measure_it Documentation

Release 0.5.0

Pete Fein

Apr 16, 2018

Contents

instrument provides instrumentation primitives for metrics and benchmarking.

Python *Instrument* was formerly known as *Measure It*.

author Pete Fein <pete@wearpants.org>

license BSD

versions Python 2.7 & 3

source https://github.com/wearpants/instrument

homepage https://instrument.readthedocs.org/en/latest/

CHAPTER 1

Installation

instrument subscribes to the Python batteries-included philosophy. It ships with support for a number of different *metric backends*, and eventually instrumentation for popular packages. Since most users will want only a subset of this functionality, optional dependencies are not installed by default.

Note that the included `requirements.txt` file includes *all* depedencies (for build/test purposes), and is almost certainly not what you want.

1.1 Minimal

To install the base package (with no depedencies beyond the standard library):

```
pip install instrument
```

This includes: all generic measurement functions, *`print_metric()`* and *`csv`* metrics.

1.2 Batteries Included

You should have completed the minimal install already. To install the dependencies for an optional component, specify it in brackets with `--upgrade`:

```
pip install --upgrade instrument[statsd]
```

The following extra targets are supported:

- `statsd`: statsd metric
- `numpy`: statistics metrics, based on numpy
- `plot`: graphs of metrics, based on matplotlib. Because Python packaging is brain damaged, you must install the `numpy` target first. You'll need the agg backend.

CHAPTER 2

Basic Usage

2.1 Measurements

Iterators & generators often encapsulate I/O, number crunching or other operations we want to gather metrics for:

```
>>> from time import sleep
>>> def math_is_hard(N):
...     x = 0
...     while x < N:
...         sleep(.1)
...         yield x * x
...         x += 1
```

Timing iterators is tricky. *instrument.iter()*, *instrument.each()* and *instrument.first()* record metrics for time and element count for iteratables.

```
>>> import instrument
```

Wrap an iterator in *instrument.iter()* to time how long it takes to consume entirely:

```
>>> underlying = math_is_hard(5)
>>> underlying
<generator object math_is_hard at ...>
>>> _ = instrument.iter(underlying)
>>> squares = list(_)
5 elements in 0.50 seconds
```

The wrapped iterator yields the same results as the underlying iterable:

```
>>> squares
[0, 1, 4, 9, 16]
```

The *instrument.each()* wrapper measures the time taken to produce each item:

```
>>> _ = instrument.each(math_is_hard(5))
>>> list(_)
1 elements in 0.10 seconds
1 elements in 0.10 seconds
1 elements in 0.10 seconds
1 elements in 0.10 seconds
1 elements in 0.10 seconds
[0, 1, 4, 9, 16]
```

The *instrument.first()* wrapper measures the time taken to produce the first item:

```
>>> _ = instrument.first(math_is_hard(5))
>>> list(_)
1 elements in 0.10 seconds
[0, 1, 4, 9, 16]
```

You can provide a custom name for the metric:

```
>>> _ = instrument.iter(math_is_hard(5), name="bogomips")
>>> list(_)
bogomips: 5 elements in 0.50 seconds
[0, 1, 4, 9, 16]
```

2.2 Decorators

If you have a generator function (one that uses `yield`), you can wrap it with a decorator using `.func()`. You can pass the same `name` and `metric` arugments:

```
>>> @instrument.each.func()
... def slow(N):
...     for i in range(N):
...         sleep(.1)
...         yield i
>>> list(slow(3))
__main__.slow: 1 elements in 0.10 seconds
__main__.slow: 1 elements in 0.10 seconds
__main__.slow: 1 elements in 0.10 seconds
[0, 1, 2]
```

Decorators work inside classes too. If you don't provide a name, a decent one will be inferred:

```
>>> class Database(object):
...     @instrument.iter.func()
...     def batch_get(self, ids):
...         # you'd actually hit database here
...         for i in ids:
...             sleep(.1)
...             yield {"id":i, "square": i*i}
>>> database = Database()
>>> _ = database.batch_get([1, 2, 3, 9000])
>>> list(_)
__main__.Database.batch_get: 4 elements in 0.40 seconds
[{'id': 1, 'square': 1}, {'id': 2, 'square': 4}, {'id': 3, 'square': 9}, {'id': 9000,
↪'square': 81000000}]
```

2.3 Reducers & Producers

instrument.reduce() and *instrument.produce()* are decorators for functions, *not* iterators.

