metrix

Burton DeWilde

Oct 17, 2020

API REFERENCE:

`metrix` is a Python library for tracking metrics through streams, with configurable tagging, batching, aggregating, and outputting of individual elements. It's designed for handling multiple metrics collected individually at high rates with shared output destinations, such as a log file and/or database, especially in the case that outputting metrics is only required or desired at lower rates.

CHAPTER

ONE

INSTALLATION

This package hasn't yet been published to PyPi (working on it...), but is still readily installable via `pip`:

```
$ python -m pip install git+https://github.com/bdewilde/metrix.git#egg=metrix
```

As usual, this will also install direct dependencies – `streamz` and `toolz` – as well as a few other packages needed for them to work.

CHAPTER

TWO

OVERVIEW

Users' entry into `metrix` is primarily through the *`MCoordinator`* class. It coordinates the flow of metric elements through one or multiple streams into one or multiple output destinations (called "sinks"), with an optional rate limit imposed before each sink to avoid any problematic pileups.

Configuration is left up to users, since it's entirely dependent on specific contexts and use cases. There are a few key considerations:

- the metrics to track: their names; any tags to apply to elements by default; how often to batch elements, either in time or number; and the aggregation(s) to be performed on batches of values
- what to do with the aggregated metrics: where results should go, and if they should be submitted at throttled rates

It's important to note that a single metric stream may output *multiple* aggregated values per batch, since each unique (name, aggregation, tags) combination is grouped together before sending them on to their destinations.

CHAPTER

THREE

EXAMPLE

Let's say we want to track the total number of articles published to a news site in 5-minute tumbling windows, as well as the total and average word counts per batch. We also want to tag articles by the section in which they're published, but this is only needed for the total published counts. Lastly, we'd like to log the aggregated results for diagnostic purposes.

How can we do this with `metrix`?

```
>>> import logging, statistics
>>> from metrix import MCoordinator, MStream, MSinkLogger
>>> # configure MC with metric streams and sinks
>>> mc = MCoordinator(
...     mstreams=[
...         MStream(
...             "n_articles",  # metric name
...             agg=sum,  # function to get the total number per batch
...             default_tags={"section": "NA"},  # default article tag, to avoid null␣
↪values
...             window_size=300,  # aggregate every 300 seconds (5 minutes)
...         ),
...         MStream(
...             "wc",  # metric name (shorthand for "word count")
...             agg=[sum, statistics.mean],  # functions to get total and average␣
↪values
...             window_size=300,  # same as n_articles, tho this isn't required
...         ),
...     ],
...     msinks=[
...         MSinkLogger(level=logging.INFO),  # log agg'd metrics at "info" level
...     ],
...     rate_limit=0.5,  # impose a half-second rate limit for [technical reason]
... )
>>> # fake articles data (but let's pretend)
>>> articles = [
...     {"text": "...", "section": "politics"},
...     {"text": "...", "section": "science"},
...     {"text": "..."},  # missing section!
...     ...
... ]
>>> # send metric elements for each article into corresponding streams via the MC
>>> for article in articles:
...     mc.send(
...         "n_articles",  # metric name
...         1,  # metric value, to be aggregated with other values
...         tags={"section": article["section"]} if article.get("section") else None,␣
↪ # optional tags
```

