pytest-benchmark

Release 3.0.0

November 08, 2015

Contents

Contents:

CHAPTER 1

Installation

At the command line:

```
pip install pytest-benchmark
```

CHAPTER 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):
    """
    Function that needs some serious benchmarking.
    """
    time.sleep(duration)
    # You may return anything you want, like the result of a computation
    return 123

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

    # Extra code, to verify that the run completed correctly.
    # Sometimes you may want to check the result, fast functions
    # are no good if they return incorrect results :-)
    assert result == 123
```

You can also pass extra arguments:

```
def test_my_stuff(benchmark):
    benchmark(time.sleep, 0.02)
```

Or even keyword arguments:

```
def test_my_stuff(benchmark):
    benchmark(time.sleep, duration=0.02)
```

Another pattern seen in the wild, that is not recommended for micro-benchmarks (very fast code) but may be convenient:

```
def test_my_stuff(benchmark):
    @benchmark
    def something():  # unnecessary function call
        time.sleep(0.000001)
```

A better way is to just benchmark the final function:

```
def test_my_stuff(benchmark):
    benchmark(time.sleep, 0.000001)  # way more accurate results!
```

If you need to do fine control over how the benchmark is run (like a *setup* function, exact control of *iterations* and *rounds*) there's a special mode - pedantic:

```
def my_special_setup():
    ...

def test_with_setup(benchmark):
    benchmark.pedantic(something, setup=my_special_setup, args=(1, 2, 3), kwargs={'foo': 'bar'}, ite
```

2.1 Commandline options

`py.test` command-line options:

--benchmark-min-time=SECONDS Minimum time per round in seconds. Default: '0.000005'

--benchmark-max-time=SECONDS Maximum run time per test - it will be repeated until this total time is reached. It may be exceeded if test function is very slow or –benchmark-min-rounds is large (it takes precedence). Default: '1.0'

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

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

--benchmark-group-by=LABEL How to group tests. Can be one of: 'group', 'name', 'fullname', 'func', 'fullfunc' or 'param'. Default: 'group'

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

--benchmark-calibration-precision=NUM Precision to use when calibrating number of iterations. Precision of 10 will make the timer look 10 times more accurate, at a cost of less precise measure of deviations. Default: 10

--benchmark-warmup=KIND Activates warmup. Will run the test function up to number of times in the calibration phase. See *–benchmark-warmup-iterations*. Note: Even the warmup phase obeys –benchmark-max-time. Available KIND: 'auto', 'off', 'on'. Default: 'auto' (automatically activate on PyPy).

--benchmark-warmup-iterations=NUM 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.

--benchmark-save=NAME Save the current run into 'STORAGE-PATH/counter- NAME.json'. Default: '<commitid>_<date>_<time>_<isdirty>', example: 'e689af57e7439b9005749d806248897ad550eab5_20150811_041632_uncommitted-changes'.

--benchmark-autosave Autosave the current run into 'STORAGE-PATH/<counter>_<commitid>_<date>_<time>_<isdirty>', example: 'STORAGE-PATH/0123_525685bcd6a51d1ade0be75e2892e713e02dfd19_20151028_221708_changes.json'

--benchmark-save-data Use this to make –benchmark-save and –benchmark- autosave include all the timing data, not just the stats.

--benchmark-compare=NUM Compare the current run against run NUM or the latest saved run if unspecified.

--benchmark-compare-fail=EXPR Fail test if performance regresses according to given EXPR (eg: min:5% or mean:0.001 for number of seconds). Can be used multiple times.

--benchmark-storage=STORAGE-PATH Specify a different path to store the runs (when –benchmark-save or –benchmark-autosave are used). Default: './.benchmarks/<os>-<pyimplementation>-<pyversion>-<arch>bit', example: 'Linux-CPython-2.7-64bit'.

--benchmark-histogram=FILENAME-PREFIX Plot graphs of min/max/avg/stddev over time in FILENAME-PREFIX-test_name.svg. If FILENAME-PREFIX contains slashes ('/') then directories will be created. Default: 'benchmark_<date>_<time>', example: 'benchmark_20150811_041632'.

--benchmark-json=PATH Dump a JSON report into PATH. Note that this will include the complete data (all the timings, not just the stats).

2.2 Markers

You can set per-test options with the `benchmark` marker:

