timerit Documentation

Release 1.0.1

Jon Crall

Jan 14, 2023

CONTENTS

Timerit is a powerful multiline alternative to Python's builtin `timeit` module.

Easily do robust timings on existing blocks of code by simply indenting them. There is no need to refactor into a string representation or convert to a single line.

Timerit makes it easy to benchmark complex blocks of code in either scripted or interactive sessions (e.g. Jupyter notebooks). The following example only times a single line, but including more is trivial.

```
>>> import math
>>> from timerit import Timerit
>>> t1 = Timerit(num=200, verbose=2)
>>> for timer in t1:
>>>     setup_vars = 10000
>>>     with timer:
>>>         math.factorial(setup_vars)
>>> # xdoctest: +IGNORE_WANT
>>> print('t1.total_time = %r' % (t1.total_time,))
Timing for 200 loops
Timed for: 200 loops, best of 3
    time per loop: best=2.064 ms, mean=2.115 +- 0.05 ms
t1.total_time = 0.4227177629695507
```

CHAPTER

ONE

TIMERIT PACKAGE

1.1 Submodules

1.1.1 timerit.core module

First, *Timer* is a context manager that times a block of indented code. Also has *tic* and *toc* methods for a more matlab like feel.

Next, *Timerit* is an alternative to the builtin timeit module. I think it is also a better alternative - maybe Tim Peters can show me otherwise. Perhaps there is a reason timeit chooses to work on strings and can't be placed around existing code with a context manager. But as far as I can tell it is simpler and accomplishes the same task.

Example

```
>>> # xdoc: +IGNORE_WANT
>>> #
>>> # The Timerit class allows for robust benchmarking based
>>> # It can be used in normal scripts by simply adjusting the indentation
>>> import math
>>> from timerit import Timerit
>>> for timer in Timerit(num=12, verbose=3):
>>>     with timer:
>>>         math.factorial(100)
Timing for: 200 loops, best of 3
Timed for: 200 loops, best of 3
    body took: 331.840 µs
    time per loop: best=1.569 µs, mean=1.615 ± 0.0 µs
```

```
>>> # xdoc: +SKIP
>>> # In Contrast, timeit is similar, but not having to worry about setup
>>> # and inputing the program as a string, is nice.
>>> import timeit
>>> timeit.timeit(stmt='math.factorial(100)', setup='import math')
1.12695...
```

Example

```
>>> # xdoc: +IGNORE_WANT
>>> #
>>> # The Timer class can also be useful for quick checks
>>> #
>>> import math
>>> from timerit import Timer
>>> timer = Timer('Timer demo!', verbose=1)
>>> x = 100000  # the input for example output
>>> x = 10      # the input for test speed considerations
>>> with timer:
>>>     math.factorial(x)
tic('Timer demo!')
...toc('Timer demo!')=0.1959s
```

class timerit.core.**Timer**(*label=''*, *verbose=None*, *newline=True*, *counter='auto'*)

Bases: `object`

Measures time elapsed between a start and end point.

Can be used as a context manager, or using the MATLAB inspired tic/toc API [MathWorksTic].

Variables

- **tstart** (*float*) – Timestamp of the last "tic" in seconds.
- **elapsed** (*float*) – Number of seconds measured at the last "toc".

References

Example

```
>>> # Create and start the timer using the context manager
>>> import math
>>> from timerit import Timer
>>> timer = Timer('Timer test!', verbose=1)
>>> with timer:
>>>     math.factorial(10)
>>> assert timer.elapsed > 0
tic('Timer test!')
...toc('Timer test!')=...
```

Example

```
>>> # Create and start the timer using the tic/toc interface
>>> import timerit
>>> timer = timerit.Timer().tic()
>>> elapsed1 = timer.toc()
>>> elapsed2 = timer.toc()
>>> elapsed3 = timer.toc()
>>> assert elapsed1 <= elapsed2
>>> assert elapsed2 <= elapsed3
```