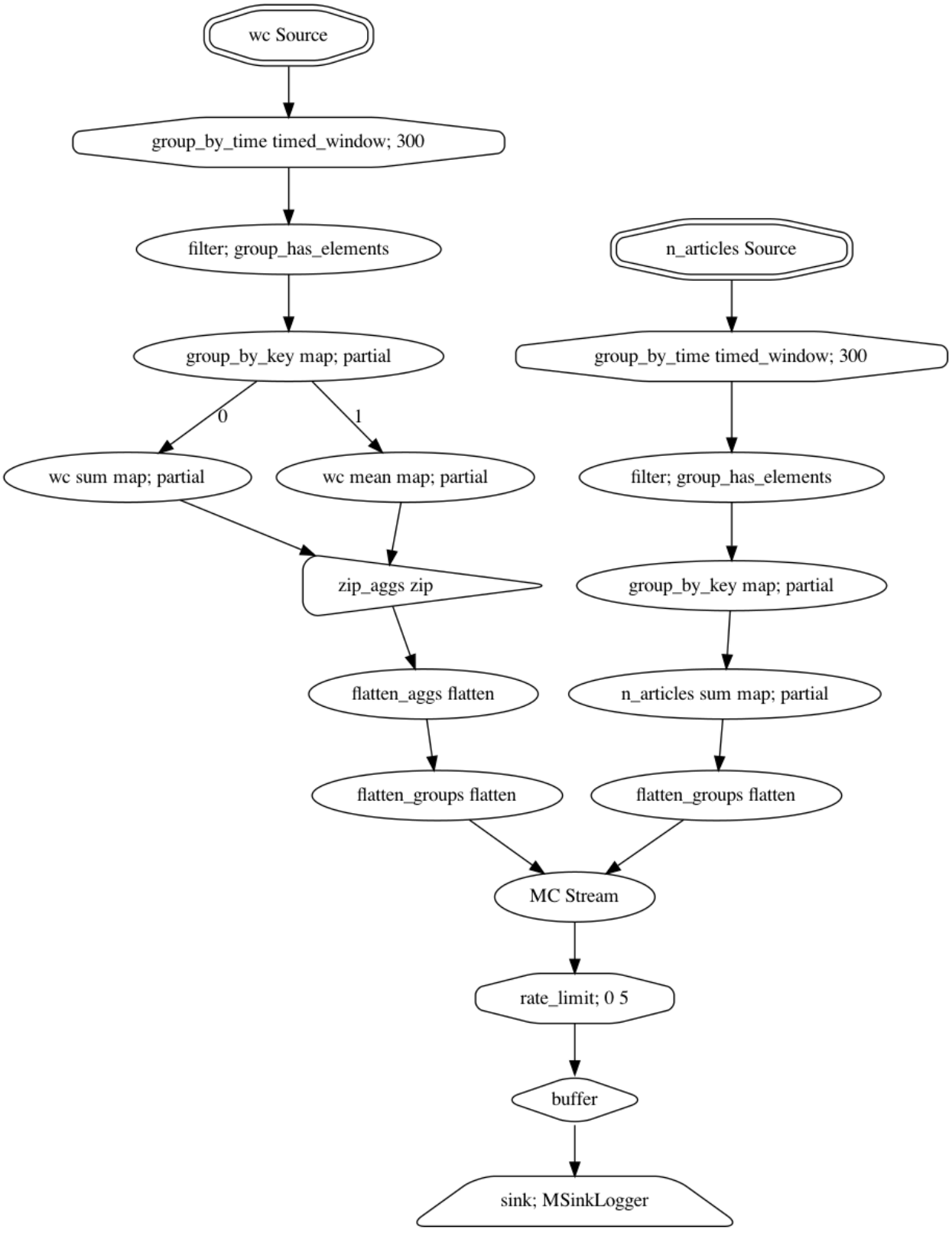
(continues on next page)

(continued from previous page)

```
...     )
...     mc.send("wc", len(article["text"].split()))
>>> # five minutes later... (fake aggregated metrics data, but again, let's pretend)
INFO:metrix.sinks:MElement(name='n_articles.sum', value=625, tags={'section':
↪'politics'})
INFO:metrix.sinks:MElement(name='n_articles.sum', value=290, tags={'section': 'science
↪'})
INFO:metrix.sinks:MElement(name='n_articles.sum', value=35, tags={'section': 'NA'})
INFO:metrix.sinks:MElement(name='wc.sum', value=161690, tags=None)
INFO:metrix.sinks:MElement(name='wc.mean', value=170.2, tags=None)
```