```
@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
```

2.3 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 aspectlib` or `pip install pytest-benchmark[aspect]`):

```
def test_foo(benchmark):
    benchmark.weave(Foo.internal, lazy=True):
    f = Foo()
    f.run()
```

CHAPTER 3

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).

By default `pytest-benchmark` will try to run your function as many times needed to fit a *10 x TIMER_RESOLUTION* period. You can fine tune this with the `--benchmark-min-time` and `--benchmark-calibration-precision` options.

CHAPTER 4

Pedantic mode

pytest-benchmark allows a special mode that doesn't do any automatic calibration. To make it clear it's only for people that know exactly what they need it's called "pedantic".

```
def test_with_setup(benchmark):
    benchmark.pedantic(stuff, args=(1, 2, 3), kwargs={'foo': 'bar'}, iterations=10, rounds=100)
```

4.1 Reference

benchmark.**pedantic**(*target*, *args=()*, *kwargs=None*, *setup=None*, *rounds=1*, *warmup_rounds=0*, *iterations=1*)

Parameters

- **target** (*callable*) – Function to benchmark.
- **args** (*list or tuple*) – Positional arguments to the target function.
- **kwargs** (*dict*) – Named arguments to the target function.
- **setup** (*callable*) – A function to call right before calling the target function.

  The setup function can also return the arguments for the function (in case you need to create new arguments every time).