Example

```
>>> import timerit
>>> timer1 = timerit.Timer(counter='perf_counter').tic()
>>> print(timer1.toc())
>>> print(timer1.toc())
>>> print(timer1._raw_toc())
>>> print(timer1._raw_toc())
```

Parameters

- **label** (*str*) – Identifier for printing. Default is ‘’.
- **verbose** (*int | None*) – Verbosity level. If unspecified, defaults to is 1 if label is given, otherwise 0.
- **newline** (*bool*) – if False and verbose, print tic and toc on the same line. Defaults to True.
- **counter** (*str*) – Can be ‘auto’, ‘perf_counter’, or ‘perf_counter_ns’ (if Python 3.7+). Defaults to auto.

`property tstart`

Returns: float: The timestamp of the last tic in seconds.

`property elapsed`

Returns: float: The elapsed time duration in seconds

`tic()`

Starts the timer.

Returns
self

Return type
Timer

`toc()`

Stops the timer.

Returns
Amount of time that passed in seconds since the last tic.

Return type
float

`class timerit.core.Timerit(num=1, label=None, bestof=3, unit=None, verbose=None, disable_gc=True, timer_cls=None)`

Bases: `object`

Reports the average time to run a block of code.

Unlike `timeit`, *`Timerit`* can handle multiline blocks of code. It runs inline, and doesn’t depend on magic or strings. Just indent your code and place in a Timerit block.

Variables

- **`measures`** (*`dict`*) – Labeled measurements taken by this object
- **`rankings`** (*`dict`*) – Ranked measurements (useful if more than one measurement was taken)

Example

```
>>> import math
>>> import timerit
>>> num = 3
>>> t1 = timerit.Timerit(num, label='factorial', verbose=1)
>>> for timer in t1:
>>>     # <write untimed setup code here> this example has no setup
>>>     with timer:
>>>         # <write code to time here> for example...
>>>         math.factorial(100)
Timed best=..., mean=... for factorial
>>> # <you can now access Timerit attributes>
>>> assert t1.total_time > 0
>>> assert t1.n_loops == t1.num
>>> assert t1.n_loops == num
```

Example

```
>>> # xdoc: +IGNORE_WANT
>>> import math
>>> import timerit
>>> num = 10
>>> # If the timer object is unused, time will still be recorded,
>>> # but with less precision.
>>> for _ in timerit.Timerit(num, 'concise', verbose=2):
>>>     math.factorial(10000)
Timed concise for: 10 loops, best of 3
    time per loop: best=4.954 ms, mean=4.972 ± 0.018 ms
>>> # Using the timer object results in the most precise timings
>>> for timer in timerit.Timerit(num, 'precise', verbose=3):
>>>     with timer: math.factorial(10000)
Timing precise for: 15 loops, best of 3
Timed precise for: 15 loops, best of 3
    time per loop: best=2.474 ms, mean=2.54 ± 0.046 ms
```

Parameters

- **num** (*int*) – Number of times to run the loop. Defaults to 1.
- **label** (*str | None*) – An identifier for printing and differentiating between different measurements. Can be changed by calling *reset()*. Defaults to None
- **bestof** (*int*) – When computing statistics, groups measurements into chunks of this size and takes the minimum time within each group. This reduces the effective sample size, but improves robustness of the mean to noise in the measurements.
- **unit** (*str | None*) – What units time is reported in. Can be ‘s’, ‘us’, ‘ms’, or ‘ns’. If unspecified a reasonable value is chosen.
- **verbose** (*int | None*) – Verbosity level. Higher is more verbose, distinct text is written at levels 1, 2, and 3. If unspecified, defaults to 1 if label is given and 0 otherwise.
- **disable_gc** (*bool*) – If True, disables the garbage collector while timing, defaults to True.