With the optional `networkx` and `graphviz` dependencies installed, we can easily visualize what the corresponding collection of metric streams and sinks looks like:

```
>>> mc.stream.visualize(filename=None)
```


wc Source
group_by_time timed_window; 300
filter; group_has_elements
group_by_key map; partial
0
1
wc sum map; partial
wc mean map; partial
zip_aggs zip
flatten_aggs flatten
flatten_groups flatten
n_articles Source
group_by_time timed_window; 300
filter; group_has_elements
group_by_key map; partial
n_articles sum map; partial
flatten_groups flatten
MC Stream
rate_limit; 0 5
buffer
sink; MSinkLogger

CHAPTER

FOUR

PERFORMANCE

`metrix` provides features and an API tailored to a particular use case – "metrics tracking through streams" – but under the hood, `streamz` and `toolz` do the heavy lifting. As such, this package's performance is largely dependent on theirs. According to the docs, `streamz` adds microsecond overhead to typical Python operations.

In practical terms, `metrix` can process up to about 7500 metric elements per second (not including any I/O costs associated with sinks, such as network connections or disk writes). This throughput doesn't vary much by the complexity of the input or output streams; it's dominated by the total number of messages sent in.

4.1 metric coordinator

class metrix.coordinator.**MCoordinator**(*, *mstreams: Optional[Sequence[metrix.stream.MStream]] = None*, *msinks: Optional[Sequence[metrix.sinks.MSink]] = None*, *rate_limit: Optional[Union[int, float, Sequence[Union[int, float]]]] = None*)

Class that coordinates the flow of metric elements through one or multiple streams into one or multiple sinks, with an optional rate limit imposed before the end.

Here are a few simple examples to illustrate key features:

```
>>> import statistics, time
>>> from metrix import MCoordinator, MStream, MSinkLogger, MSinkPrinter
>>> # one stream, one agg, one sink
>>> mc = MCoordinator(
...     mstreams=[MStream("n", agg=sum, batch_size=2)],
...     msinks=[MSinkPrinter()],
... )
>>> mc.send("n", 1)
>>> mc.send("n", 2)
MElement(name=n.sum, value=3, tags=None)
>>> # one stream, two aggs, two sinks
>>> mc = MCoordinator(
...     mstreams=[MStream("n", agg=[max, statistics.mean], batch_size=3)],
...     msinks=[MSinkPrinter(), MSinkLogger()],
... )
>>> mc.send("n", 1)
>>> mc.send("n", 2)
>>> mc.send("n", 3)
MElement(name=n.max, value=3, tags=None)
MElement(name=n.mean, value=2, tags=None)
INFO:metrix.sinks:MElement(name=n.max, value=3, tags=None)
INFO:metrix.sinks:MElement(name=n.mean, value=2, tags=None)
```

(continues on next page)

(continued from previous page)

```
>>> # two streams, default and element tags, and a timer
>>> mc = MCoordinator(
...     mstreams=[
...         MStream("n", agg=sum, batch_size=3, default_tags={"foo": "bar"}),
...         MStream("time", agg={"avg": statistics.mean}, window_size=1),
...     ],
...     msinks=[MSinkPrinter()],
... )
>>> mc.send("n", 1)
>>> mc.send("n", 1)
>>> mc.send("n", 1, tags={"foo": "BAR!"})
MElement(name=n.sum, value=2, tags={'foo': 'bar'})
MElement(name=n.sum, value=1, tags={'foo': 'BAR!'})
>>> with mc.timer("time", scale=1):
...     time.sleep(0.5)
>>> with mc.timer("time", scale=1):
...     time.sleep(0.75)
>>> with mc.timer("time", scale=1):
...     time.sleep(0.5)
>>> with mc.timer("time", scale=1):
...     time.sleep(0.25)
MElement(name=time.avg, value=0.5028860028833151, tags=None)
MElement(name=time.avg, value=0.7517436337657273, tags=None)
MElement(name=time.avg, value=0.377787658944726, tags=None)
```