The *instrument.reduce()* decorator measures functions that consume many items. Examples include aggregators or a *batch_save()*:

```
>>> @instrument.reduce()
... def sum_squares(L):
...     total = 0
...     for i in L:
...         sleep(.1)
...         total += i*i
...     return total
...
>>> sum_squares(range(5))
__main__.sum_squares: 5 elements in 0.50 seconds
30
```

This works with `*args` functions too:

```
>>> @instrument.reduce()
... def sum_squares2(*args):
...     total = 0
...     for i in args:
...         sleep(.1)
...         total += i*i
...     return total
...
>>> sum_squares2(*range(5))
__main__.sum_squares2: 5 elements in 0.50 seconds
30
```

The *instrument.produce()* decorator measures a function that produces many items. This is similar to `instrument.iter.func()`, but for functions that return lists instead of iterators (or other object supporting `len()`):

```
>>> @instrument.produce()
... def list_squares(N):
...     sleep(0.1 * N)
...     return [i * i for i in range(N)]
>>> list_squares(5)
__main__.list_squares: 5 elements in 0.50 seconds
[0, 1, 4, 9, 16]
```

instrument.reduce() and *instrument.produce()* can be used inside classes:

```
>>> class Database(object):
...     @instrument.reduce()
...     def batch_save(self, rows):
...         for r in rows:
...             # you'd actually commit to your database here
...             sleep(0.1)
...
...     @instrument.reduce()
...     def batch_save2(self, *rows):
...         for r in rows:
...             # you'd actually commit to your database here
```

```
...             sleep(0.1)
...
...     @instrument.produce()
...     def dumb_query(self, x):
...         # you'd actually talk to your database here
...         sleep(0.1 * x)
...         return [{"id":i, "square": i*i} for i in range(x)]
...
>>> rows = [{'id':i} for i in range(5)]
>>> database = Database()
>>> database.batch_save(rows)
__main__.Database.batch_save: 5 elements in 0.50 seconds
>>> database.batch_save2(*rows)
__main__.Database.batch_save2: 5 elements in 0.50 seconds
>>> database.dumb_query(3)
__main__.Database.dumb_query: 3 elements in 0.30 seconds
[{'id': 0, 'square': 0}, {'id': 1, 'square': 1}, {'id': 2, 'square': 4}]
```

2.4 Functions

The *instrument.func()* decorator simply measures total function execution time:

```
>>> @instrument.func()
... def slow():
...     # you'd do something useful here
...     sleep(.1)
...     return "SLOW"
>>> slow()
__main__.slow: 1 elements in 0.10 seconds
'SLOW'
```

This works in classes too:

```
>>> class CrunchCrunch(object):
...     @instrument.func()
...     def slow(self):
...         # you'd do something useful here
...         sleep(.1)
...         return "SLOW"
>>> CrunchCrunch().slow()
__main__.CrunchCrunch.slow: 1 elements in 0.10 seconds
'SLOW'
```

2.5 Blocks

To measure the excecution time of a block of code, use a *instrument.block()* context manager:

```
>>> with instrument.block(name="slowcode"):
...     # you'd do something useful here
...     sleep(0.2)
slowcode: 1 elements in 0.20 seconds
```

You can also pass your own value for *count*; this is useful to measure a resource used by a block (the number of bytes in a file, for example):

```
>>> with instrument.block(name="slowcode", count=42):
...     # you'd do something useful here
...     sleep(0.2)
slowcode: 42 elements in 0.20 seconds
```

CHAPTER 3

Data Output

By default, metrics are printed to standard out. You can provide your own metric recording funtion. It should take three arguments: `count` of items, `elapsed` time in seconds, and *name*, which can be None:

```
>>> def my_metric(name, count, elapsed):
...     print("Iterable %s produced %d items in %d milliseconds"%(name, count,␣
↪int(round(elapsed*1000))))
...
>>> _ = instrument.iter(math_is_hard(5), metric=my_metric, name="bogomips")
>>> list(_)
Iterable bogomips produced 5 items in 5000 milliseconds
[0, 1, 4, 9, 16]
```

Unless individually specified, metrics are reported using the global *default_metric()*. To change the active default, simply assign another metric function to this attribute. In general, you should configure your metric functions at program startup, **before** recording any metrics. *make_multi_metric()* creates a single metric function that records to several outputs.