- **timer_cls** (*None | Any*) – If specified, replaces the default *Timer* class with a customized one. Mainly useful for testing.

reset(*label=None*, *measures=False*)

Clears all measurements, allowing the object to be reused

Parameters

- **label** (*str | None*) – Change the label if specified
- **measures** (*bool, default=False*) – If True reset measures

Returns
self

Return type
Timerit

Example

```
>>> import math
>>> from timerit import Timerit
>>> ti = Timerit(num=10, unit='us', verbose=True)
>>> _ = ti.reset(label='10!').call(math.factorial, 10)
Timed best=...s, mean=...s for 10!
>>> _ = ti.reset(label='20!').call(math.factorial, 20)
Timed best=...s, mean=...s for 20!
>>> _ = ti.reset().call(math.factorial, 20)
Timed best=...s, mean=...s for 20!
>>> _ = ti.reset(measures=True).call(math.factorial, 20)
```

call(*func*, **args*, ***kwargs*)

Alternative way to time a simple function call using condensed syntax.

Returns

self :
Use *min*, or *mean* to get a scalar. Use *print* to output a report to stdout.

Return type
'Timerit'

Returns
self

Return type
Timerit

Example

```
>>> import math
>>> from timerit import Timerit
>>> time = Timerit(num=10).call(math.factorial, 50).min()
>>> assert time > 0
```

robust_times()

Returns a subset of *self.times* where outliers have been rejected.

Returns
The measured times reduced by bestof sampling.

Return type
List[float]

property rankings

Orders each list of measurements by ascending time.

Only useful if the same Timerit object was used to compare multiple code blocks using the reset method to give each a different label.

Returns
A mapping from a statistics type to a mapping from label to values for that statistic.

Return type
Dict[str, Dict[str, float]]

Example

```
>>> import math
>>> from timerit import Timerit
>>> ti = Timerit(num=1)
>>> _ = ti.reset('a').call(math.factorial, 5)
>>> _ = ti.reset('b').call(math.factorial, 10)
>>> _ = ti.reset('c').call(math.factorial, 20)
>>> _ = ti.reset('d').call(math.factorial, 1000)
>>> _ = ti.reset('e').call(math.factorial, 100000)
>>> # xdoctest: +REQUIRES(module:ubelt)
>>> # xdoctest: +IGNORE_WANT
>>> import ubelt as ub
>>> print('ti.rankings = {}'.format(ub.repr2(ti.rankings, nl=2, precision=8)))
>>> print('ti.consistency = {}'.format(ub.repr2(ti.consistency, nl=1,␣
↪precision=8)))
>>> print(ti.summary())
ti.rankings = {
    'mean': {
        'c': 0.00000055,
        'b': 0.00000062,
        'a': 0.00000173,
        'd': 0.00002542,
        'e': 0.07673144,
    },
    'mean+std': {
```

(continues on next page)

(continued from previous page)

```
        'c': 0.00000055,
        'b': 0.00000062,
        'a': 0.00000173,
        'd': 0.00002542,
        'e': 0.07673144,
    },
    'mean-std': {
        'c': 0.00000055,
        'b': 0.00000062,
        'a': 0.00000173,
        'd': 0.00002542,
        'e': 0.07673144,
    },
    'min': {
        'c': 0.00000055,
        'b': 0.00000062,
        'a': 0.00000173,
        'd': 0.00002542,
        'e': 0.07673144,
    },
}
ti.consistency = 1.00000000
d is 99.97% faster than e
a is 93.19% faster than d
b is 64.05% faster than a
c is 11.25% faster than b
```

property consistency

" Take the hamming distance between the preference profiles to as a measure of consistency.

Returns
Hamming distance

Return type
float

min()

The best time overall.

This is typically the best metric to consider when evaluating the execution time of a function. To understand why consider this quote from the docs of the original timeit module:

"' In a typical case, the lowest value gives a lower bound for how fast your machine can run the given code snippet; higher values in the result vector are typically not caused by variability in Python's speed, but by other processes interfering with your timing accuracy. So the min() of the result is probably the only number you should be interested in. "'

Returns
Minimum measured seconds over all trials

Return type
float

Example

```
>>> import math
>>> from timerit import Timerit
>>> self = Timerit(num=10, verbose=0)
>>> self.call(math.factorial, 50)
>>> assert self.min() > 0
```

mean()

The mean of the best results of each trial.

