
pytest-benchmark

Release 5.2.3

Ionel Cristian Mărieș

Jun 16, 2026

CONTENTS

1	Screenshots	3
2	User guide	5
2.1	Installation	5
2.2	Usage	5
2.3	Calibration	10
2.4	Pedantic mode	11
2.5	Comparing past runs	13
2.6	Hooks	15
2.7	Frequently Asked Questions	17
2.8	Glossary	18
2.9	Contributing	18
2.10	Authors	20
2.11	Changelog	21
3	Indices and tables	29
	Python Module Index	31
	Index	33

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

Notable features and goals:

- Sensible defaults and automatic calibration for micro-benchmarks
- Good integration with pytest
- Comparison and regression tracking
- Exhaustive statistics
- JSON export

Examples:

```
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

def test_my_stuff_different_arg(benchmark):
    # benchmark something, but add some arguments
    result = benchmark(something, 0.001)
    assert result == 123
```


SCREENSHOTS

Normal run:

```

===== test session starts =====
platform linux -- Python 3.4.3, pytest-2.8.3.dev1, py-1.4.30, pluggy-0.3.1
benchmark: 3.0.0b3 (defaults: timer=time.perf_counter disable_gc=False min_rounds=5 min_time=5.00us max_time=1.00s calibration_precision=10 warmup=False warmup_iterations=100000)
rootdir: /home/ionel/osp/pytest-benchmark, infile: setup.cfg
plugins: instaFail-0.3.0, xdist-1.13.1, benchmark-3.0.0b3
collected 9 items

tests/test_normal.py .....

----- benchmark: 8 tests -----
Name (time in us)      Min          Max          Mean          StdDev          Median          IQR          Outliers(*)  Rounds  Iterations
-----
test_parametrized[1]  70.5000 (1.0)  6,004.5000 (2.36)  131.2122 (1.01)  102.9481 (5.60)  125.3000 (1.00)  14.6000 (1.07)  121;409  7525  1
test_parametrized[3]  75.2000 (1.07)  2,539.1000 (1.0)  131.9495 (1.01)  81.4228 (4.43)  128.1500 (1.11)  14.9000 (1.10)  129;397  6850  1
test_fast             77.0000 (1.09)  6,836.2000 (2.69)  130.1859 (1.0)  165.6270 (9.01)  115.6000 (1.0)  13.6000 (1.0)  39;172  3771  1
test_parametrized[0]  83.0000 (1.19)  2,681.6000 (1.06)  130.5126 (1.00)  87.1091 (4.74)  125.1000 (1.00)  16.4000 (1.21)  140;346  7881  1
test_parametrized[2]  87.3000 (1.30)  3,578.1000 (1.41)  133.4347 (1.02)  105.3837 (5.74)  125.5000 (1.00)  14.2000 (1.04)  143;430  7463  1
test_parametrized[4]  108.7000 (1.54)  8,282.2000 (3.26)  130.4617 (1.00)  138.8991 (7.56)  118.6000 (1.03)  17.7001 (1.30)  110;321  7559  1
test_slow             1,031.9000 (14.64)  3,704.6000 (1.46)  1,094.0810 (8.40)  115.1258 (6.27)  1,082.5000 (9.36)  22.4999 (1.65)  18;59  938  1
test_slower          10,071.6000 (142.86)  10,166.0000 (4.00)  10,097.2040 (77.56)  18.3750 (1.0)  10,098.6000 (87.36)  26.1000 (1.92)  28;3  100  1
-----

(*) Outliers: 1 Standard Deviation from Mean; 1.5 IQR (InterQuartile Range) from 1st Quartile and 3rd Quartile.
===== 9 passed in 8.98 seconds =====
summary
3.4-pytest28-xdist-nocov: commands succeeded
congratulations :)
```