3.1 Comma Separated

csv saves raw metrics as comma separated text files. This is useful for conducting external analysis. *csv* is thread-safe; use under multiprocessing requires some care.

CSVFileMetric saves all metrics to a single file with three columns: metric name, item count & elapsed time. Create an instance of this class and pass its *CSVFileMetric.metric()* method to measurement functions. The `outfile` parameter controls where to write data; an existing file will be overwritten.

```
>>> from instrument.csv import CSVFileMetric
>>> csvfm = CSVFileMetric("/tmp/my_metrics_file.csv")
>>> _ = instrument.iter(math_is_hard(5), metric=csvfm.metric, name="bogomips")
>>> list(_)
[0, 1, 4, 9, 16]
```

CSVDirMetric saves metrics to multiple files, named after each metric, with two columns: item count & elapsed time. This class is global to your program; do not manually create instances. Instead, use the classmethod *CSVDirMetric.metric()*. Set the class variable `outdir` to a directory in which to store files. The contents of this directory will be deleted on startup.

```
>>> from instrument.csv import CSVDirMetric
>>> CSVDirMetric.outdir = "/tmp/my_metrics_dir"
>>> _ = instrument.iter(math_is_hard(5), metric=CSVDirMetric.metric, name="bogomips")
>>> list(_)
[0, 1, 4, 9, 16]
```

Both classes support at `dump_atexit` flag, which will register a handler to write data when the interpreter finishes execution. Set to false to manage yourself.

3.2 Plots & Statistics

numpy generates aggregate plots (graphs) and statistics. This is useful for benchmarking or batch jobs; for live systems, *statsd (graphite)* is a better choice. *numpy* is threadsafe; use under multiprocessing requires some care.

NumpyMetric subclasses are global to your program; do not manually create instances. Instead, use the classmethod *NumpyMetric.metric()*. The `dump_atexit` flag will register a handler to write data when the interpreter finishes execution. Set to false to manage yourself.

```
>>> from instrument.numpy import TableMetric, PlotMetric
>>> _ = instrument.iter(math_is_hard(5), metric=TableMetric.metric, name="bogomips")
>>> list(_)
[0, 1, 4, 9, 16]
```

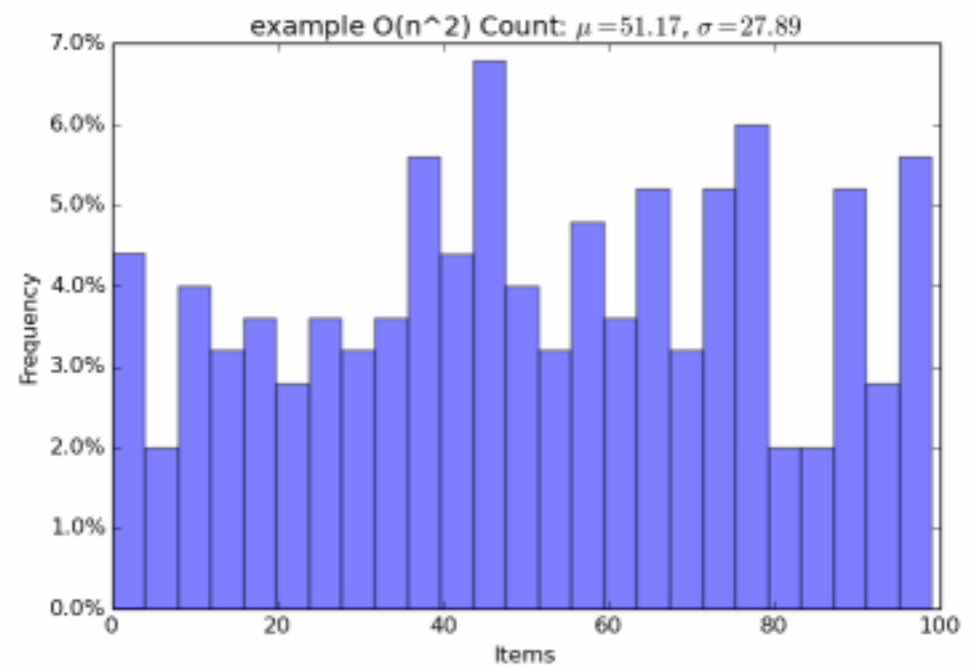
TableMetric prints pretty tables of aggregate population statistics. Set the class variable `outfile` to a file-like object (defaults to `stderr`):