```
def test_with_setup(benchmark):
    def setup():
        # can optionally return a (args, kwargs) tuple
        return (1, 2, 3), {'foo': 'bar'}
    benchmark.pedantic(stuff, setup=setup, rounds=100)
```

Note: if you use a setup function then you cannot use the args, kwargs and iterations options.

- **rounds** (*int*) – Number of rounds to run.
- **iterations** (*int*) – Number of iterations.

  In the non-pedantic mode (eg: benchmark(stuff, 1, 2, 3, foo='bar')) the iterations is automatically chosen depending on what timer you have. In other words, be careful in what you chose for this option.

The default value (`1`) is **unsafe** for benchmarking very fast functions that take under 100μs (100 microseconds).

- **`warmup_rounds`** (*int*) – Number of warmup rounds.

  Set to non-zero to enable warmup. Warmup will run with the same number of iterations.

  Example: if you have `iteration=5, warmup_rounds=10` then your function will be called 50 times.

CHAPTER 5

Comparing past runs

Before comparing different runs it's ideal to make your tests as consistent as possible, see Frequently Asked Questions for more details.

pytest-benchmark has support for storing stats and data for the previous runs.

To store a run just add `--benchmark-autosave` or `--benchmark-save=some-name` to your pytest arguments. All the files are saved in a path like `.benchmarks/Linux-CPython-3.4-64bit`.

- `--benchmark-autosave` saves a file like `0001_c9cca5de6a4c7eb2_20150815_215724.json` where:
  - `0001` is an automatically incremented id, much like how django migrations have a number.
  - `c9cca5de6a4c7eb2` is the commit id (if you use Git or Mercurial)
  - `20150815_215724` is the current time

  You should add `--benchmark-autosave` to `addopts` in you pytest configuration so you dont have to specify it all the time.
- `--benchmark-name=foobar` works similarly, but saves a file like `0001_foobar.json`. It's there in case you want to give specific name to the run.

After you have saved your first run you can compare against it with `--benchmark-compare=0001`. You will get an additional row for each test in the result table, showing the differences.

You can also make the suite fail with `--benchmark-compare-fail=<stat>:<num>%` or `--benchmark-compare-fail=<stat>:<num>`. Examples:

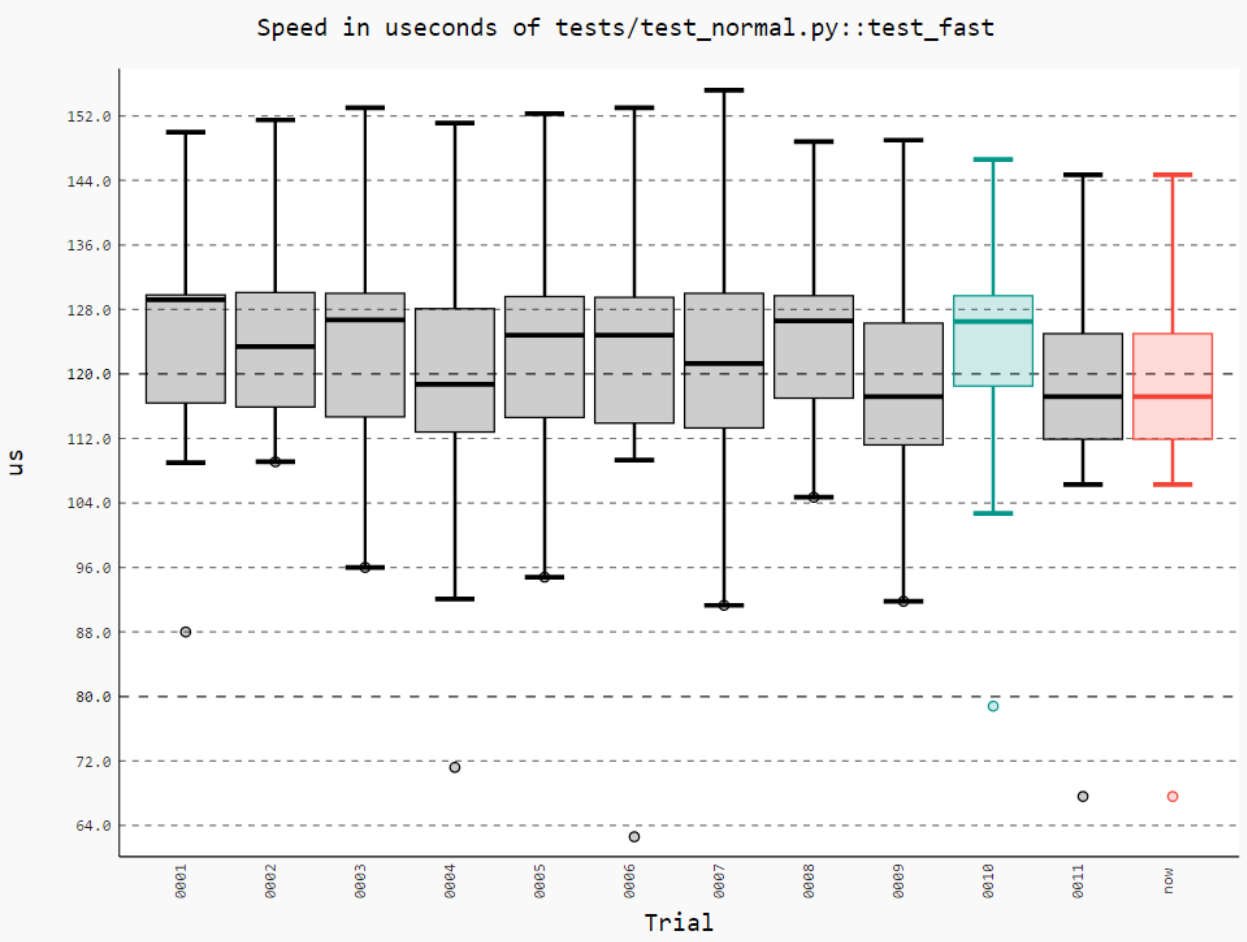
- `--benchmark-compare-fail=min:5%` will make the suite fail if `Min` is 5% slower for any test.
- `--benchmark-compare-fail=mean:0.001` will make the suite fail if `Mean` is 0.001 seconds slower for any test.

5.1 Plotting

Note: To use plotting you need to `pip install pygal pygaljs` or `pip install pytest-benchmark[histogram]`.

You can also get a nice plot with `--benchmark-histogram`. The result is a modified Tukey box and wiskers plot where the outliers (the small bullets) are `Min` and `Max`.

Example output:


Speed in useconds of tests/test_normal.py::test_fast
us
152.0
144.0
136.0
128.0
120.0
112.0
104.0
96.0
88.0
80.0
72.0
64.0
0001
0002
0003
0004
0005
0006
0007
0008
0009
0010
0011
now
Trial

CHAPTER 6

Hooks

Hooks for customizing various parts of pytest-benchmark.

pytest_benchmark.hookspec.**pytest_benchmark_compare_machine_info**(*config*, *benchmarksession*, *machine_info*, *compared_benchmark*)

You may want to use this hook to implement custom checks or abort execution. pytest-benchmark builtin hook does this:

```
def pytest_benchmark_compare_machine_info(config, benchmarksession, machine_info, compared_bench
    if compared_benchmark["machine_info"] != machine_info:
        benchmarksession.logger.warn(
            "Benchmark machine_info is different. Current: %s VS saved: %s." % (
                format_dict(machine_info),
                format_dict(compared_benchmark["machine_info"]),
            )
        )
```

pytest_benchmark.hookspec.**pytest_benchmark_generate_commit_info**(*config*)

To completely replace the generated commit_info do something like this:

```
def pytest_benchmark_generate_commit_info(config):
    return {'id': subprocess.check_output(['svnversion']).strip()}
```

pytest_benchmark.hookspec.**pytest_benchmark_generate_json**(*config*, *benchmarks*, *include_data*)

You should read pytest-benchmark's code if you really need to wholly customize the json.

> **Warning:** Improperly customizing this may cause breakage if --benchmark-compare or --benchmark-histogram are used.

By default, pytest_benchmark_generate_json strips benchmarks that have errors from the output. To prevent this, implement the hook like this:

```
@pytest.mark.hookwrapper
def pytest_benchmark_generate_json(config, benchmarks, include_data):
    for bench in benchmarks:
        bench.has_error = False
    yield
```

pytest_benchmark.hookspec.**pytest_benchmark_generate_machine_info**(*config*)

To completely replace the generated machine_info do something like this:

```
def pytest_benchmark_update_machine_info(config):
    return {'user': getpass.getuser()}
```

pytest_benchmark.hookspec.**pytest_benchmark_group_stats**(*config*, *benchmarks*, *group_by*)

You may perform grouping customization here, in case the builtin grouping doesn't suit you.

Example:

```
@pytest.mark.hookwrapper
def pytest_benchmark_group_stats(config, benchmarks, group_by):
    outcome = yield
    if group_by == "special":  # when you use --benchmark-group-by=special
        result = defaultdict(list)
        for bench in benchmarks:
            # `bench.special` doesn't exist, replace with whatever you need
            result[bench.special].append(bench)
        outcome.force_result(result.items())
```

pytest_benchmark.hookspec.**pytest_benchmark_update_commit_info**(*config*, *info*)

To add something into the commit_info, like the commit message do something like this:

```
def pytest_benchmark_update_commit_info(config, info):
    info['message'] = subprocess.check_output(['git', 'log', '-1', '--pretty=%B']).strip()
```

pytest_benchmark.hookspec.**pytest_benchmark_update_json**(*config*, *benchmarks*, *output_json*)

Use this to add custom fields in the output JSON.

Example:

```
def pytest_benchmark_update_json(config, benchmarks, output_json):
    output_json['foo'] = 'bar'
```

pytest_benchmark.hookspec.**pytest_benchmark_update_machine_info**(*config*, *info*)

If benchmarks are compared and machine_info is different then warnings will be shown.

To add the current user to the commit info override the hook in your conftest.py like this:

```
def pytest_benchmark_update_machine_info(config, info):
    info['user'] = getpass.getuser()
```

CHAPTER 7

Frequently Asked Questions

Why is my `StdDev` so high? There can be few causes for this:

- Bad isolation. You run other services in your machine that eat up your cpu or you run in a VM and that makes machine performance inconsistent. Ideally you'd avoid such setups, stop all services and applications and use bare metal machines.
- Bad tests or too much complexity. The function you're testing is doing I/O, using external resources, has side-effects or doing other non-deterministic things. Ideally you'd avoid testing huge chunks of code.

  One special situation is PyPy: it's GC and JIT can add unpredictable overhead - you'll see it as huge spikes all over the place. You should make sure that you have a good amount of warmup (using `--benchmark-warmup` and `--benchmark-warmup-iterations`) to prime the JIT as much as possible. Unfortunately not much can be done about GC overhead.

  If you cannot make your tests more predictable and remove overhead you should look at different stats like: IQR and Median. IQR is often better than StdDev.

My is my `Min` way lower than `Q1-1.5IQR`? You may see this issue in the histogram plot. This is another instance of *bad isolation*.

For example, Intel CPUs has a feature called Turbo Boost wich overclocks your CPU depending how many cores you have at that time and hot your CPU is. If your CPU is too hot you get no Turbo Boost. If you get Turbo Boost active then the CPU quickly gets hot. You can see how this won't work for sustained workloads.

When Turbo Boost kicks in you may see "speed spikes" - and you'd get this strange outlier `Min`.

When you have other programs running on your machine you may also see the "speed spikes" - the other programs idle for a brief moment and that allows your function to run way faster in that brief moment.

I can't avoid using VMs or running other programs. What can I do? As a last ditch effort pytest-benchmark allows you to plugin in custom timers (`--benchmark-timer`). You could use something like `time.process_time` (Python 3.3+ only) as the timer. Process time doesn't include sleeping or waiting for I/O.

The histogram doesn't show `Max` time. What gives?! The height of the plot is limited to `Q3+1.5IQR` because `Max` has the nasty tendency to be way higher and making everything else small and undiscerning. For this reason `Max` is *plotted outside*.

Most people don't care about `Max` at all so this is fine.

CHAPTER 8

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.

Mean TODO

Median TODO

IQR InterQuertile Range. This is a different way to measure variance. Good explanation here

StdDev TODO: Standard Deviation

Outliers TODO

CHAPTER 9

Contributing

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

9.1 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.

9.2 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.

9.3 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 :)

9.4 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.

9.4.1 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`.

9.4.2 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 10

Authors

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

CHAPTER 11

Changelog

11.1 3.0.0 (2015-08-11)

- Improved `--help` text for `--benchmark-histogram`, `--benchmark-save` and `--benchmark-autosave`.
- Benchmarks that raised exceptions during test now have special highlighting in result table (red background).
- Benchmarks that raised exceptions are not included in the saved data anymore (you can still get the old behavior back by implementing `pytest_benchmark_generate_json` in your `conftest.py`).
- The plugin will use pytest's warning system for warnings. There are 2 categories: `WBENCHMARK-C` (compare mode issues) and `WBENCHMARK-U` (usage issues).
- The red warnings are only shown if `--benchmark-verbose` is used. They still will be always be shown in the pytest-warnings section.
- Using the benchmark fixture more than one time is disallowed (will raise exception).