Returns
Mean of measured seconds

Return type
float

Note: This is typically less informative than simply looking at the min. It is recommended to use min as the expectation value rather than mean in most cases.

Example

```
>>> import math
>>> from timerit import Timerit
>>> self = Timerit(num=10, verbose=0)
>>> self.call(math.factorial, 50)
>>> assert self.mean() > 0
```

std()

The standard deviation of the best results of each trial.

Returns
Standard deviation of measured seconds

Return type
float

Note: As mentioned in the timeit source code, the standard deviation is not often useful. Typically the minimum value is most informative.

Example

```
>>> import math
>>> from timerit import Timerit
>>> self = Timerit(num=10, verbose=1)
>>> self.call(math.factorial, 50)
>>> assert self.std() >= 0
```

summary(*stat='mean'*)

Summarize a timerit session.

Only useful if multiple measurements are made with different labels using the reset method.

Parameters
stat (*str*) – Can be mean or min.

Returns
Summary text describing relative change between different labeled measurements.

Return type
str

Example

```
>>> import math
>>> from timerit import Timerit
>>> ti = Timerit(num=1)
>>> x = 32
>>> ti.reset('mul').call(lambda x: x * x, x)
>>> ti.reset('pow').call(lambda x: x ** 2, x)
>>> ti.reset('sum').call(lambda x: sum(x for _ in range(int(x))), x)
>>> print(ti.summary())  # xdoc: +IGNORE_WANT
mul is 48.69% faster than sum
pow is 36.45% faster than mul
```

report(*verbose=1*)

Creates a human readable report

Parameters
verbose (*int*) – Verbosity level. Either 1, 2, or 3.

Returns
The report text summarizing the most recent measurement.

Return type
str

SeeAlso:
Timerit.print()

Example

```
>>> import math
>>> from timerit import Timerit
>>> ti = Timerit(num=1).call(math.factorial, 5)
>>> print(ti.report(verbose=3))  # xdoctest: +IGNORE_WANT
Timed for: 1 loops, best of 1
    body took: 1.742 µs
    time per loop: best=1.742 µs, mean=1.742 ± 0.0 µs
```

print(*verbose=1*)

Prints human readable report using the print function

Parameters
verbose (*int*) – Verbosity level

SeeAlso:
Timerit.report()

Example

```
>>> import math
>>> from timerit import Timer
>>> Timerit(num=10).call(math.factorial, 50).print(verbose=1)
>>> Timerit(num=10).call(math.factorial, 50).print(verbose=2)
>>> Timerit(num=10).call(math.factorial, 50).print(verbose=3)
Timed best=...s, mean=...s
Timed for: 10 loops, best of 3
    time per loop: best=...s, mean=...s
Timed for: 10 loops, best of 3
    body took: ...
    time per loop: best=...s, mean=...s
```

1.1.2 timerit.relative module

Helpers for making relative statements about an increase or decrase

class timerit.relative.**Relative**

Bases: object

static percent_change(*new*, *old*)

new is *old* changed by *percent*

Parameters

- **new** (*Number*) – the value before a change
- **old** (*Number*) – the value after a change

Return type
float

Notes

negative numbers are percent increases positive numbers are percent decreases

Example

```
>>> Relative.percent_change(5, 1)
-400.0
>>> Relative.percent_change(1, 5)
80.0
```

static percent_decrease(*new*, *old*)

new is *percent`% smaller than `old*

Parameters

- **new** (*Number*) – the value before a change
- **old** (*Number*) – the value after a change