Compare mode (--benchmark-compare):

```

===== test session starts =====
platform linux -- Python 3.4.3, pytest-2.8.3.dev1, py-1.4.30, pluggy-0.3.1
benchmark: 3.0.0b3 (defaults: timer=time.perf_counter disable_gc=False min_rounds=5 min_time=5.00us max_time=1.00s calibration_precision=10 warmup=False warmup_iterations=100000)
rootdir: /home/ione1/osp/pytest-benchmark, inifile: setup.cfg
plugins: instafail-0.3.0, xdist-1.13.1, benchmark-3.0.0b3
collected 9 items

tests/test_normal.py .....

Saved benchmark data in /home/ione1/osp/pytest-benchmark/.benchmarks/Linux-CPython-3.4-64bit/0010_b296fc4a65dbc1c2824c29bbd2ae8bf0c1079f5d_20151023_230227_uncommitted-changes.json
Comparing against benchmark 0009_b296fc4a65dbc1c2824c29bbd2ae8bf0c1079f5d_20151023_230201_uncommitted-changes.json:
| commit info: {dirty: true, id: "b296fc4a65dbc1c2824c29bbd2ae8bf0c1079f5d"}
| saved at: 2015-10-23T20:02:10.846816
| saved using pytest-benchmark 3.0.0b3:

----- benchmark 'test_fast': 2 tests -----
Name (time in us)      Min          Max          Mean          StdDev         Median          IQR          Outliers(*)  Rounds  Iterations
-----
test_fast (NOW)       78.8000 (1.0)  3,242.1000 (1.08)  133.5086 (1.04)  93.0785 (1.14)  126.5000 (1.08)  11.2000 (1.0)  68;299      4406     1
test_fast (0009)     91.8000 (1.16)  2,998.3000 (1.0)  128.4669 (1.0)  81.7567 (1.0)  117.2000 (1.0)  15.1000 (1.35)  168;412     7332     1

----- benchmark 'test_parametrized[0]': 2 tests -----
Name (time in us)      Min          Max          Mean          StdDev         Median          IQR          Outliers(*)  Rounds  Iterations
-----
test_parametrized[0] (NOW)  41.6000 (1.0)  4,734.1000 (1.0)  132.0823 (1.01)  119.1848 (1.0)  125.4000 (1.06)  12.7000 (1.0)  60;337     5862     1
test_parametrized[0] (0009) 72.2000 (1.74)  6,738.7000 (1.42)  131.1306 (1.0)  157.1079 (1.32)  118.3000 (1.0)  17.1000 (1.35)  118;317     8292     1

----- benchmark 'test_parametrized[1]': 2 tests -----
Name (time in us)      Min          Max          Mean          StdDev         Median          IQR          Outliers(*)  Rounds  Iterations
-----
test_parametrized[1] (0009) 41.0000 (1.0)  7,478.2000 (1.22)  135.1095 (1.05)  170.6950 (1.82)  116.4000 (1.0)  14.6001 (1.0)  111;602     7210     1
test_parametrized[1] (NOW) 62.6000 (1.53)  6,109.5000 (1.0)  128.0900 (1.0)  93.9518 (1.0)  124.2000 (1.07)  14.8001 (1.01)  80;343     8592     1

----- benchmark 'test_parametrized[2]': 2 tests -----
Name (time in us)      Min          Max          Mean          StdDev         Median          IQR          Outliers(*)  Rounds  Iterations
-----
test_parametrized[2] (0009) 88.5000 (1.0)  2,663.3000 (1.0)  132.2282 (1.02)  90.8058 (1.21)  119.5000 (1.0)  16.8000 (1.23)  186;647     8526     1
test_parametrized[2] (NOW) 94.1000 (1.06)  3,236.4000 (1.22)  129.8800 (1.0)  75.1645 (1.0)  125.5000 (1.05)  13.7000 (1.0)  128;383     8584     1

----- benchmark 'test_parametrized[3]': 2 tests -----
Name (time in us)      Min          Max          Mean          StdDev         Median          IQR          Outliers(*)  Rounds  Iterations
-----
test_parametrized[3] (0009) 73.8000 (1.0)  5,033.5000 (3.34)  134.3037 (1.06)  102.5793 (2.19)  119.6000 (1.0)  17.0000 (1.34)  174;598     7893     1
test_parametrized[3] (NOW) 95.0000 (1.29)  1,507.6000 (1.0)  127.2507 (1.0)  46.8490 (1.0)  125.3000 (1.05)  12.7000 (1.0)  137;329     8584     1

----- benchmark 'test_parametrized[4]': 2 tests -----
Name (time in us)      Min          Max          Mean          StdDev         Median          IQR          Outliers(*)  Rounds  Iterations
-----
test_parametrized[4] (0009) 76.3000 (1.0)  3,739.6000 (2.05)  132.0518 (1.02)  101.2133 (1.61)  119.3000 (1.0)  16.3000 (1.22)  181;577     8788     1
test_parametrized[4] (NOW) 103.6000 (1.36)  1,825.3000 (1.0)  129.1454 (1.0)  62.9725 (1.0)  125.4000 (1.05)  13.4000 (1.0)  142;351     8272     1

----- benchmark 'test_slow': 2 tests -----
Name (time in ms)      Min          Max          Mean          StdDev         Median          IQR          Outliers(*)  Rounds  Iterations
-----
test_slow (0009)       1.0242 (1.0)  5.3503 (3.70)  1.1180 (1.03)  0.2396 (8.04)  1.0777 (1.0)  0.0296 (1.36)  23;93      841     1
test_slow (NOW)       1.0610 (1.04)  1.4473 (1.0)  1.0905 (1.0)  0.0298 (1.0)  1.0864 (1.01)  0.0218 (1.0)  80;69      837     1

----- benchmark 'test_slower': 2 tests -----
Name (time in ms)      Min          Max          Mean          StdDev         Median          IQR          Outliers(*)  Rounds  Iterations
-----
test_slower (NOW)     10.0716 (1.0)  10.1565 (1.0)  10.0992 (1.0)  0.0156 (1.0)  10.0999 (1.00)  0.0240 (1.0)  32;1       99     1
test_slower (0009)   10.0747 (1.00)  16.9446 (1.67)  10.2042 (1.01)  0.7216 (46.38)  10.0942 (1.0)  0.0284 (1.18)  3;10      100     1

(*) Outliers: 1 Standard Deviation from Mean; 1.5 IQR (InterQuartile Range) from 1st Quartile and 3rd Quartile.
===== 9 passed in 9.26 seconds =====
summary
3.4-pytest28-xdist-nocov: commands succeeded
congratulations :)

```

Histogram (--benchmark-histogram):

Also, it has nice tooltips.