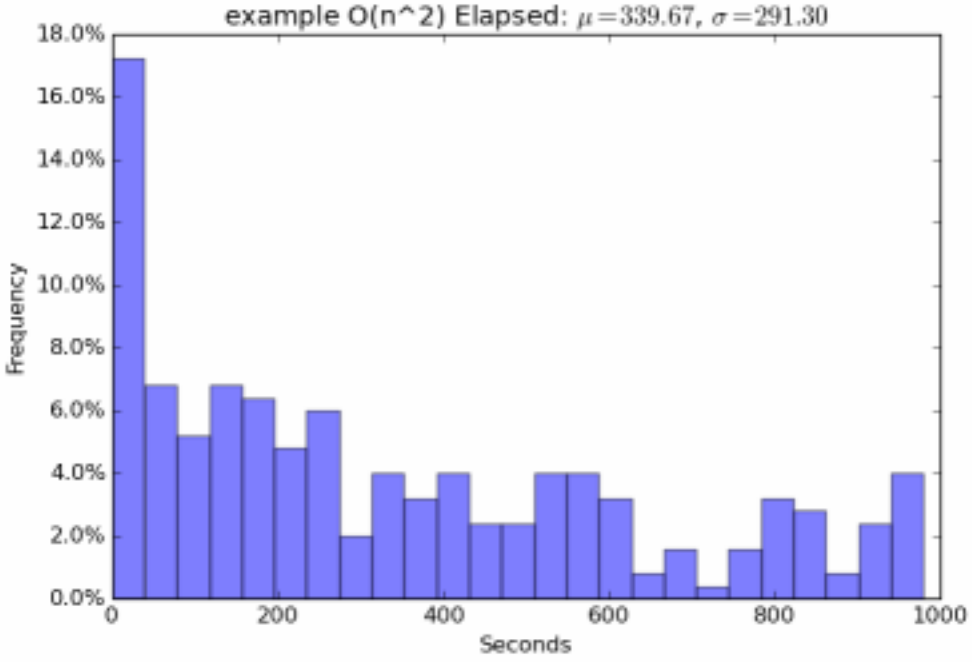
```
Name            Count Mean        Count Stddev        Elapsed Mean        Elapsed␣
↪Stddev
alice             47.96             28.44              310.85              291.16
bob               50.08             28.84              333.98              297.11
charles           51.79             29.22              353.58              300.82
```

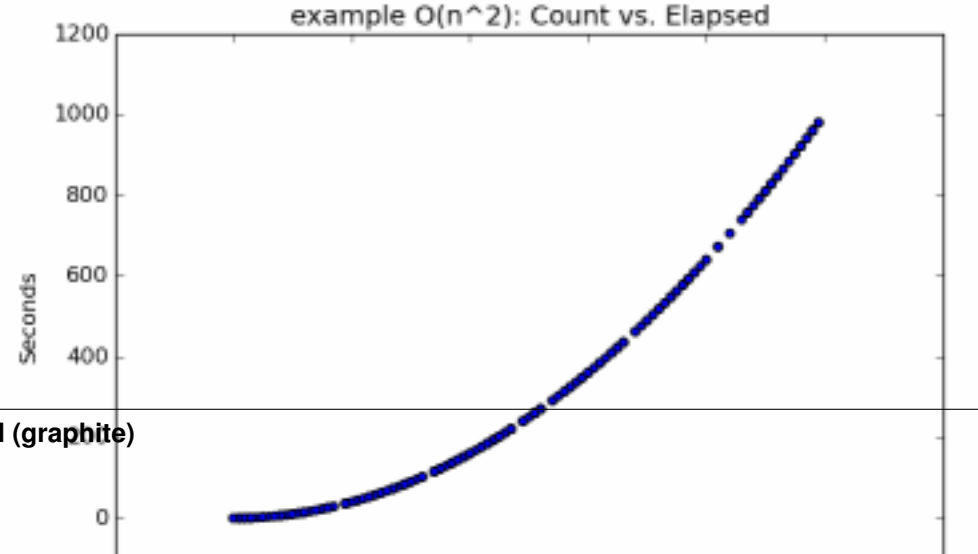
PlotMetric generates plots using matplotlib. Plots are saved to multiple files, named after each metric. Set the class variable `outdir` to a directory in which to store files. The contents of this directory will be deleted on startup.