In typical production usage, you'll be tracking a few metrics and periodically logging and/or sending aggregated values to TSDB. Here's how that might look:

```
>>> from metrics import MSinkTSDB
>>> mc = MCoordinator(
...     mstreams=[
...         MStream("n_msgs", agg=sum, window_size=3),
...         MStream("msg_len", agg=[statistics.mean, statistics.stdev], window_
↪size=5)
...     ],
...     msinks=[MSinkLogger(), MSinkTSDB(<TSDB_CLIENT>)],
...     rate_limit=[0, 1.0],
... )
>>> msgs = list(range(10))  # fake data ;)
>>> for msg in msgs:
...     mc.send("n_msgs", 1)
...     mc.send("msg_len", msg)
INFO:metrix.sinks:MElement(name=n_msgs.sum, value=10, tags=None)
INFO:metrix.sinks:MElement(name=msg_len.mean, value=4.5, tags=None)
INFO:metrix.sinks:MElement(name=msg_len.stdev, value=3.0276503540974917,␣
↪tags=None)
```

Parameters

- **mstreams** – One or more *MStream* s through which metric elements are sent. Typically provided on init, but may also be passed individually via *MCoordinator.add_mstream()*.
- **msinks** – One or more *MSink* s to which metric elements are sent. Typically provided on init, but may also be passed individually via *MCoordinator.add_msink()*. In a development context, the simple `MSinkPrinter` will give visibility into the outputs of metric streams, but in a production, you'll want to specify more persistent metric sinks like

`MSinkLogger` and `MSinkTSDB`.

- **`rate_limit`** – Optional rate limit that prevents two metric elements from streaming into a sink in an interval shorter than `rate_limit` seconds. If a single number, this is applied to all `msinks`; if a sequence of numbers with the same length as `msinks`, limits will be applied element- wise to the corresponding metric sinks.

  For example: `rate_limit=1.5` causes elements to be sent on to each sink in `msinks` at least 1.5 seconds apart. If `rate_limit=[1.5, 0.5]` (and two sinks are specified), then the first sink will have a 1.5-second rate limit while the second will have a 0.5-second rate limit applied.

Warning: If `MSinkTSDB` is added as a sink, be sure to have `rate_limit` set to at least 1.0 seconds to prevent data loss, since OpenTSDB doesn't support sub-second data. (Yes, I know – it's bonkers.)

Given this constraint, you must also be mindful of the total number of unique metric (name, agg, tags) pairs passing through `mstreams` per second to ensure that the output sinks can keep up with the rate of input metrics. For example, if 2 streams use a single aggregator with default `window_size=10` and `rate_limit=1.0`, then you should limit yourself to no more than 5 distinct tag sets per metric. If you have more tags or aggs, increase your window size accordingly! Here's a useful formula:

```
sum((num_aggs * num_unique_tag_sets / window_size) for stream in mstreams) =␣
↪num_total_metrics_per_sec
```

If `num_total_metrics_per_sec > rate_limit`, you have a problem.

`stream`
Base metric coordinator stream, to which metric streams connect and from which metric sinks extend.

`metric_mstreams`
Mapping of metric name to metric stream, each of which is upstream from and connected to *`MCoordinator.stream`*.

`msinks`
Sequence of metric sinks, each of which is downstream from and connected to *`MCoordinator.stream`*.

`add_mstream`(*mstream:* metrix.stream.MStream) → None
Add a metric stream to this coordinator by connecting it to all sink streams and making it accessible by name via *`MCoordinator.metric_mstreams`*.

`add_msink`(*msink:* metrix.sinks.MSink, *rate_limit: Optional[Union[int, float]] = None*) → None
Add a metric sink to this coordinator by branching off *`MCoordinator.stream`* with a buffered, optionally rate-limited stream that ends in `msink`.

`send`(*name: str*, *value: Union[int, float]*, *, *tags: Optional[Dict] = None*) → None
Send a metric value to a particular metric stream; optionally, pass tags to add new and overwrite existing default tags associated with the stream.

Parameters

- **`name`** – Metric name.
- **`value`** – Numeric metric value.
- **`tags`** – Optional tags to associate with this specific metric `value`.

See also:

`MStream.send()`

timer(*name: str*, *scale: int = 1*, ***, *tags: Optional[Dict] = None*)

Get a context manager for a particular stream that measures the elapsed time spent running statements enclosed by the `with` statement, and sends that time to the stream.