2.1 Installation

At the command line:

```
pip install pytest-benchmark
```

2.2 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'}, iterations=10, rounds=100)
```

2.2.1 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-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** Argument: [KIND] (optional) 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-disable-gc** Disable GC during benchmarks.
- benchmark-skip** Skip running any tests that contain benchmarks.
- benchmark-disable** Disable benchmarks. Benchmarked functions are only ran once and no stats are reported. Use this is you want to run the test but don't do any benchmarking.
- benchmark-enable** Forcibly enable benchmarks. Use this option to override `--benchmark-disable` (in case you have it in `pytest` configuration).

- benchmark-only** Only run benchmarks. This overrides `--benchmark-skip`.
- benchmark-save NAME** Save the current run into `'STORAGE-PATH/counter_NAME.json'`.
- benchmark-autosave** Autosave the current run into `'STORAGE-PATH/counter_...json'`
- benchmark-save-data** Use this to make `--benchmark-save` and `--benchmark-autosave` include all the timing data, not just the stats.
- benchmark-json PATH** Dump a JSON report into PATH. Note that this will include the complete data (all the timings, not just the stats).
- benchmark-compare** Argument: NUM | ID
Compare the current run against run NUM (or prefix of `_id` in elasticsearch) 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-cprofile COLUMN** If specified cProfile will be enabled. Top functions will be stored for the given column. Available columns: `'ncalls_recursion'`, `'ncalls'`, `'tottime'`, `'tottime_per'`, `'cumtime'`, `'cumtime_per'`, `'function_name'`.
- benchmark-cprofile-loops LOOPS** How many times to run the function in cprofile. Available options: `'auto'`, or an integer.
- benchmark-cprofile-top COUNT** How many rows to display.
- benchmark-cprofile-dump** Argument: [FILENAME-PREFIX] (optional)
Save cprofile dumps as `FILENAME-PREFIX-test_name.prof`. If `FILENAME-PREFIX` contains slashes (`'/'`) then directories will be created. Default: `'benchmark_20251107_124338'`
- benchmark-storage URI** Specify a path to store the runs as uri in form `file://path` or `elasticsearch+http[s]://host1,host2/[index/doctype?project_name=Project]` (when `--benchmark-save` or `--benchmark-autosave` are used). For backwards compatibility unexpected values are converted to `file://<value>`. Default: `'file://./benchmarks'`.
- benchmark-netrc** Argument: [BENCHMARK_NETRC] (optional)
Load elasticsearch credentials from a netrc file. Default: `''`.
- benchmark-verbose** Dump diagnostic and progress information.
- benchmark-quiet** Disable reporting. Verbose mode takes precedence.
- benchmark-sort COL** Column to sort on. Can be one of: `'min'`, `'max'`, `'mean'`, `'stddev'`, `'name'`, `'fullname'`. Default: `'min'`
- benchmark-group-by LABEL** How to group tests. Can be one of: `'group'`, `'name'`, `'fullname'`, `'func'`, `'fullfunc'`, `'param'` or `'param:NAME'`, where NAME is the name passed to `@pytest.parametrize`. Default: `'group'`
- benchmark-columns LABELS** Comma-separated list of columns to show in the result table. Default: `'min, max, mean, stddev, median, iqr, outliers, ops, rounds, iterations'`
- benchmark-name FORMAT** How to format names in results. Can be one of `'short'`, `'normal'`, `'long'`, or `'trial'`. Default: `'normal'`

--benchmark-time-unit COLUMN Unit to scale the results to. Available units: ‘ns’, ‘us’, ‘ms’, ‘s’. Default: ‘auto’.

--benchmark-histogram Argument: [FILENAME-PREFIX] (optional)

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_20251107_124338’

Comparison CLI

An extra `py.test-benchmark` bin is available for inspecting previous benchmark data:

```
py.test-benchmark [-h [COMMAND]] [--storage URI] [--netrc [NETRC]]
                  [--verbose]
                  {help,list,compare} ...
```

Commands:

```
help      Display help and exit.
list      List saved runs.
compare   Compare saved runs.
```

The compare command takes almost all the `--benchmark` options, minus the prefix:

positional arguments:

glob_or_file Glob or exact path for json files. If not specified all runs are loaded.

options:

- h, --help** show this help message and exit
- sort=COL** Column to sort on. Can be one of: ‘min’, ‘max’, ‘mean’, ‘stddev’, ‘name’, ‘fullname’. Default: ‘min’
- group-by=LABELS** Comma-separated list of categories by which to group tests. Can be one or more of: ‘group’, ‘name’, ‘fullname’, ‘func’, ‘fullfunc’, ‘param’ or ‘param:NAME’, where NAME is the name passed to `@pytest.parametrize`. Default: ‘group’
- columns=LABELS** Comma-separated list of columns to show in the result table. Default: ‘min, max, mean, stddev, median, iqr, outliers, rounds, iterations’
- name=FORMAT** How to format names in results. Can be one of ‘short’, ‘normal’, ‘long’, or ‘trial’. Default: ‘normal’
- 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>’
- csv=FILENAME** Save a csv report. If FILENAME contains slashes (‘/’) then directories will be created. Default: ‘benchmark_<date>_<time>’
- k EXPR** Only show benchmarks matching the given expression. Uses the same syntax as `pytest`’s `-k` option (e.g. ‘foo and not bar’).

examples:

```
pytest-benchmark compare 'Linux-CPython-3.5-64bit/*'
```

Loads all benchmarks ran with that interpreter. Note the special quoting that disables your shell's glob expansion.

```
pytest-benchmark compare 0001
```

Loads first run from all the interpreters.

```
pytest-benchmark compare /foo/bar/0001_abc.json /lorem/ipsum/
0001_sir_dolor.json
```

Loads runs from exactly those files.

2.2.2 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
```

2.2.3 Extra info

You can set arbitrary values in the `benchmark.extra_info` dictionary, which will be saved in the JSON if you use `--benchmark-autosave` or similar:

```
def test_my_stuff(benchmark):
    benchmark.extra_info['foo'] = 'bar'
    benchmark(time.sleep, 0.02)
```

2.2.4 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
```

(continues on next page)

(continued from previous page)

```
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.