3.3 statsd (graphite)

For monitoring production systems, the *statsd_metric()* function can be used to record metrics to statsd. Each metric will generate two buckets: a count and a timing.


example O(n^2) Count: $\mu=51.17$, $\sigma=27.89$
Frequency
Items
0.0%
1.0%
2.0%
3.0%
4.0%
5.0%
6.0%
7.0%
0
20
40
60
80
100


example O(n^2) Elapsed: $\mu=339.67$, $\sigma=291.30$
Frequency
Seconds
0.0%
2.0%
4.0%
6.0%
8.0%
10.0%
12.0%
14.0%
16.0%
18.0%
0
200
400
600
800
1000


example O(n^2): Count vs. Elapsed
Seconds
0
400
600
800
1000
1200

CHAPTER 4

API Documentation

4.1 instrument

you are not expected to understand this implementation. that's why it has tests. the above-mentioned 'you' includes the author. :-}

`instrument.`**`iter`**(*iterable*, *name=None*, *metric=<function call_default>*)
Measure total time and element count for consuming an iterable

Parameters

- **`iterable`** – any iterable
- **`metric`** (*`function`*) – f(name, count, total_time)
- **`name`** (*`str`*) – name for the metric

`instrument.`**`each`**(*iterable*, *name=None*, *metric=<function call_default>*)
Measure time elapsed to produce each item of an iterable

Parameters

- **`iterable`** – any iterable
- **`metric`** (*`function`*) – f(name, 1, time)
- **`name`** (*`str`*) – name for the metric

`instrument.`**`reduce`**(*name=None*, *metric=<function call_default>*)
Decorator to measure a function that consumes many items.

The wrapped `func` should take either a single `iterable` argument or `*args` (plus keyword arguments).

Parameters

- **`metric`** (*`function`*) – f(name, count, total_time)
- **`name`** (*`str`*) – name for the metric

instrument.**produce**(*name=None*, *metric=<function call_default>*)
Decorator to measure a function that produces many items.

The function should return an object that supports __len__ (ie, a list). If the function returns an iterator, use `iter.func()` instead.

Parameters

- **metric** (*function*) – f(name, count, total_time)
- **name** (*str*) – name for the metric

instrument.**func**(*name=None*, *metric=<function call_default>*)
Decorator to measure function execution time.

Parameters

- **metric** (*function*) – f(name, 1, total_time)
- **name** (*str*) – name for the metric

instrument.**first**(*iterable*, *name=None*, *metric=<function call_default>*)
Measure time elapsed to produce first item of an iterable

Parameters

- **iterable** – any iterable
- **metric** (*function*) – f(name, 1, time)
- **name** (*str*) – name for the metric

instrument.**block**(**args*, ***kwds*)
Context manager to measure execution time of a block

Parameters

- **metric** (*function*) – f(name, 1, time)
- **name** (*str*) – name for the metric
- **count** (*int*) – user-supplied number of items, defaults to 1

instrument.**print_metric**(*name*, *count*, *elapsed*)
A metric function that prints

Parameters

- **name** (*str*) – name of the metric
- **count** (*int*) – number of elements
- **elapsed** (*float*) – time in seconds

instrument.**default_metric**(*name*, *count*, *elapsed*)
A metric function that prints

Parameters

- **name** (*str*) – name of the metric
- **count** (*int*) – number of elements
- **elapsed** (*float*) – time in seconds

instrument.**make_multi_metric**(**metrics*)
Make a new metric function that calls the supplied metrics

Parameters **metrics** (*functions*) – metric functions

Return type function

4.2 instrument.csv

class instrument.csv.**CSVDirMetric**(*name*)
Write metrics to multiple CSV files

Do not create instances of this class directly. Simply pass the classmethod *metric()* to a measurement function. Output using *dump()*. These are the only public methods.

Each metric consumes one open file and 32K of memory while running.