Parameters

- **scale** – Multiplier applied to the elapsed time value, in seconds by default. For example, to report time in milliseconds, use `scale=1000`.
- **tags** – Optional tags to associate with this specific timer value.

See also:

MStream.timer()

4.2 metric stream

class metrix.stream.**MStream**(*name: str*, ***, *agg: Union[Callable[[Iterable[Union[int, float]]], Union[int, float]], Sequence[Callable[[Iterable[Union[int, float]]], Union[int, float]]], Mapping[str, Callable[[Iterable[Union[int, float]]], Union[int, float]]]]*, *default_tags: Optional[Dict] = None*, *window_size: Optional[int] = None*, *batch_size: Optional[int] = None*)

A stream of *MElement* s that groups elements into batches of fixed time or number, further groups batches by distinct assigned tags, then aggregates each group's values by one or multiple functions.

To do any useful work, metric streams must be connected to a *MSink*, which operates on elements in a visible / persistent way. In typical usage, you'll want to connect multiple streams to multiple sinks using a centralized coordinator: *MCoordinator*.

```
>>> from metrix import MElement, MStream
>>> eles = [{"value": 1}, {"value": 2}, {"value": 1, "tags": {"foo": "bar"}}]
>>> mstream = MStream("m", agg=sum, default_tags={"foo": "BAR!"}, window_size=1)
>>> # HACK! we'll add a sink directly so we can see what happens
>>> mstream.stream.sink(print)
>>> for ele in eles:
...     mstream.send(**ele)
MElement(name=m.sum, value=3, tags={'foo': 'BAR!'})
MElement(name=m.sum, value=1, tags={'foo': 'bar'})
```

Parameters

- **name** – Name of the metric whose elements are sent into this stream.
- **agg** – One or multiple aggregation functions to be applied to groups of metric elements' values in order to produce new, aggregated metric elements. This may be specified as a single callable or a sequence of callables, in which case the corresponding components of the *MStream.stream* are named after the functions themselves; this may also be specified as a mapping of component name to callable, in which case the user-specified names are used instead.
- **default_tags** – Optional set of tags to apply to all metric elements by default. Tags specified on individual elements override and append to this def
- **window_size** – Size of tumbling window in *seconds* with which to group elements. For example: If `window_size=10`, all elements sent into the stream within a given 10-second window will be grouped together before their values are aggregated, as specified by `agg`.

- **batch_size** – Size of batches in *number of elements* with which to group elements. For example: If `batch_size=10`, every 10 successive elements sent into the stream will be grouped together before their values are aggregated, as specified by `agg`. Note that setting `batch_size=1` will effectively skip grouping, in which case aggregating values doesn't make sense, either.

Note: You *must* set either `window_size` or `batch_size` when initializing a metric stream. No default is set because it depends entirely on context: the rate with which metric elements are sent into the stream, the desired resolution on aggregated metrics, and any rate limit requirements on connected metric sinks. This is the only stream attribute that demands deliberate thought. Choose wisely! :)

---

name

agg

default_tags

window_size

batch_size

source
Entry point to the metric stream.

stream
Data processing stream to which metric elements are sent.

send(*value: Union[int, float]*, *, *tags: Optional[Dict] = None*) → None
Send a given metric value to the stream; optionally, pass metric-specific tags to add new and overwrite existing default tags associated with the stream.

Parameters

- **value** – Numeric metric value.
- **tags** – Optional tags to associate with this specific metric `value`.

timer(*scale: int = 1*, *, *tags: Optional[Dict] = None*)
Context manager that measures the elapsed time spent running statements enclosed by the `with` statement, and sends that time to the stream.

Parameters

- **scale** – Multiplier applied to the elapsed time value, in seconds by default. For example, to report time in milliseconds, use `scale=1000`.
- **tags** – Optional tags to associate with this specific timer value.