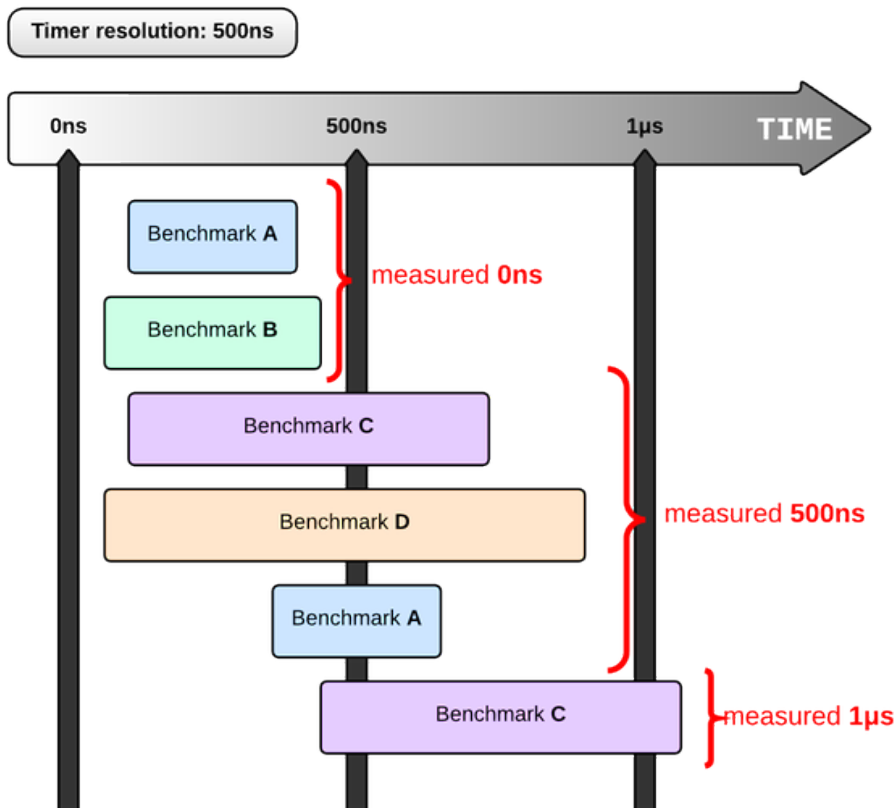
For this there's an experimental `benchmark_weave` fixture that can patch stuff using `aspectlib` (make sure you `pip install aspectlib` or `pip install pytest-benchmark[aspect]`):

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

2.3 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 $10 \times \text{TIMER_RESOLUTION}$ period. You can fine tune this with the `--benchmark-min-time` and `--benchmark-calibration-precision` options.

2.4 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)
```

2.4.1 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 in the first iteration of every round.

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

```
def stuff(a, b, c, foo):
    pass

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) # stuff(1, 2,
    ↪ 3, foo='bar') will be benchmarked
```

Note

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

- **teardown** (*callable*) – A function to call after the last iteration of every round.

```
def stuff(a, b, c, foo):
    pass

def test_with_teardown(benchmark):
    def teardown():
        # cleanup the side effect of the previous bench mark round.
        pass
    benchmark.pedantic(stuff, teardown=teardown, rounds=100)
```

Note

the teardown function receives the same args and kwargs as the target.

- **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 100s (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.

2.5 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 your pytest configuration so you don't have to specify it all the time.

- `--benchmark-save=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.

2.5.1 Comparing outside of pytest

There is a convenience CLI for listing/comparing past runs: `pytest-benchmark` ([Comparison CLI](#)).

Example:

```
pytest-benchmark compare 0001 0002
```

Comparing between source files

When comparing benchmarks from multiple files (e.g. a main branch run vs. a feature branch run), the default output shows all benchmarks in a single flat table. The `--between` flag pivots the table so that each row is a unique benchmark, with columns showing the metric value from each source file and the relative change. It takes a comma-separated list of metrics to compare:

```
pytest-benchmark compare --between=min 0001 0002
```

Example output:

```

----- benchmark: 9 tests, 2 sources -----
↪ -
Name (time in ns)           0001_f41c0c7 Min  0002_8e68892 Min      Min
-----
↪ -
test_getattr_thread_critical      790.9257      245.7993      -68.9
↪ %
test_setattr_thread_critical      899.9952      254.1521      -71.8
↪ %
...

```

The first source is the reference (shown in cyan on the terminal). Each subsequent source is followed by a column showing the percentage change.

You can compare multiple metrics and control the sort order with `--sort`:

```

pytest-benchmark compare --between=min,mean --sort=mean 0001 0002

```

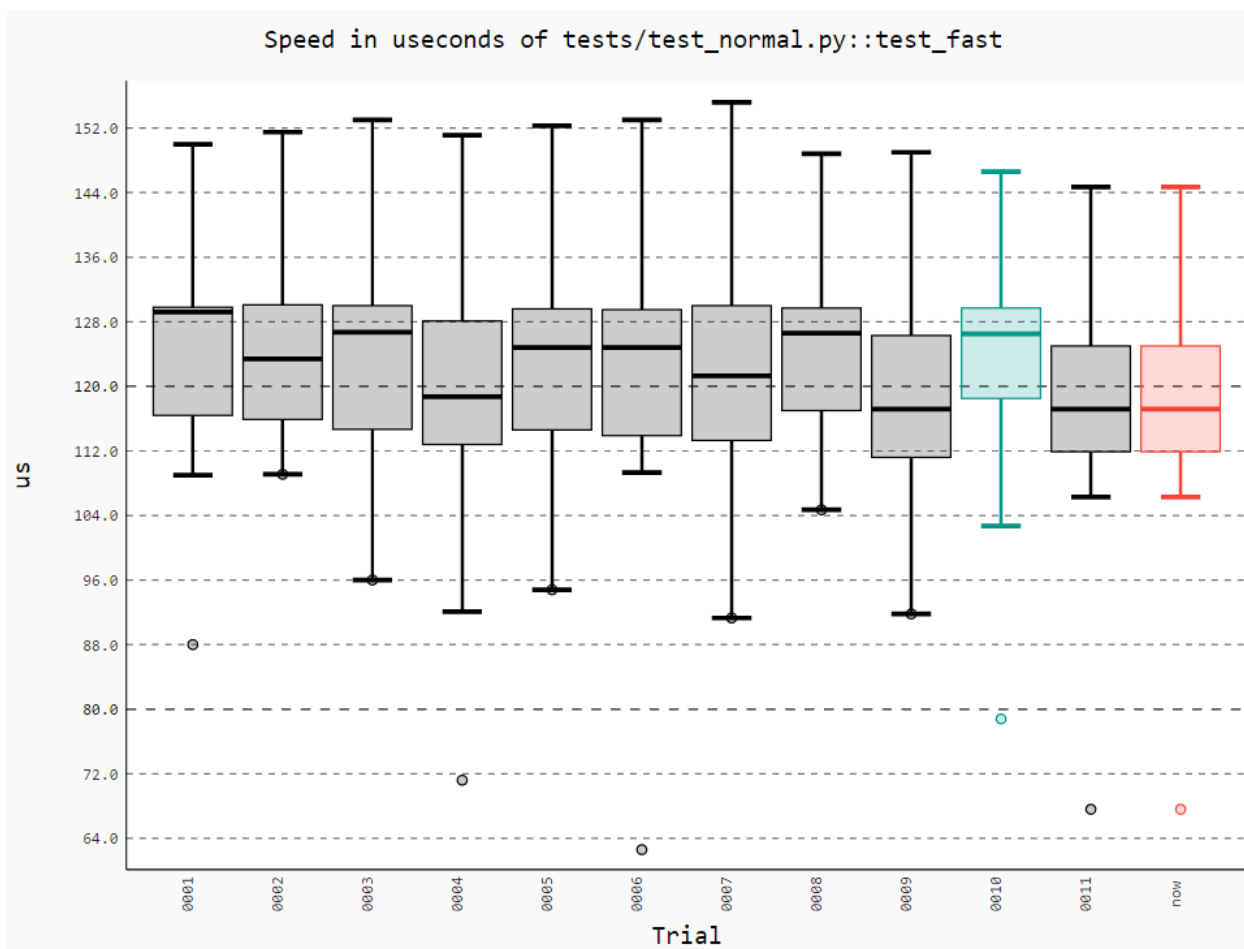
2.5.2 Plotting

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 whisker plot where the outliers (the small bullets) are `Min` and `Max`. Note that if you do not supply a name for the plot it is recommended that `--benchmark-histogram` is the last option passed.

Example output:



2.6 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_benchmark):
    if compared_benchmark["machine_info"] != machine_info:
        benchmarksession.logger.warning(
            "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, machine_info, commit_info)`

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.hookimpl(wrapper=True)
def pytest_benchmark_generate_json(config, benchmarks, include_data, machine_info,
    → commit_info):
    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_generate_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.hookimpl(wrapper=True)
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_scale_unit(config, unit, benchmarks, best, worst, sort)`

To have custom time scaling do something like this:

```
def pytest_benchmark_scale_unit(config, unit, benchmarks, best, worst, sort):
    if unit == 'seconds':
        prefix = ''
        scale = 1.0
```

(continues on next page)

(continued from previous page)

```

elif unit == 'operations':
    prefix = 'K'
    scale = 0.001
else:
    raise RuntimeError("Unexpected measurement unit %r" % unit)

return prefix, scale

```

`pytest_benchmark.hookspec.pytest_benchmark_update_commit_info(config, commit_info)`

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

```

def pytest_benchmark_update_commit_info(config, commit_info):
    commit_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, machine_info)`

If benchmarks are compared and `machine_info` is different, 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, machine_info):
    machine_info['user'] = getpass.getuser()

```

2.7 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.

Why 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 have a feature called [Turbo Boost](#) which overclocks your CPU depending on how many cores you have at that time and how 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.

2.8 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

The arithmetic mean (average) of all round durations.

Median

The middle value when all round durations are sorted. Unlike the mean, the median is not affected by extreme outliers.

IQR

Interquartile Range. The difference between the third quartile (`Q3`) and the first quartile (`Q1`) of round durations. This is a robust way to measure variance that is not affected by outliers.

StdDev

Standard Deviation. A measure of how spread out the round durations are from the mean. A low standard deviation indicates that durations are clustered close to the mean; a high standard deviation indicates more variation.

Outliers

Rounds with durations that fall significantly outside the typical range. Reported as two semicolon-separated counts: `StdDev outliers` (rounds beyond $\text{mean} \pm 1$ standard deviation) and `IQR outliers` (rounds beyond $Q1 + 1.5 \times IQR$ or $Q3 + 1.5 \times IQR$, using Tukey's fences).

2.9 Contributing

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

2.9.1 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.

2.9.2 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.

2.9.3 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 code contributions are welcome :)

2.9.4 Development

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

1. Fork *pytest-benchmark* (look for the “Fork” button).
2. Clone your fork locally:

```
git clone git@github.com:YOURGITHUBNAME/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 and docs builder with 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.