Return type
float

```
>>> Relative.percent_decrease(1, 5)
80.0
>>> Relative.percent_decrease(2153, 3469)
37.9360...
```

static percent_increase(*new*, *old*)

new is *percent`% larger than `old*

Parameters

- **new** (*Number*) – the value before a change
- **old** (*Number*) – the value after a change

Return type
float

Example

```
>>> Relative.percent_increase(5, 1)
400.0
>>> Relative.percent_increase(8.6, 8.5)
1.176...
```

static percent_smaller(*new*, *old*)

new is *percent`% smaller than `old*

Parameters

- **new** (*Number*) – the value before a change
- **old** (*Number*) – the value after a change

Returns
a percent decrease

Return type
float

static percent_bigger(*new*, *old*)

new is *percent`% smaller than `old*

Parameters

- **new** (*Number*) – the value before a change
- **old** (*Number*) – the value after a change

Returns
a percent increase

Return type
float

`static percent_slower`(*new*, *old*)

new is X percent slower than *old*

Parameters

- **new** (*float*) – measure of duration before a change
- **old** (*float*) – measure of duration after a change (with same units as new)

Returns
a percent increase in duration

Return type
float

Example

```
>>> from timerit.relative import Relative
>>> old = 8.72848
>>> new = 9.59755
>>> print('{:.3f}% slower'.format(Relative.percent_slower(new, old)))
9.957% slower
>>> new = 3.6053
>>> old = 1.3477
>>> Relative.percent_slower(new, old)
>>> print('{:.3f}% slower'.format(Relative.percent_slower(new, old)))
167.515% slower
```

`static percent_faster`(*new*, *old*)

new is *percent `% faster than `old*

Parameters

- **new** (*float*) – measure of duration before a change
- **old** (*float*) – measure of duration after a change (with same units as new)

Returns
a percent decrease in duration

Return type
float

References

Notes

Equivalent to Relative.percent_decrease, because Faster means time is decreasing.

Example

```
>>> new = 8.59755
>>> old = 8.72848
>>> print('{:.3f}% faster'.format(Relative.percent_faster(new, old)))
1.500% faster
>>> new = 0.6053
>>> old = 1.3477
>>> Relative.percent_faster(new, old)
>>> print('{:.3f}% faster'.format(Relative.percent_faster(new, old)))
55.086% faster
```

1.2 Module contents

Timerit is a powerful multiline alternative to Python's builtin `timeit` module.

Easily do robust timings on existing blocks of code by simply indenting them. There is no need to refactor into a string representation or convert to a single line.

Timerit makes it easy to benchmark complex blocks of code in either scripted or interactive sessions (e.g. Jupyter notebooks). The following example only times a single line, but including more is trivial.

```
>>> import math
>>> from timerit import Timerit
>>> t1 = Timerit(num=200, verbose=2)
>>> for timer in t1:
>>>     setup_vars = 10000
>>>     with timer:
>>>         math.factorial(setup_vars)
>>> # xdoctest: +IGNORE_WANT
>>> print('t1.total_time = %r' % (t1.total_time,))
Timing for 200 loops
Timed for: 200 loops, best of 3
    time per loop: best=2.064 ms, mean=2.115 +- 0.05 ms
t1.total_time = 0.4227177629695507
```

class timerit.**Timer**(*label=''*, *verbose=None*, *newline=True*, *counter='auto'*)

Bases: object

Measures time elapsed between a start and end point.

Can be used as a context manager, or using the MATLAB inspired tic/toc API [MathWorksTic].

Variables

- **tstart** (*float*) – Timestamp of the last "tic" in seconds.
- **elapsed** (*float*) – Number of seconds measured at the last "toc".