Variables

- **dump_atexit** – automatically call *dump()* when the interpreter exits. Defaults to True.
- **outdir** – directory to save CSV files in. Defaults to ./mit_csv.

classmethod dump()
Output all recorded metrics

classmethod metric(*name*, *count*, *elapsed*)
A metric function that writes multiple CSV files

Parameters

- **name** (*str*) – name of the metric
- **count** (*int*) – number of elements
- **elapsed** (*float*) – time in seconds

class instrument.csv.**CSVFileMetric**(*outfile='mit.csv'*, *dump_atexit=True*)
Write metrics to a single CSV file

Pass the method *metric()* to a measurement function. Output using *dump()*. These are the only public methods.

Variables

- **outfile** – file to save to. Defaults to ./mit.csv.
- **dump_atexit** – automatically call *dump()* when the interpreter exits. Defaults to True.

dump()
Output all recorded metrics

metric(*name*, *count*, *elapsed*)
A metric function that writes a single CSV file

Parameters

- **name** (*str*) – name of the metric
- **count** (*int*) – number of elements
- **elapsed** (*float*) – time in seconds

4.3 instrument.numpy

numpy-based metrics

class instrument.numpy.**NumpyMetric**
Do not create instances of this class directly. Simply pass the classmethod *metric()* to a measurement function. Output using *dump()*. These are the only public methods. This is an abstract base class; you should use one of the concrete subclases in this module instead.

Each metric consumes one open file and 32K of memory while running. Output requires enough memory to load all data points for each metric.

Variables **dump_atexit** – automatically call *dump()* when the interpreter exits. Defaults to True.

classmethod metric(*name*, *count*, *elapsed*)
A metric function that buffers through numpy

Parameters

- **name** (*str*) – name of the metric
- **count** (*int*) – number of elements
- **elapsed** (*float*) – time in seconds

classmethod dump()
Output all recorded metrics

class instrument.numpy.**TableMetric**
Print a table of statistics

Variables **outfile** – output file. Defaults to sys.stderr.

class instrument.numpy.**PlotMetric**
Plot graphs of metrics.

Variables **outdir** – directory to save plots in. Defaults to ./mit_plots.

4.4 instrument.statsd

save metrics to statsd

instrument.statsd.**statsd_metric**(*name*, *count*, *elapsed*)
Metric that records to statsd/graphite

CHAPTER 5

Backwards Incompatibilities

5.1 0.4 -> 0.5

- main package renamed from *measure_it* to *instrument*
- prefixed *measure_iter*, etc. functions no longer available; use *instrument.iter* instead

5.2 0.3 -> 0.4

- remove deprecated `measure()`; use `measure_iter()` instead
- convert to package; *`statsd_metric()`* moved to its own module
- swap order of name & metric arguments

CHAPTER 6

Future Plans

Some rough thoughts.

6.1 Automagic Instrumentation

Support for automagic instrumentation of popular 3rd-party packages:

- django, using introspection logic from django-statsd
- generic WSGI middleware. Possibly flask.
- any dbapi-compatible database, with names derived by parsing SQL for table/query type
- HTTP clients: requests and urllib
- storage engines: MongoDB, memcached, redis, Elastic Search. Possibly sqlalchemy

6.2 More metric backends

- lightweight running stats, based on forthcoming stdlib statistics module. May include support for periodic stats output, as a low-budget alternative to statsd.
- Prometheus, Datadog, etc.

6.3 Bonus features

- sampling & filtering for metric functions
- integration of nice Jupyter notebook for analysis

6.4 Modernization

- rip out old 2.7 compatibility stuff
- pypy test & support

CHAPTER 7

Changelog

See also: *Backwards Incompatibilities*.

7.1 0.5.0

- rename project to *instrument* from *measure_it*
- update to modern tooling: pytest, pipenv, etc..
- improved testing: tox, travis
- fix to work with newer versions of statsd, including its django support

7.2 0.4

- add default_metric
- add make_multi_metric
- add TableMetric and PlotMetric, based on numpy
- add CSVDirMetric and CSVFileMetric
- measure_block supports user-supplied count

7.3 0.3

- add measure_first, measure_produce, measure_func, measure_block
- rename measure to measure_iter and deprecate old name

7.4 0.2

- add measure_reduce

7.5 0.1

Initial release

Python Module Index

i

Index