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`).

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`.

Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel*:

```
tox -p auto
```

2.10 Authors

- Ionel Cristian Mărieș - <https://blog.ionelmc.ro>
- Marc Abramowitz - <http://marc-abramowitz.com>
- Dave Collins - <https://github.com/thedavecollins>
- Stefan Krastanov - <http://blog.krastanov.org>
- Thomas Waldmann - <https://github.com/ThomasWaldmann>
- Antonio Cuni - <http://antocuni.eu>
- Petr Šebek - <https://github.com/Artimi>
- Swen Kooij - <https://github.com/Photonios>
- “varac” - <https://github.com/varac>
- Andre Bianchi - <https://github.com/drebs>
- Jeremy Dobbins-Bucklad - <https://github.com/jad-b>
- Alexey Popravka - <https://github.com/popravich>
- Ken Crowell - <https://github.com/oeuftete>
- Matthew Feickert - <https://github.com/matthewfeickert>
- Julien Nicoulaud - <https://github.com/nicoulaj>
- Pablo Aguiar - <https://github.com/scorphus>
- Alex Ford - <https://github.com/asford>
- Francesco Ballarin - <https://github.com/francesco-ballarin>
- Lincoln de Sousa - <https://github.com/clarete>
- Jose Eduardo - <https://github.com/JoseKilo>
- Ofek Lev - <https://github.com/ofek>
- Anton Lodder - <https://github.com/AnjoMan>
- Alexander Duryagin - <https://github.com/daa>
- Stanislav Levin - <https://github.com/stanislavlevin>

- Grygorii Iermolenko - <https://github.com/gyermolenko>
- Jonathan Simon Prates - <https://github.com/jonathansp>
- Miroslav Šedivý - <https://github.com/eumiro>
- Dimitris Rozakis - <https://github.com/dimrozakis>
- Friedrich Delgado - <https://github.com/TauPan>
- Sam James - <https://github.com/thesamesam>
- Florian Bruhin - <https://github.com/The-Compiler>
- Johnny Huang - <https://github.com/jnhyperion>
- Tony Kuo - <https://github.com/tony92151>
- Eugeniy - <https://github.com/zcoder>
- Patrick Winter - <https://github.com/winpat>
- Ali-Akber Saiffee - <https://github.com/alisaiffee>
- Ben Avrahami - <https://github.com/bentheiii>
- Sorin Sbarnea - <https://github.com/ssbarnea>
- Enno Gotthold - <https://github.com/SchoolGuy>
- Thomas B. Brunner - <https://github.com/thomasbbrunner>
- Hugo van Kemenade - <https://github.com/hugovk>
- Aarni Koskela - <https://github.com/akx>

2.11 Changelog

2.11.1 v5.2.3 (2025-11-09)

- Add support for pytest 9.0.
- Moved the README.rst/CHANGELOG.rst concatenation from setup.py to pyproject.toml.

2.11.2 v5.2.2 (2025-11-07)

- Fixed auto-disable to work with newer xdist (pytest-benchmark auto disables benchmarks if xdist is enabled by design). Contributed by Thomas B. Brunner in #294.

2.11.3 v5.2.1 (2025-11-04)

- Add markers so pytest doesn't try to assert-rewrite the plugin internals (fixes those `pytest.PytestAssertRewriteWarning: Module already imported so cannot be rewritten; pytest_benchmark warnings`).

2.11.4 v5.2.0 (2025-10-30)

- Added support for a per-round `teardown` function to pedantic mode. Contributed Patrick Winter by #264.
- Added `--benchmark-time-unit` option. Contributed by Tony Kuo in #281.
- Fixed deprecated hook examples in docstrings. Contributed by Ali-Akber Saiffee in #284.

- Changed `--benchmark-compare-fail` to accept percentages higher than 100%. Contributed by Ben Avrahami in #280.
- Added minimal typing support. Contributed by Sorin Sbarnea in #290.
- Fixed support for Python 3.9. Contributed by Enno Gotthold in #291.
- Replaced the complicated and broken code of `pytest_benchmark.utils.clonefunc` with a simple return of the input. That function was supposed to allow benchmarking with the cost of PyPy JIT included but it's a hassle to maintain.
- Moved the instrumentation pause outside the round loops (in addition to tracing, profiling is paused too). Pedantic mode will keep doing this per round (as the user manually controls the round count). This is necessary because in some scenarios setting and unsetting the tracer too much will overflow an internal counter (found to cause “OverflowError: events set too many times” at least on Python 3.13).

2.11.5 v5.1.0 (2024-10-30)

- Fixed broken hooks handling on pytest 8.1 or later (the `TypeError: import_path() missing 1 required keyword-only argument: 'consider_namespace_packages'` issue). Unfortunately this sets the minimum supported pytest version to 8.1.

2.11.6 v5.0.1 (2024-10-30)

- Fixed bad fixture check that broke down then `nbmake` was enabled.