- Not using the benchmark fixutre (but requiring it) will issue a warning (`WBENCHMARK-U1`).

11.2 3.0.0rc1 (2015-10-25)

- Changed `--benchmark-warmup` to take optional value and automatically activate on PyPy (default value is `auto`). *MAY BE BACKWARDS INCOMPATIBLE*
- Removed the version check in compare mode (previously there was a warning if current version is lower than what's in the file).

11.3 3.0.0b3 (2015-10-22)

- Changed how comparison is displayed in the result table. Now previous runs are shown as normal runs and names get a special suffix indicating the origin. Eg: "test_foobar (NOW)" or "test_foobar (0123)".
- Fixed sorting in the result table. Now rows are sorted by the sort column, and then by name.
- Show the plugin version in the header section.
- Moved the display of default options in the header section.

11.4 3.0.0b2 (2015-10-17)

- Add a `--benchmark-disable` option. It's automatically activated when xdist is on
- When xdist is on or *statistics* can't be imported then `--benchmark-disable` is automatically activated (instead of `--benchmark-skip`). *BACKWARDS INCOMPATIBLE*
- Replace the deprecated `__multicall__` with the new hookwrapper system.
- Improved description for `--benchmark-max-time`.

11.5 3.0.0b1 (2015-10-13)

- Tests are sorted alphabetically in the results table.
- Failing to import *statistics* doesn't create hard failures anymore. Benchmarks are automatically skipped if import failure occurs. This would happen on Python 3.2 (or earlier Python 3).

11.6 3.0.0a4 (2015-10-08)

- Changed how failures to get commit info are handled: now they are soft failures. Previously it made the whole test suite fail, just because you didn't have `git/hg` installed.

11.7 3.0.0a3 (2015-10-02)

- Added progress indication when computing stats.

11.8 3.0.0a2 (2015-09-30)

- Fixed accidental output capturing caused by capturemanager misuse.

11.9 3.0.0a1 (2015-09-13)

- Added JSON report saving (the `--benchmark-json` command line arguments).
- Added benchmark data storage(the `--benchmark-save` and `--benchmark-autosave` command line arguments).
- Added comparison to previous runs (the `--benchmark-compare` command line argument).
- Added performance regression checks (the `--benchmark-compare-fail` command line argument).
- Added possibility to group by various parts of test name (the *–benchmark-compare-group-by*' command line argument).
- Added historical plotting (the `--benchmark-histogram` command line argument).
- Added option to fine tune the calibration (the `--benchmark-calibration-precision` command line argument and `calibration_precision` marker option).

- Changed `benchmark_weave` to no longer be a context manager. Cleanup is performed automatically. *BACKWARDS INCOMPATIBLE*
- Added `benchmark.weave` method (alternative to `benchmark_weave` fixture).
- Added new hooks to allow customization:
  - `pytest_benchmark_generate_machine_info(config)`
  - `pytest_benchmark_update_machine_info(config, info)`
  - `pytest_benchmark_generate_commit_info(config)`
  - `pytest_benchmark_update_commit_info(config, info)`
  - `pytest_benchmark_group_stats(config, benchmarks, group_by)`
  - `pytest_benchmark_generate_json(config, benchmarks, include_data)`
  - `pytest_benchmark_update_json(config, benchmarks, output_json)`
  - `pytest_benchmark_compare_machine_info(config, benchmarksession, machine_info, compared_benchmark)`
- Changed the timing code to:
  - Tracers are automatically disabled when running the test function (like coverage tracers).
  - Fixed an issue with calibration code getting stuck.
- Added *pedantic mode* via `benchmark.pedantic()`. This mode disables calibration and allows a setup function.

11.10 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.

11.11 2.4.1 (2015-03-16)

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

11.12 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.

11.13 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.

11.14 2.2.0 (2014-12-26)

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

11.15 2.1.0 (2014-12-20)

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

11.16 2.0.0 (2014-12-19)

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

11.17 1.0.0 (2014-12-15)

- Use a precise default timer for PyPy.

11.18 ? (?)

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

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

p

B

P