References

Example

```
>>> # Create and start the timer using the context manager
>>> import math
>>> from timerit import Timer
>>> timer = Timer('Timer test!', verbose=1)
>>> with timer:
>>>     math.factorial(10)
>>> assert timer.elapsed > 0
tic('Timer test!')
...toc('Timer test!')=...
```

Example

```
>>> # Create and start the timer using the tic/toc interface
>>> import timerit
>>> timer = timerit.Timer().tic()
>>> elapsed1 = timer.toc()
>>> elapsed2 = timer.toc()
>>> elapsed3 = timer.toc()
>>> assert elapsed1 <= elapsed2
>>> assert elapsed2 <= elapsed3
```

Example

```
>>> import timerit
>>> timer1 = timerit.Timer(counter='perf_counter').tic()
>>> print(timer1.toc())
>>> print(timer1.toc())
>>> print(timer1._raw_toc())
>>> print(timer1._raw_toc())
```

Parameters

- **label** (*str*) – Identifier for printing. Default is ‘’.
- **verbose** (*int | None*) – Verbosity level. If unspecified, defaults to is 1 if label is given, otherwise 0.
- **newline** (*bool*) – if False and verbose, print tic and toc on the same line. Defaults to True.
- **counter** (*str*) – Can be ‘auto’, ‘perf_counter’, or ‘perf_counter_ns’ (if Python 3.7+). Defaults to auto.

property tstart

Returns: float: The timestamp of the last tic in seconds.

property elapsed

Returns: float: The elapsed time duration in seconds

tic()

Starts the timer.

Returns
self

Return type
Timer

toc()

Stops the timer.

Returns
Amount of time that passed in seconds since the last tic.

Return type
float

class timerit.**Timerit**(*num=1*, *label=None*, *bestof=3*, *unit=None*, *verbose=None*, *disable_gc=True*, *timer_cls=None*)

Bases: object

Reports the average time to run a block of code.

Unlike `timeit`, *Timerit* can handle multiline blocks of code. It runs inline, and doesn't depend on magic or strings. Just indent your code and place in a Timerit block.

Variables

- **measures** (*dict*) – Labeled measurements taken by this object
- **rankings** (*dict*) – Ranked measurements (useful if more than one measurement was taken)

Example

```
>>> import math
>>> import timerit
>>> num = 3
>>> t1 = timerit.Timerit(num, label='factorial', verbose=1)
>>> for timer in t1:
>>>     # <write untimed setup code here> this example has no setup
>>>     with timer:
>>>         # <write code to time here> for example...
>>>         math.factorial(100)
Timed best=..., mean=... for factorial
>>> # <you can now access Timerit attributes>
>>> assert t1.total_time > 0
>>> assert t1.n_loops == t1.num
>>> assert t1.n_loops == num
```

Example

```
>>> # xdoc: +IGNORE_WANT
>>> import math
>>> import timerit
>>> num = 10
>>> # If the timer object is unused, time will still be recorded,
>>> # but with less precision.
>>> for _ in timerit.Timerit(num, 'concise', verbose=2):
>>>     math.factorial(10000)
Timed concise for: 10 loops, best of 3
    time per loop: best=4.954 ms, mean=4.972 ± 0.018 ms
>>> # Using the timer object results in the most precise timings
>>> for timer in timerit.Timerit(num, 'precise', verbose=3):
>>>     with timer: math.factorial(10000)
Timing precise for: 15 loops, best of 3
Timed precise for: 15 loops, best of 3
    time per loop: best=2.474 ms, mean=2.54 ± 0.046 ms
```

Parameters

- **num** (*int*) – Number of times to run the loop. Defaults to 1.
- **label** (*str | None*) – An identifier for printing and differentiating between different measurements. Can be changed by calling *reset()*. Defaults to None
- **bestof** (*int*) – When computing statistics, groups measurements into chunks of this size and takes the minimum time within each group. This reduces the effective sample size, but improves robustness of the mean to noise in the measurements.
- **unit** (*str | None*) – What units time is reported in. Can be ‘s’, ‘us’, ‘ms’, or ‘ns’. If unspecified a reasonable value is chosen.
- **verbose** (*int | None*) – Verbosity level. Higher is more verbose, distinct text is written at levels 1, 2, and 3. If unspecified, defaults to 1 if label is given and 0 otherwise.
- **disable_gc** (*bool*) – If True, disables the garbage collector while timing, defaults to True.
- **timer_cls** (*None | Any*) – If specified, replaces the default *Timer* class with a customized one. Mainly useful for testing.

