreted nature makes deterministic profiling practical. The interpreter already dispatches events for function calls and returns, so the profiler can hook into this mechanism without requiring code modification. The overhead tends to be moderate relative to the inherent cost of interpretation, making deterministic profiling suitable for most development workflows.

Deterministic profiling helps answer questions like:

- How many times was this function called?
- What is the complete call graph of my program?
- Which functions are called by a particular function?
- Are there unexpected function calls happening?

Call count statistics can identify bugs (surprising counts) and inline expansion opportunities (high call counts). Internal time statistics reveal "hot loops" that warrant optimization. Cumulative time statistics help identify algorithmic inefficiencies. The handling of cumulative times in this profiler allows direct comparison of recursive and iterative implementations.

Command-line interface

profiling.tracing

The `!profiling.tracing` module can be invoked as a script to profile another script or module:

```
python -m profiling.tracing [-o output_file] [-s sort_order] (-m module | script.py)
```

This runs the specified script or module under the profiler and prints the results to standard output (or saves them to a file).

`-o <output_file>`

Write the profile results to a file instead of standard output. The output file can be read by the `pstats` module for later analysis.

`-s <sort_order>`

Sort the output by the specified key. This accepts any of the sort keys recognized by `pstats.Stats.sort_stats`, such as `cumulative`, `time`, `calls`, or `name`. This option only applies when `-o <profiling.tracing -o>` is not specified.

`-m <module>`

Profile a module instead of a script. The module is located using the standard import mechanism.

3.7 The `-m` option for `cProfile`.

3.8 The `-m` option for `profile`.

Programmatic usage examples

For more control over profiling, use the module's functions and classes directly.

Basic profiling

The simplest approach uses the `!run` function:

```
import profiling.tracing
profiling.tracing.run('my_function()')
```

This profiles the given code string and prints a summary to standard output. To save results for later analysis:

```
profiling.tracing.run('my_function()', 'output.prof')
```

Using the `!Profile` class

The `!Profile` class provides fine-grained control:

```
import profiling.tracing
import pstats
from io import StringIO

pr = profiling.tracing.Profile()
pr.enable()
# ... code to profile ...
pr.disable()

# Print results
s = StringIO()
ps = pstats.Stats(pr, stream=s).sort_stats(pstats.SortKey.CUMULATIVE)
ps.print_stats()
print(s.getvalue())
```

The `!Profile` class also works as a context manager:

```
import profiling.tracing

with profiling.tracing.Profile() as pr:
    # ... code to profile ...

pr.print_stats()
```

Module reference

profiling.tracing

run(command, filename=None, sort=-1)

Profile execution of a command and print or save the results.

This function executes the *command* string using `exec` in the `__main__` module's namespace:

```
exec(command, __main__.__dict__, __main__.__dict__)
```

If *filename* is not provided, the function creates a `pstats.Stats` instance and prints a summary to standard output. If *filename* is provided, the raw profile data is saved to that file for later analysis with `pstats`.

The *sort* argument specifies the sort order for printed output, accepting any value recognized by `pstats.Stats.sort_stats`.

runctx(command, globals, locals, filename=None, sort=-1)

Profile execution of a command with explicit namespaces.

Like `run`, but executes the command with the specified *globals* and *locals* mappings instead of using the `__main__` module's namespace:

```
exec(command, globals, locals)
```

A profiler object that collects execution statistics.

The optional *timer* argument specifies a custom timing function. If not provided, the profiler uses a platform-appropriate default timer. When supplying a custom timer, it must return a single number representing the current time. If the timer returns integers, use *timeunit* to specify the duration of one time unit (for example, `0.001` for milliseconds).

The *subcalls* argument controls whether the profiler tracks call relationships between functions. The *builtins* argument controls whether built-in functions are profiled.

3.8 Added context manager support.

enable()

Start collecting profiling data.

disable()

Stop collecting profiling data.

create_stats()

Stop collecting data and record the results internally as the current profile.

print_stats(sort=-1)

Create a `pstats.Stats` object from the current profile and print the results to standard output.

The *sort* argument specifies the sorting order. It accepts a single key or a tuple of keys for multi-level sorting, using the same values as `pstats.Stats.sort_stats`.

3.13 Support for a tuple of sort keys.

dump_stats(filename)

Write the current profile data to *filename*. The file can be read by `pstats.Stats` for later analysis.

run(cmd)

Profile the command string via `exec`.

runctx(cmd, globals, locals)

Profile the command string via `exec` with the specified namespaces.

runcall(func, /, *args,*kwargs)

Profile a function call. Returns whatever *func* returns:

```
result = pr.runcall(my_function, arg1, arg2, keyword=value)
```

> Profiling requires that the profiled code returns normally. If the interpreter terminates (for example, via `sys.exit`) during profiling, no results will be available.

Using a custom timer

The `Profile` class accepts a custom timing function, allowing you to measure different aspects of execution such as wall-clock time or CPU time. Pass the timing function to the constructor:

```
pr = profiling.tracing.Profile(my_timer_function)
```

The timer function must return a single number representing the current time. If it returns integers, also specify *timeunit* to indicate the duration of one unit:

```
# Timer returns time in milliseconds
pr = profiling.tracing.Profile(my_ms_timer, 0.001)
```

For best performance, the timer function should be as fast as possible. The profiler calls it frequently, so timer overhead directly affects profiling overhead.

The `time` module provides several functions suitable for use as custom timers:

- `time.perf_counter` for high-resolution wall-clock time
- `time.process_time` for CPU time (excluding sleep)
- `time.monotonic` for monotonic clock time

Limitations

Deterministic profiling has inherent limitations related to timing accuracy.

The underlying timer typically has a resolution of about one millisecond. Measurements cannot be more accurate than this resolution. With enough measurements, timing errors tend to average out, but individual measurements may be imprecise.

There is also latency between when an event occurs and when the profiler captures the timestamp. Similarly, there is latency after reading the timestamp before user code resumes. Functions called frequently accumulate this latency, which can make them appear slower than they actually are. This error is typically less than one clock tick per call but can become significant for functions called many times.

The `!profiling.tracing` module (and its `cProfile` alias) is implemented as a C extension with low overhead, so these timing issues are less pronounced than with the deprecated pure Python `profile` module.

`profiling`
Overview of Python profiling tools and guidance on choosing a profiler.

`profiling.sampling`
Statistical sampling profiler for production use.

`pstats`
Statistics analysis and formatting for profile data.

`profile`
Deprecated pure Python profiler (includes calibration documentation).

□□□ #2
□ □□□ □□□ 2026-10-31 17:17:26 UTC
□ □□□ □□□ 2026-10-31 17:17:28 UTC