2.11.7 v5.0.0 (2024-10-29)

- Dropped support for now EOL Python 3.8. Also moved tests suite to only test the latest pytest versions (8.3.x).
- Fix generate parametrize tests benchmark csv report errors (issue #268). Contributed by Johnny Huang in #269.
- Added the `--benchmark-time-unit` cli option for overriding the measurement unit used for display. Contributed by Tony Kuo in #257.
- Fixes spelling in some help texts. Contributed by Eugeniy in #267.
- Added new cprofile options:
 - `--benchmark-cprofile-loops=LOOPS` - previously profiling only ran the function once, this allow customization.
 - `--benchmark-cprofile-top=COUNT` - allows showing more rows.
 - `--benchmark-cprofile-dump=[FILENAME-PREFIX]` - allows saving to a file (that you can load in `snakeviz`, `RunSnakeRun` or other tools).
- Removed hidden dependency on `py.path` (replaced with `pathlib`).

2.11.8 v4.0.0 (2022-10-26)

- Dropped support for legacy Pythons (2.7, 3.6 or older).
- Switched CI to GitHub Actions.
- Removed dependency on the `py` library (that was not properly specified as a dependency anyway).
- Fix skipping test in `test_utils.py` if appropriate VCS not available. Also fix typo. Contributed by Sam James in #211.
- Added support for pytest 7.2.0 by using `pytest.hookimpl` and `pytest.hookspec` to configure hooks. Contributed by Florian Bruhin in #224.

- Now no save is attempted if `--benchmark-disable` is used. Fixes #205. Contributed by Friedrich Delgado in #207.

2.11.9 v3.4.1 (2021-04-17)

- Republished with updated changelog.

I intended to publish a 3.3.0 release but I messed it up because bumpversion doesn't work well with pre-commit apparently... thus 3.4.0 was set in by accident.

2.11.10 v3.4.0 (2021-04-17)

- Disable progress indication unless `--benchmark-verbose` is used. Contributed by Dimitris Rozakis in #149.
- Added Python 3.9, dropped Python 3.5. Contributed by Miroslav Šedivý in #189.
- Changed the “cpu” data in the json output to include everything that `cpuinfo` outputs, for better or worse as `cpuinfo 6.0` changed some fields. Users should now ensure they are an adequate `cpuinfo` package installed. **MAY BE BACKWARDS INCOMPATIBLE**
- Changed behavior of `--benchmark-skip` and `--benchmark-only` to apply early in the collection phase. This means skipped tests won't make `pytest` run fixtures for said tests unnecessarily, but unfortunately this also means the skipping behavior will be applied to any tests that requires a “benchmark” fixture, regardless if it would come from `pytest-benchmark` or not. **MAY BE BACKWARDS INCOMPATIBLE**
- Added `--benchmark-quiet` - option to disable reporting and other information output.
- Squelched unnecessary warning when `--benchmark-disable` and save options are used. Fixes #199.
- `PerformanceRegression` exception no longer inherits `pytest.UsageError` (apparently a *final* class).

2.11.11 v3.2.3 (2020-01-10)

- Fixed “already-imported” `pytest` warning. Contributed by Jonathan Simon Prates in #151.
- Fixed breakage that occurs when benchmark is disabled while using `cprofile` feature (by disabling `cprofile` too).
- Dropped Python 3.4 from the test suite and updated test deps.
- Fixed `pytest_benchmark.utils.clonefunc` to work on Python 3.8.

2.11.12 v3.2.2 (2017-01-12)

- Added support for `pytest` items without `funcargs`. Fixes interoperability with other `pytest` plugins like `pytest-flake8`.

2.11.13 v3.2.1 (2017-01-10)

- Updated changelog entries for 3.2.0. I made the release for `pytest-cov` on the same day and thought I updated the changelogs for both plugins. Alas, I only updated `pytest-cov`.
- Added missing version constraint change. Now `pytest >= 3.8` is required (due to `pytest 4.1` support).
- Fixed couple CI/test issues.
- Fixed broken `pytest_benchmark.__version__`.

2.11.14 v3.2.0 (2017-01-07)

- Added support for simple `trial` x-axis histogram label. Contributed by Ken Crowell in #95).
- Added support for Pytest 3.3+, Contributed by Julien Nicoulaud in #103.
- Added support for Pytest 4.0. Contributed by Pablo Aguiar in #129 and #130.
- Added support for Pytest 4.1.
- Various formatting, spelling and documentation fixes. Contributed by Ken Crowell, Ofek Lev, Matthew Feickert, Jose Eduardo, Anton Lodder, Alexander Duryagin and Grygorii Iermolenko in #97, #105, #110, #111, #115, #123, #131 and #140.
- Fixed broken `pytest_benchmark_update_machine_info` hook. Contributed by Alex Ford in #109.
- Fixed bogus `xdist` warning when using `--benchmark-disable`. Contributed by Francesco Ballarin in #113.
- Added support for `pathlib2`. Contributed by Lincoln de Sousa in #114.
- Changed handling so you can use `--benchmark-skip` and `--benchmark-only`, with the later having priority. Contributed by Ofek Lev in #116.
- Fixed various CI/testing issues. Contributed by Stanislav Levin in #134, #136 and #138.

2.11.15 v3.1.1 (2017-07-26)

- Fixed loading data from old json files (missing `ops` field, see #81).
- Fixed regression on broken SCM (see #82).

2.11.16 v3.1.0 (2017-07-21)

- Added “operations per second” (`ops` field in `Stats`) metric – shows the call rate of code being tested. Contributed by Alexey Popravka in #78.
- Added a `time` field in `commit_info`. Contributed by “varac” in #71.
- Added a `author_time` field in `commit_info`. Contributed by “varac” in #75.
- Fixed the leaking of credentials by masking the URL printed when storing data to elasticsearch.
- Added a `--benchmark-netrc` option to use credentials from a netrc file when storing data to elasticsearch. Both contributed by Andre Bianchi in #73.
- Fixed docs on hooks. Contributed by Andre Bianchi in #74.
- Remove `git` and `hg` as system dependencies when guessing the project name.

2.11.17 3.1.0a2 (2017-03-27)

- `machine_info` now contains more detailed information about the CPU, in particular the exact model. Contributed by Antonio Cuni in #61.
- Added `benchmark.extra_info`, which you can use to save arbitrary stuff in the JSON. Contributed by Antonio Cuni in the same PR as above.
- Fix support for latest PyGal version (histograms). Contributed by Swen Kooij in #68.
- Added support for getting `commit_info` when not running in the root of the repository. Contributed by Vara Canero in #69.
- Added short form for `--storage/--verbose` options in CLI.

- Added an alternate `pytest-benchmark` CLI bin (in addition to `py.test-benchmark`) to match the madness in `pytest`.
- Fix some issues with `--help` in CLI.
- Improved git remote parsing (for `commit_info` in JSON outputs).
- Fixed default value for `--benchmark-columns`.
- Fixed comparison mode (loading was done too late).
- Remove the project name from the autosave name. This will get the old brief naming from 3.0 back.

2.11.18 3.1.0a1 (2016-10-29)

- Added `--benchmark-columns` command line option. It selects what columns are displayed in the result table. Contributed by Antonio Cuni in #34.
- Added support for grouping by specific test parametrization (`--benchmark-group-by=param:NAME` where `NAME` is your param name). Contributed by Antonio Cuni in #37.
- Added support for `name` or `fullname` in `--benchmark-sort`. Contributed by Antonio Cuni in #37.
- Changed signature for `pytest_benchmark_generate_json` hook to take 2 new arguments: `machine_info` and `commit_info`.
- Changed `--benchmark-histogram` to plot groups instead of name-matching runs.
- Changed `--benchmark-histogram` to plot exactly what you compared against. Now it's 1:1 with the compare feature.
- Changed `--benchmark-compare` to allow globs. You can compare against all the previous runs now.
- Changed `--benchmark-group-by` to allow multiple values separated by comma. Example: `--benchmark-group-by=param:foo,param:bar`
- Added a command line tool to compare previous data: `py.test-benchmark`. It has two commands:
 - `list` - Lists all the available files.
 - `compare` - Displays result tables. Takes options:
 - * `--sort=COL`
 - * `--group-by=LABEL`
 - * `--columns=LABELS`
 - * `--histogram=[FILENAME-PREFIX]`
- Added `--benchmark-cprofile` that profiles last run of benchmarked function. Contributed by Petr Šebek.
- Changed `--benchmark-storage` so it now allows elasticsearch storage. It allows to store data to elasticsearch instead to json files. Contributed by Petr Šebek in #58.

2.11.19 v3.0.0 (2015-11-08)

- 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 fixture (but requiring it) will issue a warning (`WBENCHMARK-U1`).

2.11.20 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).

2.11.21 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.

2.11.22 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`.

2.11.23 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).

2.11.24 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.

2.11.25 3.0.0a3 (2015-10-02)

- Added progress indication when computing stats.

2.11.26 3.0.0a2 (2015-09-30)

- Fixed accidental output capturing caused by `capturemanager` misuse.

2.11.27 3.0.0a1 (2015-09-13)

- Added JSON report saving (the `--benchmark-json` command line arguments). Based on initial work from Dave Collins in #8.
- 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.

2.11.28 v2.5.0 (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.

2.11.29 v2.4.1 (2015-03-16)

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

2.11.30 v2.4.0 (2015-03-12)

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

2.11.31 v2.3.0 (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.

2.11.32 v2.2.0 (2014-12-26)

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

2.11.33 v2.1.0 (2014-12-20)

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

2.11.34 v2.0.0 (2014-12-19)

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

2.11.35 v1.0.0 (2014-12-15)

- Use a precise default timer for PyPy.

2.11.36 v0.0.1 - v0.12.1 (2014-10-09 - 2014-12-15)

- README and styling fixes. Contributed by Marc Abramowitz in #4.
- Lots of wild changes and too many irrelevant releases to include.

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

`pytest_benchmark.hookspec`, 15

B

`benchmark.pedantic()`
built-in function, 12
built-in function
 `benchmark.pedantic()`, 12

M

module
 `pytest_benchmark.hookspec`, 15

P

`pytest_benchmark.hookspec`
 module, 15
`pytest_benchmark_compare_machine_info()` (in
 module `pytest_benchmark.hookspec`), 15
`pytest_benchmark_generate_commit_info()` (in
 module `pytest_benchmark.hookspec`), 15
`pytest_benchmark_generate_json()` (in module
 `pytest_benchmark.hookspec`), 16
`pytest_benchmark_generate_machine_info()` (in
 module `pytest_benchmark.hookspec`), 16
`pytest_benchmark_group_stats()` (in module
 `pytest_benchmark.hookspec`), 16
`pytest_benchmark_scale_unit()` (in module
 `pytest_benchmark.hookspec`), 16
`pytest_benchmark_update_commit_info()` (in mod-
 ule `pytest_benchmark.hookspec`), 17
`pytest_benchmark_update_json()` (in module
 `pytest_benchmark.hookspec`), 17
`pytest_benchmark_update_machine_info()` (in
 module `pytest_benchmark.hookspec`), 17