reset(*label=None*, *measures=False*)

Clears all measurements, allowing the object to be reused

Parameters

- **label** (*str | None*) – Change the label if specified
- **measures** (*bool, default=False*) – If True reset measures

Returns

self

Return type

Timerit

Example

```
>>> import math
>>> from timerit import Timerit
>>> ti = Timerit(num=10, unit='us', verbose=True)
>>> _ = ti.reset(label='10!').call(math.factorial, 10)
Timed best=...s, mean=...s for 10!
>>> _ = ti.reset(label='20!').call(math.factorial, 20)
Timed best=...s, mean=...s for 20!
>>> _ = ti.reset().call(math.factorial, 20)
Timed best=...s, mean=...s for 20!
>>> _ = ti.reset(measures=True).call(math.factorial, 20)
```

call(*func*, **args*, ***kwargs*)

Alternative way to time a simple function call using condensed syntax.

Returns

self :
Use *min*, or *mean* to get a scalar. Use *print* to output a report to stdout.

Return type
'Timerit'

Returns
self

Return type
Timerit

Example

```
>>> import math
>>> from timerit import Timerit
>>> time = Timerit(num=10).call(math.factorial, 50).min()
>>> assert time > 0
```

robust_times()

Returns a subset of *self.times* where outliers have been rejected.

Returns
The measured times reduced by bestof sampling.

Return type
List[float]

property rankings

Orders each list of measurements by ascending time.

Only useful if the same Timerit object was used to compare multiple code blocks using the reset method to give each a different label.

Returns
A mapping from a statistics type to a mapping from label to values for that statistic.

Return type
Dict[str, Dict[str, float]]

Example

```
>>> import math
>>> from timerit import Timerit
>>> ti = Timerit(num=1)
>>> _ = ti.reset('a').call(math.factorial, 5)
>>> _ = ti.reset('b').call(math.factorial, 10)
>>> _ = ti.reset('c').call(math.factorial, 20)
>>> _ = ti.reset('d').call(math.factorial, 1000)
>>> _ = ti.reset('e').call(math.factorial, 100000)
>>> # xdoctest: +REQUIRES(module:ubelt)
>>> # xdoctest: +IGNORE_WANT
>>> import ubelt as ub
>>> print('ti.rankings = {}'.format(ub.repr2(ti.rankings, nl=2, precision=8)))
>>> print('ti.consistency = {}'.format(ub.repr2(ti.consistency, nl=1,␣
↪precision=8)))
>>> print(ti.summary())
ti.rankings = {
    'mean': {
        'c': 0.00000055,
        'b': 0.00000062,
        'a': 0.00000173,
        'd': 0.00002542,
        'e': 0.07673144,
    },
    'mean+std': {
        'c': 0.00000055,
        'b': 0.00000062,
        'a': 0.00000173,
        'd': 0.00002542,
        'e': 0.07673144,
    },
    'mean-std': {
        'c': 0.00000055,
        'b': 0.00000062,
        'a': 0.00000173,
        'd': 0.00002542,
        'e': 0.07673144,
    },
    'min': {
        'c': 0.00000055,
        'b': 0.00000062,
        'a': 0.00000173,
        'd': 0.00002542,
        'e': 0.07673144,
    },
}
ti.consistency = 1.00000000
d is 99.97% faster than e
a is 93.19% faster than d
b is 64.05% faster than a
c is 11.25% faster than b
```

property consistency

” Take the hamming distance between the preference profiles to as a measure of consistency.

Returns
Hamming distance

Return type
float

min()

The best time overall.