See also:

MStream.send()

metrix.stream.**group_has_elements**(*group: Sequence[*metrix.element.MElement*]*) → bool
Return True if `group` contains any metric elements, and False otherwise; used to filter out empty group from a metric stream.

4.3 metric sinks

class metrix.sinks.**MSink**
Base class for subclasses that are called on a *MElement* and perform some useful action on it.

class metrix.sinks.**MSinkPrinter**
Class that's called on a *MElement* and prints it to stdout. That's it! This class is useful in development when experimenting with *MCoordinator* so users can see stream contents, but is not suitable for production.

```
>>> from metrix import MElement, MSinkPrinter
>>> msink = MSinkPrinter()
>>> msink(MElement("foo", 1))
MElement(name=foo, value=1, tags=None)
```

class metrix.sinks.**MSinkLogger**(*name: str = 'metrix.sinks'*, *level: int = 20*, *msg_fmt_str: str = '%s'*)
Class that's called on a *MElement* and logs it, as-is.

```
>>> from metrix import MElement, MSinkLogger
>>> me = MElement("foo", 1)
>>> msink = MSinkLogger()
>>> msink(me)
INFO:metrix.sinks:MElement(name=foo, value=1, tags=None)
>>>  msink = MSinkLogger(name="my-logger", level=30, msg_fmt_str="[metric] %s")
WARNING:my-logger:[metric] MElement(name=foo, value=1, tags=None)
```

Parameters

- **name** – Name of the logger to use when logging metric elements.
- **level** – Level at which metric elements are logged.
- **msg_fmt_str** – Message format string into which metric elements are merged using a string formatting operator. Must contain exactly one "%s" for a given *MElement*; may contain any other hard-coded text you wish.

logger
logging.Logger

level
int

msg_fmt_str
str

class metrix.sinks.**MSinkTSDB**(*tsdb_client*)
Class that's called on a *MElement* and sends its data to OpenTSDB via an instantiated TSDB client.

Parameters **tsdb_client** – Instantiated TSDB client with a `send` method, such as `potsdb.Client`.

tsdb_client

Note: It's the user's responsibility to ensure that a suitable TSDB client library is available in the environment where this class is instantiated.

4.4 metric element

class metrix.element.**MElement**(*name: str*, *value: Union[int, float]*, **, tags: Optional[Dict] = None*)

An individual metric element – a single data point – to be sent through a stream and on to one or more sinks.

```
>>> from metrix import MElement
>>> me = MElement("counts", 1, tags={"env": "dev", "foo": "bar"})
>>> print(me)
MElement(name='counts', value=1, tags={'env': 'dev', 'foo': 'bar'})
>>> me.key
'env:dev|foo:bar'
```

Parameters

- **name** – Base name of the metric to which the element belongs.
- **value** – Numeric value of the metric element.
- **tags** – Optional tags to associate with this (name, value) pair.

Note: In typical usage, users will not directly instantiate this class; instead, they'll pass (name, value, tags) into `MCoordinator.send()` or `MCoordinator.timer()`, which will create a corresponding `MElement` under the hood.

metrix.element.**key_from_tags**(*tags: Optional[Dict]*) → Optional[str]

Generate a (hashable!) key string from a collection of `tags`, where tags are pipe-delimited and each tag's field and value are colon-delimited.

```
>>> key_from_tags({"foo": "bar})
"foo:bar"
>>> key_from_tags({"foo": "bar", "bat": "baz"})
"bat:baz|foo:bar"
```

Note: The ordering of items in `tags` doesn't matter, since the generated key is always ordered alphabetically.

metrix.element.**tags_from_key**(*key: Optional[str]*) → Optional[Dict]

Generate a collection of tags from a key string, where tags are pipe-delimited and each tag's field and value are colon-delimited.

```
>>> tags_from_key(None)
None
>>> tags_from_key("foo:bar")
{"foo": "bar"}
>>> tags_from_key("foo:bar|bat:baz")
{"bat": "baz", "foo": "bar"}
```

Note: The ordering of items in `key` doesn't matter, since the generated tags items are always ordered alphabetically.

See also:

key_from_tags()

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX