pytest-benchmark

Release 2.5.0

September 13, 2015

Contents:

CHAPTER 1

Overview

1.1 pytest-benchmark

docs	
tests	
package	

A py.test fixture for benchmarking code. It will group the tests into rounds that are calibrated to the chosen timer. See: *calibration*.

- Free software: BSD license

1.1.1 Installation

```
pip install pytest-benchmark
```

1.1.2 Usage

This plugin provides a *benchmark* fixture. This fixture is a callable object that will benchmark any function passed to it.

Example:

```
def something(duration=0.000001):
    # Code to be measured
    return time.sleep(duration)

def test_my_stuff(benchmark):
    # benchmark something
    result = benchmark(something)

    # Extra code, to verify that the run completed correctly.
    # Note: this code is not measured.
    assert result is None
```

You can also pass extra arguments:

```
def test_my_stuff(benchmark):
    # benchmark something
    result = benchmark(something, 0.02)
```

If you need to do some wrapping (like special setup), you can use it as a decorator around a wrapper function:

```
def test_my_stuff(benchmark):
    @benchmark
    def result():
        # Code to be measured
        return something(0.0002)

    # Extra code, to verify that the run completed correctly.
    # Note: this code is not measured.
    assert result is None
```

`py.test` command-line options:

--benchmark-min-time=BENCHMARK_MIN_TIME Minimum time per round. Default: 25.00us

--benchmark-max-time=BENCHMARK_MAX_TIME Maximum time to spend in a benchmark. Default: 1.00s

--benchmark-min-rounds=BENCHMARK_MIN_ROUNDS Minimum rounds, even if total time would exceed *–max-time*. Default: 5

--benchmark-sort=BENCHMARK_SORT Column to sort on. Can be one of: ‘min’, ‘max’, ‘mean’ or ‘stddev’. Default: min

--benchmark-timer=BENCHMARK_TIMER Timer to use when measuring time. Default: time.perf_counter

--benchmark-warmup Runs the benchmarks two times. Discards data from the first run.

--benchmark-warmup-iterations=BENCHMARK_WARMUP_ITERATIONS Max number of iterations to run in the warmup phase. Default: 100000

--benchmark-verbose Dump diagnostic and progress information.

--benchmark-disable-gc Disable GC during benchmarks.

--benchmark-skip Skip running any benchmarks.

--benchmark-only Only run benchmarks.

Setting per-test options:

```
@pytest.mark.benchmark(
    group="group-name",
    min_time=0.1,
    max_time=0.5,
    min_rounds=5,
    timer=time.time,
    disable_gc=True,
    warmup=False
)
def test_my_stuff(benchmark):
    @benchmark
    def result():
        # Code to be measured
        return time.sleep(0.000001)
    # Extra code, to verify that the run
    # completed correctly.
    # Note: this code is not measured.
    assert result is None
```

1.1.3 Glossary

Iteration A single run of your benchmarked function.

Round A set of iterations. The size of a *round* is computed in the calibration phase.

Stats are computed with rounds, not with iterations. The duration for a round is an average of all the iterations in that round.

See: *calibration* for an explanation of why it's like this.

1.1.4 Features

Calibration

`pytest-benchmark` will run your function multiple times between measurements. A *round'is that set of runs done between measurements. This is quite similar to the builtin ''timeit'* module but it's more robust.

The problem with measuring single runs appears when you have very fast code. To illustrate:

In other words, a *round* is a set of runs that are averaged together, those resulting numbers are then used to compute the result tables. The default settings will try to keep the round small enough (so that you get to see variance), but not too small, because then you have the timer calibration issues illustrated above (your test function is faster than or as fast as the resolution of the timer).

Patch utilities

Suppose you want to benchmark an `internal` function from a class:

```
class Foo(object):
    def __init__(self, arg=0.01):
        self.arg = arg

    def run(self):
        self.internal(self.arg)

    def internal(self, duration):
        time.sleep(duration)
```

With the `benchmark` fixture this is quite hard to test if you don't control the `Foo` code or it has very complicated construction.

For this there's an experimental `benchmark_weave` fixture that can patch stuff using aspectlib (make sure you *pip install apectlib* or *pip install pytest-benchmark[aspect]*):

```
def test_foo(benchmark_weave):
    with benchmark_weave(Foo.internal, lazy=True):
        f = Foo()
        f.run()
```

1.1.5 Documentation

https://pytest-benchmark.readthedocs.org/

1.1.6 Obligatory screenshot

1.1.7 Development

To run the all tests run:

```
tox
```

1.1.8 Credits

- Timing code and ideas taken from: https://bitbucket.org/haypo/misc/src/tip/python/benchmark.py

CHAPTER 2

Installation

At the command line:

```
pip install pytest-benchmark
```

CHAPTER 3

Usage

To use pytest-benchmark in a project:

```
import pytest_benchmark
```

CHAPTER 4

Reference

4.1 pytest_benchmark

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

5.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.2 Documentation improvements

pytest-benchmark could always use more documentation, whether as part of the official pytest-benchmark docs, in docstrings, or even on the web in blog posts, articles, and such.

5.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/ionelmc/pytest-benchmark/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.4 Development

To set up *pytest-benchmark* for local development:

1. Fork pytest-benchmark on GitHub.
2. Clone your fork locally:

```
    git clone git@github.com:your_name_here/pytest-benchmark.git
```

3. Create a branch for local development:

```
    git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

```
    tox
```

5. Commit your changes and push your branch to GitHub:

```
    git add .
    git commit -m "Your detailed description of your changes."
    git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

5.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`) [1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

5.4.2 Tips

To run a subset of tests:

```
tox -e envname -- py.test -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.

It will be slower though ...

CHAPTER 6

Authors

- Ionel Cristian Mărie - http://blog.ionelmc.ro
- Marc Abramowitz - http://marc-abramowitz.com

CHAPTER 7

Changelog

7.1 2.5.0 (2015-06-20)

- Improved test suite a bit (not using *cram* anymore).
- Improved help text on the `--benchmark-warmup` option.
- Made `warmup_iterations` available as a marker argument (eg: `@pytest.mark.benchmark(warmup_iterations=1234)`).
- Fixed `--benchmark-verbose`'s printouts to work properly with output capturing.
- Changed how warmup iterations are computed (now number of total iterations is used, instead of just the rounds).
- Fixed a bug where calibration would run forever.
- Disabled red/green coloring (it was kinda random) when there's a single test in the results table.

7.2 2.4.1 (2015-03-16)

- Fix regression, plugin was raising `ValueError: no option named 'dist'` when xdist wasn't installed.

7.3 2.4.0 (2015-03-12)

- Add a `benchmark_weave` experimental fixture.
- Fix internal failures when *xdist* plugin is active.
- Automatically disable benchmarks if *xdist* is active.

7.4 2.3.0 (2014-12-27)

- Moved the warmup in the calibration phase. Solves issues with benchmarking on PyPy.

  Added a `--benchmark-warmup-iterations` option to fine-tune that.

7.5 2.2.0 (2014-12-26)

- Make the default rounds smaller (so that variance is more accurate).
- Show the defaults in the `--help` section.

7.6 2.1.0 (2014-12-20)

- Simplify the calibration code so that the round is smaller.
- Add diagnostic output for calibration code (`--benchmark-verbose`).

7.7 2.0.0 (2014-12-19)

- Replace the context-manager based API with a simple callback interface.
- Implement timer calibration for precise measurements.

7.8 1.0.0 (2014-12-15)

- Use a precise default timer for PyPy.

7.9 ? (?)

- Readme and styling fixes (contributed by Marc Abramowitz)
- Lots of wild changes.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index