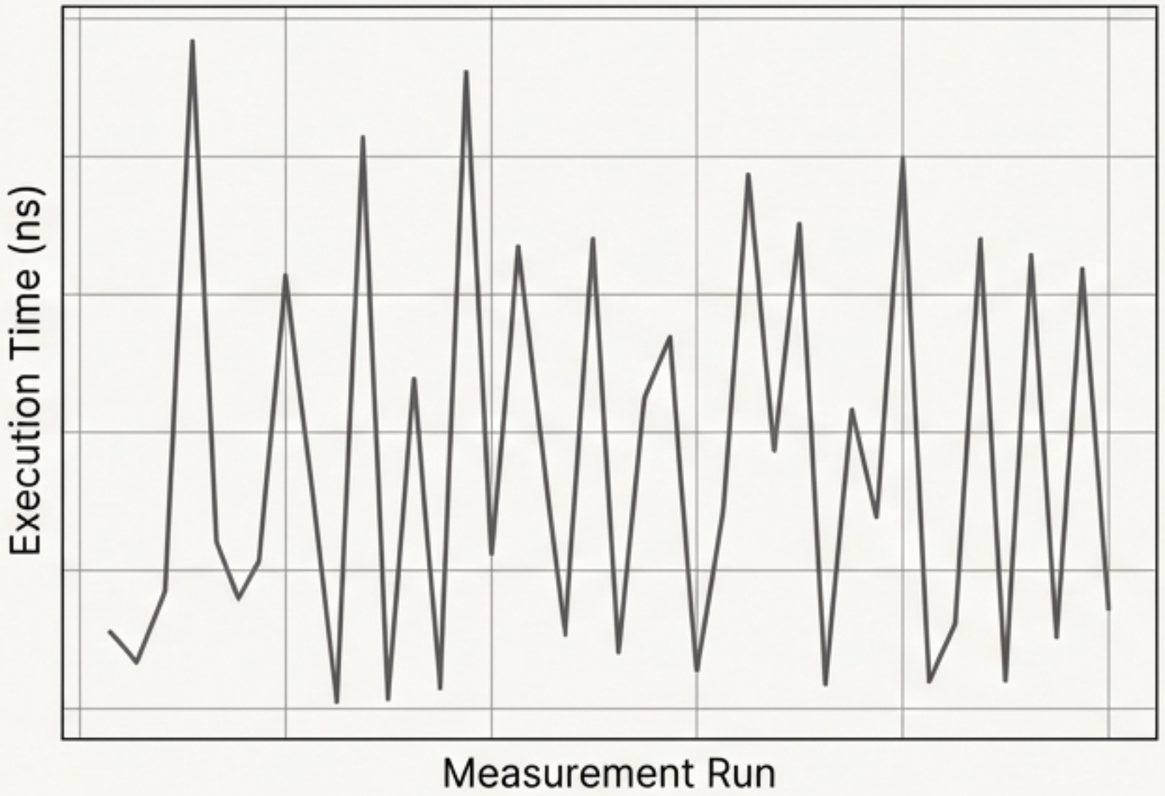
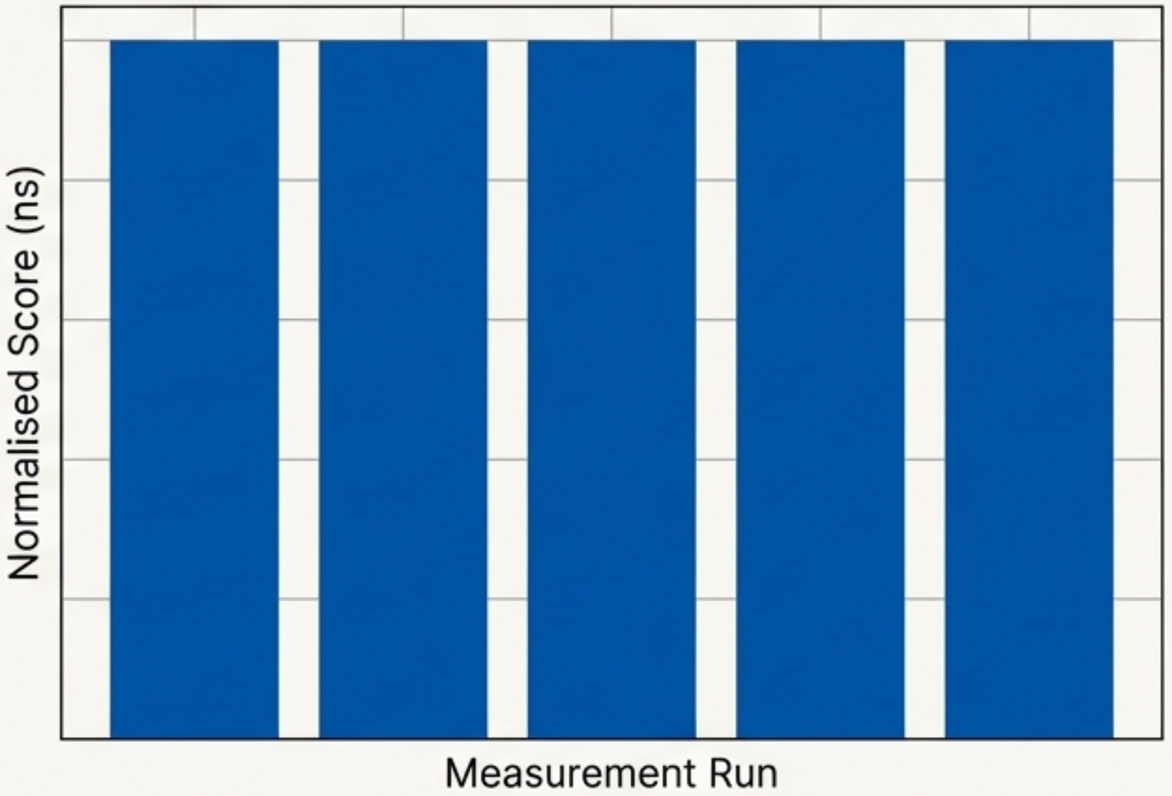
Tired of Noisy Python Benchmarks?

`time.perf_counter()`



`Performance_Measure__Session`



Standard timing is plagued by noise from garbage collection, context switching, and system load. **It's time for a statistically robust approach.**

Stable, Nanosecond-Precision Metrics for Python

Performance_Measure__Session is a high-precision benchmarking framework that produces statistically stable, reproducible performance metrics at nanosecond resolution. It eliminates the noise and variability that plagues typical Python timing approaches.

Version:	`v3.60.1`
Install:	`` `pip install osbot-utils` ``
Source:	`` `osbot_utils.testing.performance` ``
Repo:	github.com/owasp-sbot/OSBot-Utils

The Philosophy Behind Precision.

Statistical Robustness

Eliminates noise by design. Fibonacci-based sampling captures warm-up effects, while intelligent outlier trimming removes GC pauses and other system jitter.

Reproducible Scores

Achieve consistent results across runs. Dynamic score normalisation produces stable values, and CI-aware assertions automatically adjust for slower shared runners.

Zero Ceremony

Focus on your code, not the boilerplate. A clean context manager, fluent API, and sensible defaults get you started in seconds. Fully integrated with `Type_Safe`.

Your First Measurement in Three Lines

```
# 1. Define a function to test
def fibonacci(n):
    if n <= 1: return n
    return fibonacci(n-1) + fibonacci(n-2)

# 2. Import and measure
from osbot_utils.testing.performance import Perf
with Perf() as _:
    _.measure(lambda: fibonacci(10))
```

Console Output:

```
fibonacci(10)        :      2,000 ns
```

That’s it. No manual loops, no averaging, just a clean, stable result.

Assert Performance with Confidence.

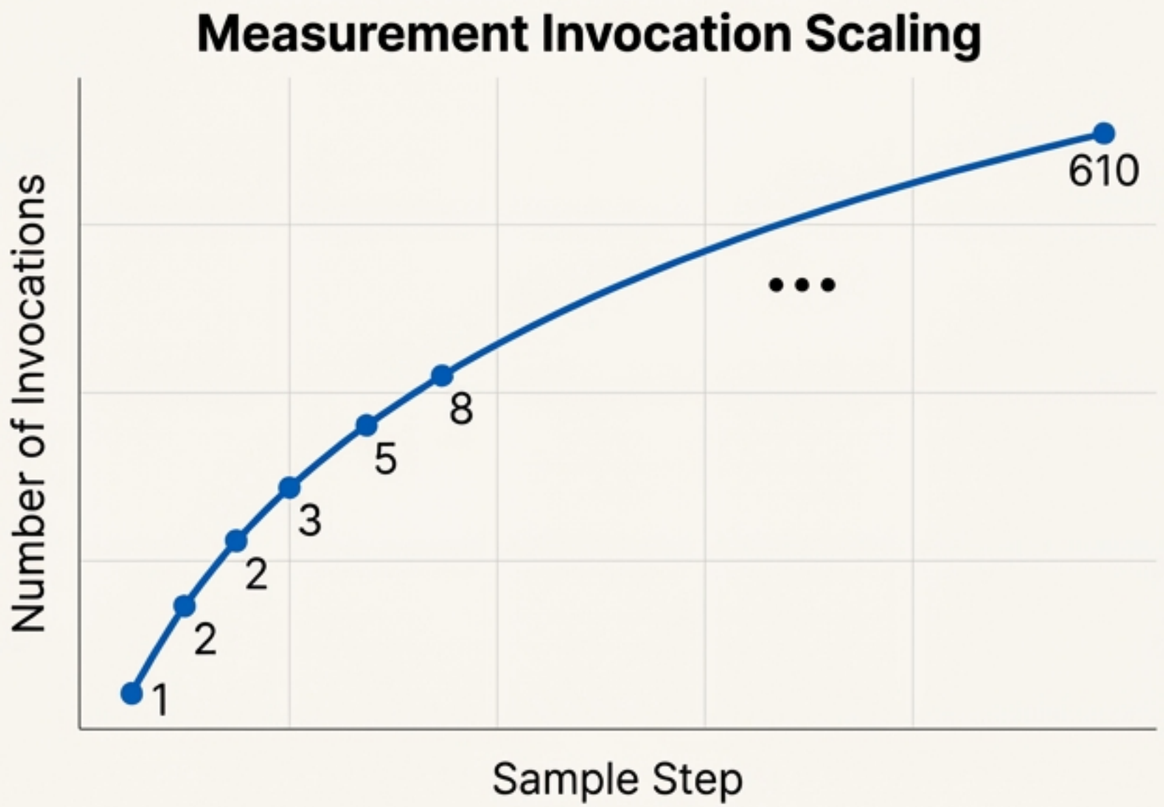
```python
import unittest
from osbot_utils.testing.performance import Perf

class TestMyCode(unittest.TestCase):
    def test_fibonacci_performance(self):
        with Perf() as _:
            _.measure(lambda: fibonacci(10))
            _.assert_time(2000)
```

What's Happening Under the Hood:

This one line triggers 1,595 measurements using Fibonacci scaling, performs outlier removal, normalises the score for stability, and automatically adjusts assertion thresholds when running in a CI environment.

The Core Engine: Fibonacci-Based Sampling.

Instead of uniform sampling, the engine uses a Fibonacci sequence (`[1, 2, 3, 5, 8, 8, 13, …, 610]`) for a total of **1,595 invocations** per measurement.



- **Rapid early sampling**: Catches cold-start and JIT behaviour.
- **Logarithmic scaling**: Effectively captures warm-up and cache effects.
- **Large final samples**: Ensures statistical stability for the final score.

The Core Engine: Raw Score vs. Final Score.

Metric	Description	Use Case
`raw_score`	Weighted median-mean of timings after outlier removal (60% median, 40% mean).	Debugging, detailed performance analysis.
`final_score`	`raw_score` dynamically normalised to a magnitude-appropriate precision.	Assertions, stable cross-run comparisons.

Why Normalisation Matters

Run 1 `raw_score`: **1,847ns** → `final_score`: **2,000ns**

Run 2 `raw_score`: **1,912ns** → `final_score`: **2,000ns**

Normalisation smooths out minor, insignificant variations to prevent flaky assertions.

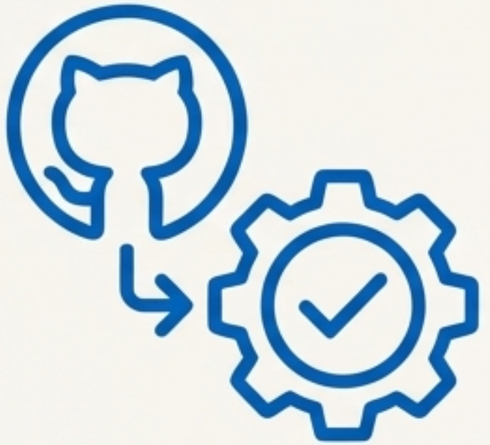
Conquer CI Flakiness Automatically

Code runs slower on shared GitHub Actions runners. Manually adjusting test thresholds is brittle and annoying.

The Solution

The framework automatically detects when it's running in GitHub Actions and adjusts assertion behaviour.

How It Works



- `assert_time(*expected)`: In CI, it asserts the score is `less than or equal to` the last expected time multiplied by 5.
- `assert_time__less_than(max)`: In CI, the `max` threshold is automatically multiplied by 6.

Write your tests once. They work reliably both locally and in your pipeline without any code changes.

Explore Your Data with a Single Command

For deep analysis, move beyond the one-line printout. The `print_report()` method gives you a full statistical breakdown.

```
Performance Report for: fibonacci(10)
...
Histogram (10 bins, width 40):
  1800ns - 1900ns [  2] |
  1900ns - 2000ns [106] |
  2000ns - 2100ns [230] |
  2100ns - 2200ns [ 98] |
  ...
```

The score marker instantly shows which performance bin contains your calculated `final_score`, giving you context on the overall distribution.

Essential Usage Patterns for Your Test Suite.

Reusable Session (Recommended)

Create the session once in `setUpClass` to reuse it across all tests in a class. This is efficient and clean.

```
class TestMySuite(unittest.TestC

    @classmethod
    def setUpClass(cls):
        cls.session = Perf()

    def test_one(self):
        with self.session as _:
            _.measure(...)
```

Chained Methods (Fluent API)

All major methods return `self`, allowing for a clean, readable, and fluent chain of operations.

```
with Perf() as p:
    p.measure(target).print()
    .assert_time(2000)
```

Measuring Class Instantiation

Simply pass the class name to `measure` to benchmark the cost of its `__init__` method.

```
class MyHeavyClass:
    # ... complex __init__ ...

with Perf() as p:
    p.measure(MyHeavyClass)
```

Choose the Right Tool: Full, Fast, or Quick?

Your function's execution speed determines the best measurement strategy.

Method	Invocations	Best For...
`measure(target)`	**1,595**	**The default.** Maximum precision for functions under 100ms.
`measure__fast(target)`	**87**	A balance between speed and precision for medium-length tasks.
`measure__quick(target)`	**19**	Very slow functions (>100ms) where full sampling is impractical.

Pro Tip

For a slow function, running the default `measure()` can add significant time to your test suite. Start with `measure__quick()` and increase precision only if needed.

Best Practices for Robust Tests: The Dos.

DO: Use a class-level session in test suites.

Avoids unnecessary object creation in each test method, making your suite faster.

DO: Define time thresholds as class attributes.

Makes expected values explicit and easy to update. E.g., `TIME_FIB_10_NS = 2000`.

DO: Provide multiple expected values for stability.

Account for minor differences between Python versions or hardware. E.g., `_.assert_time(2000, 2100)`.

DO: Use `assert_time__less_than` for upper bounds.

Perfect for when you care about 'not slower than X' rather than an exact match, which is more resilient to environment changes.

Best Practices for Robust Tests: The Don’ts.

DON’T: Create new sessions inside each test.

This is inefficient and defeats the purpose of a reusable session object.

DON’T: Measure functions with side effects.

Timing functions with I/O, network calls, or randomness will produce inconsistent, meaningless results. Measure pure computational work.

DON’T: Use raw literal values in assertions.

`_.assert_time(2000)` is a 'magic number.’ Using a named constant like `self.TIME_FIB_10_NS` is far more maintainable.

DON’T: Skip output when debugging.

Use `.print()` or `.print_report()` liberally when a test is failing to see the actual measured value.

Common Troubleshooting Scenarios.

Assertion passes locally but fails in CI (or vice-versa).

Cause: Local machine is much faster or slower than CI runner; thresholds are too tight.

Solution: Provide multiple expected values to cover both environments.

```
# Local time is 2000ns, CI
baseline is 8000ns
_.assert_time(2000, 8000)
```

Inconsistent scores between runs.

Cause: The function is not deterministic (e.g., uses I/O or randomness), or the system is under heavy load.

Solution: Ensure you are measuring pure computational functions. Run tests in an isolated environment.

Score is 0ns.

Cause: The function executes too quickly (sub-100ns) for the timer's resolution.

Solution: Wrap the function in a loop to measure a larger, more significant unit of work.

```
# Measure running the function
100 times
_.measure(lambda: [fast_func()
for _ in range(100)])
```

Your Performance Testing Checklist.

Setup & Basics

- [ ] Import `Perf` from `osbot_utils.testing.performance`.
- [ ] Create session once in `setUpClass()`.
- [ ] Use the `with self.session as _:` context manager.
- [ ] Measure pure functions (no I/O, no network).

Assertions & Stability

- [ ] Define thresholds as named constants (e.g., `time_X_ns`).
- [ ] Provide multiple expected values (local, CI) for `assert_time()`.
- [ ] Use `assert_time__less_than()` for flexible upper-bound checks.
- [ ] Remember CI assertions get a 5-6x multiplier automatically.

Debugging & Optimisation

- [ ] Use `.print()` for quick feedback during development.
- [ ] Use `.print_report()` for deep analysis with a histogram.
- [ ] Use `measure__quick()` for slow functions (>100ms).
- [ ] Set `assert_enabled=False` for exploratory analysis without exceptions.