This is typically the best metric to consider when evaluating the execution time of a function. To understand why consider this quote from the docs of the original timeit module:

‘” In a typical case, the lowest value gives a lower bound for how fast your machine can run the given code snippet; higher values in the result vector are typically not caused by variability in Python’s speed, but by other processes interfering with your timing accuracy. So the min() of the result is probably the only number you should be interested in. “’

Returns
Minimum measured seconds over all trials

Return type
float

Example

```
>>> import math
>>> from timerit import Timerit
>>> self = Timerit(num=10, verbose=0)
>>> self.call(math.factorial, 50)
>>> assert self.min() > 0
```

mean()

The mean of the best results of each trial.

Returns
Mean of measured seconds

Return type
float

Note: This is typically less informative than simply looking at the min. It is recommended to use min as the expectation value rather than mean in most cases.

Example

```
>>> import math
>>> from timerit import Timerit
>>> self = Timerit(num=10, verbose=0)
>>> self.call(math.factorial, 50)
>>> assert self.mean() > 0
```

std()

The standard deviation of the best results of each trial.

Returns
Standard deviation of measured seconds

Return type
float

Note: As mentioned in the timeit source code, the standard deviation is not often useful. Typically the minimum value is most informative.

Example

```
>>> import math
>>> from timerit import Timerit
>>> self = Timerit(num=10, verbose=1)
>>> self.call(math.factorial, 50)
>>> assert self.std() >= 0
```

summary(*stat='mean'*)

Summarize a timerit session.

Only useful if multiple measurements are made with different labels using the reset method.

Parameters
stat (*str*) – Can be mean or min.

Returns
Summary text describing relative change between different labeled measurements.

Return type
str

Example

```
>>> import math
>>> from timerit import Timerit
>>> ti = Timerit(num=1)
>>> x = 32
>>> ti.reset('mul').call(lambda x: x * x, x)
>>> ti.reset('pow').call(lambda x: x ** 2, x)
>>> ti.reset('sum').call(lambda x: sum(x for _ in range(int(x))), x)
>>> print(ti.summary())  # xdoc: +IGNORE_WANT
```

(continues on next page)

(continued from previous page)

```
mul is 48.69% faster than sum
pow is 36.45% faster than mul
```

report(*verbose=1*)

Creates a human readable report

Parameters
verbose (*int*) – Verbosity level. Either 1, 2, or 3.

Returns
The report text summarizing the most recent measurement.

Return type
str

SeeAlso:
Timerit.print()

Example

```
>>> import math
>>> from timerit import Timerit
>>> ti = Timerit(num=1).call(math.factorial, 5)
>>> print(ti.report(verbose=3))  # xdoctest: +IGNORE_WANT
Timed for: 1 loops, best of 1
    body took: 1.742 µs
    time per loop: best=1.742 µs, mean=1.742 ± 0.0 µs
```

print(*verbose=1*)

Prints human readable report using the print function

Parameters
verbose (*int*) – Verbosity level

SeeAlso:
Timerit.report()

Example

```
>>> import math
>>> from timerit import Timer
>>> Timerit(num=10).call(math.factorial, 50).print(verbose=1)
>>> Timerit(num=10).call(math.factorial, 50).print(verbose=2)
>>> Timerit(num=10).call(math.factorial, 50).print(verbose=3)
Timed best=...s, mean=...s
Timed for: 10 loops, best of 3
    time per loop: best=...s, mean=...s
Timed for: 10 loops, best of 3
    body took: ...
    time per loop: best=...s, mean=...s
```

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex

BIBLIOGRAPHY

[MathWorksTic] https://www.mathworks.com/help/matlab/ref/tic.html

[SO8127862] https://stackoverflow.com/questions/8127862/how-do-you-calculate-how-much-faster-time-x-is-from-time-y-in-terms-o

[SO716767] https://math.stackexchange.com/questions/716767/how-to-calculate-the-percentage-of-increase-decrease-with-negative-n
716770#716770

[MathWorksTic] https://www.mathworks.com/help/matlab/ref/tic.html

PYTHON MODULE INDEX

t

INDEX