aioinflux Documentation

Release 0.4.1

Gustavo Bezerra

Jan 18, 2019

Contents:

Asynchronous Python client for InfluxDB. Built on top of aiohttp and asyncio. Aioinflux is an alternative to the official InfluxDB Python client.

Aioinflux supports interacting with InfluxDB in a non-blocking way by using aiohttp. It also supports writing and querying of Pandas dataframes, among other handy functionality.

CHAPTER 1

Installation

To install the latest release:

```
$ pip install aioinflux
$ pip install aioinflux[pandas]  # For DataFrame parsing support
```

The library is still in beta, so you may also want to install the latest version from the development branch:

```
$ pip install git+https://github.com/plugaai/aioinflux@dev
```

1.1 Dependencies

Aioinflux supports Python 3.6+ **ONLY**. For older Python versions please use the official Python client. However, there is some discussion regarding Pypy/Python 3.5 support.

The main third-party library dependency is `aiohttp`, for all HTTP request handling. and `pandas` for `DataFrame` reading/writing support.

There are currently no plans to support other HTTP libraries besides `aiohttp`. If `aiohttp` + `asyncio` is not your soup, see alternatives.

CHAPTER 2

User Guide

2.1 TL;DR

This sums most of what you can do with `aioinflux`:

```
import asyncio
from aioinflux import InfluxDBClient

point = {
    'time': '2009-11-10T23:00:00Z',
    'measurement': 'cpu_load_short',
    'tags': {'host': 'server01',
             'region': 'us-west'},
    'fields': {'value': 0.64}
}

async def main():
    async with InfluxDBClient(db='testdb') as client:
        await client.create_database(db='testdb')
        await client.write(point)
        resp = await client.query('SELECT value FROM cpu_load_short')
        print(resp)


asyncio.get_event_loop().run_until_complete(main())
```

2.2 Client modes

Despite the library's name, *`InfluxDBClient`* can also run in non-async mode (a.k.a `blocking`) mode. It can be useful for debugging and exploratory data analysis.

The running mode for can be switched on-the-fly by changing the `mode` attribute:

```
client = InfluxDBClient(mode='blocking')
client.mode = 'async'
```

The `blocking` mode is implemented through a decorator that automatically runs coroutines on the event loop as soon as they are generated. Usage is almost the same as in the `async` mode, but without the need of using `await` and being able to run from outside of a coroutine function:

```
client = InfluxDBClient(db='testdb', mode='blocking')
client.ping()
client.write(point)
client.query('SELECT value FROM cpu_load_short')
```

2.3 Writing data

Input data can be:

1. A string (`str` or `bytes`) properly formatted in InfluxDB's line protocol
2. A mapping (e.g. `dict`) containing the following keys: `measurement`, `time`, `tags`, `fields`
3. A Pandas `DataFrame` with a `DatetimeIndex`
4. A *`DataPoint()`* object (see *below*)
5. An iterable of one of the above

Input data in formats 2-4 are serialized into the line protocol before being written to InfluxDB. `str` or `bytes` are assumed to already be in line protocol format and are inserted into InfluxDB as they are. All serialization from JSON (InfluxDB's only output format) and parsing to line protocol (InfluxDB's only input format) functionality is located in the *`serialization`* subpackage.

Beware that serialization is not highly optimized (C extensions / cythonization PRs are welcome!) and may become a bottleneck depending on your application's performance requirements. It is, however, reasonably faster than InfluxDB's official Python client.

The `write` method returns `True` when successful and raises an `InfluxDBError` otherwise.

2.3.1 Writing dictionary-like objects

Aioinflux accepts any dictionary-like object (mapping) as input. However, that dictionary must be properly formatted and contain the following keys:

1) **measurement**: Optional. Must be a string-like object. If omitted, must be specified when calling *`write()`* by passing a `measurement` argument.

2) **time**: Optional. The value can be `datetime.datetime`, date-like string (e.g., `2017-01-01`, `2009-11-10T23:00:00Z`) or anything else that can be parsed by `pandas.Timestamp`. See the Pandas documentation for details. If Pandas is not available, `ciso8601` is used instead for string parsing.

3) **tags**: Optional. This must contain another mapping of field names and values. Both tag keys and values should be strings.

4) **fields**: Mandatory. This must contain another mapping of field names and values. Field keys should be strings. Field values can be `float`, `int`, `str`, `bool` or `None` or any its subclasses. Attempting to use Numpy types will cause errors as `np.int64`, `np.float64`, etc are not subclasses of Python's builti-in numeric types. Use dataframes for writing data using Numpy types.

Any fields other then the above will be ignored when writing data to InfluxDB.

A typical dictionary-like point would look something like the following:

```
{'time': '2009-11-10T23:00:00Z',
'measurement': 'cpu_load_short',
'tags': {'host': 'server01', 'region': 'us-west'},
'fields': {'value1': 0.64, 'value2': True, 'value3': 10}}
```

Note: Timestamps and timezones

Working with timezones in computing tends to be quite messy. To avoid such problems, the broadly agreed upon idea is to store timestamps in UTC. This is how both InfluxDB and Pandas treat timestamps internally.

Pandas and many other libraries also assume all input timestamps are in UTC unless otherwise explicitly noted. Aioinflux does the same and assumes any timezone-unaware `datetime` object or datetime-like strings is in UTC. Aioinflux does not raise any warnings when timezone-unaware input is passed and silently assumes it to be in UTC.

2.3.2 Writing DataFrames

Aioinflux also accepts Pandas dataframes as input. The only requirements for the dataframe is that the index **must** be of type `DatetimeIndex`. Also, any column whose `dtype` is `object` will be converted to a string representation.

A typical dataframe input should look something like the following:

```
                                          LUY       BEM       AJW tag
2017-06-24 08:45:17.929097+00:00  2.545409  5.173134  5.532397   B
2017-06-24 10:15:17.929097+00:00 -0.306673 -1.132941 -2.130625   E
2017-06-24 11:45:17.929097+00:00  0.894738 -0.561979 -1.487940   B
2017-06-24 13:15:17.929097+00:00 -1.799512 -1.722805 -2.308823   D
2017-06-24 14:45:17.929097+00:00  0.390137 -0.016709 -0.667895   E
```

The measurement name must be specified with the `measurement` argument when calling *`write()`*. Columns that should be treated as tags must be specified by passing a sequence as the `tag_columns` argument. Additional tags (not present in the actual dataframe) can also be passed using arbitrary keyword arguments.

Example:

```
client = InfluxDBClient(db='testdb', mode='blocking')
client.write(df, measurement='prices', tag_columns=['tag'], asset_class='equities')
```

In the example above, `df` is the dataframe we are trying to write to InfluxDB and `measurement` is the measurement we are writing to.

`tag_columns` is in an optional iterable telling which of the dataframe columns should be parsed as tag values. If `tag_columns` is not explicitly passed, all columns in the dataframe whose dtype is not `DatetimeIndex` will be treated as InfluxDB field values.

Any other keyword arguments passed to *`write()`* are treated as extra tags which will be attached to the data being written to InfluxDB. Any string which is a valid InfluxDB identifier and valid Python identifier can be used as an extra tag key (with the exception of the strings `data`, `measurement` and `tag_columns`).

See API reference for details.

2.3.3 Writing DataPoint objects

New in version 0.4.0.

DataPoint are namedtuple-like objects that provide fast line protocol serialization by defining a schema.

A *DataPoint* class can be defined using the *datapoint* class factory function with some special types annotations:

```
from aioinflux.serialization import datapoint, InfluxType

@datapoint
class Trade:
    timestamp: InfluxType.TIMEINT
    instrument: InfluxType.TAGENUM
    source: InfluxType.TAG
    side: InfluxType.TAG
    price: InfluxType.FLOAT
    size: InfluxType.INT
    trade_id: InfluxType.STR
```

Alternatively, it can also be defined functionally:

```
Trade = datapoint(dict(
   timestamp=InfluxType.TIMEINT,
   instrument=InfluxType.TAG,
   source=InfluxType.TAG,
   side=InfluxType.TAG,
   price=InfluxType.FLOAT,
   size=InfluxType.INT,
   trade_id=InfluxType.STR,
), name='Trade')
```

The class can then be be instantiated by positional or keyword arguments:

```
# Positional
trade = Trade(1540184368785116000, 'APPL', 'NASDAQ', 'BUY',
              219.23, 100, '34a1e085-3122-429c-9662-7ce82039d287')

# Keyword
trade = Trade(
   timestamp=1540184368785116000,
   instrument='AAPL',
   source='NASDAQ',
   side='BUY',
   price=219.23,
   size=100,
   trade_id='34a1e085-3122-429c-9662-7ce82039d287'
)
```

Attributes can be accessed by dot notation (`__getattr__`) or dictionary-like notation (`__getitem__`). Iteration is also supported:

```
trade.price  # 219.23
trade['price']  # 219.23
list(trade)  # ['timestamp', 'source', 'instrument', 'size', 'price', 'trade_id',
↪'side']
list(trade.items()  # [('timestamp', 1540184368785116000), ('source', 'APPL'), (
↪'instrument', 'NASDAQ'), ('size', 'BUY'), ('price', 219.23), ('trade_id', 100), (
↪'side', '34a1e085-3122-429c-9662-7ce82039d287')]
```

Every DataPoint object has a *to_lineprotocol()* method which generates a line protocol representation of the datapoint:

```
trade.to_lineprotocol()
# b'Trade,source=APPL,instrument=NASDAQ size=BUYi,price=219.23,trade_id="100",side=
↪"34a1e085-3122-429c-9662-7ce82039d287" 1540184368785116000'
```

write() can write DataPoint objects (or iterables of DataPoint objects) to InfluxDB (by using *to_lineprotocol()* internally):

```
client = InfluxDBClient()
await client.write(trade)
```

Every class generated by *datapoint* has *DataPoint* as its base class:

```
isintance(trade, DataPoint)  # True
```

DataPoint Types

Note: In this section, the word "types" refers to members of the *InfluxType* enum

DataPoint types are defined using the *InfluxType* enum. All type annotations MUST be a *InfluxType* member. The types available are based on the native types of InfluxDB (see the InfluxDB docs for details), with some extra types to help the serialization to line protocol and/or allow more flexible usage (such as the use of `Enum` objects).

Datapoint type	Description
`MEASUREMENT`	Optional. If missing, the measurement becomes the class name
`TIMEINT`	Timestamp is a nanosecond UNIX timestamp
`TIMESTR`	Timestamp is a datetime string (somewhat compliant to ISO 8601)
`TIMEDT`	Timestamp is a `datetime` (or subclasses such as `pandas.Timestamp`)
`TAG`	Treats field as an InfluxDB tag
`TAGENUM`	Same as `TAG` but allows the use of `Enum`
`PLACEHOLDER`	Boolean field which is always true and NOT present in the class constructor. Workaround for creating field-less points (which is not supported natively by InfluxDB)
`BOOL`	Boolean field
`INT`	Integer field
`FLOAT`	Float field
`STR`	String field
`ENUM`	Same as `STR` but allows the use of `Enum`

`TAG*` types are optional. One and only one `TIME*` type must present. At least ONE field type be present.

DataPoint options

The *datapoint()* function/decorator provides some options to customize object instantiation/serialization. See the API reference for details.

Advantages compared to dictionary-like objects

- Faster (see below)
- Explicit field names: better IDE support
- Explicit types: avoids types errors when writing to InfluxDB (e.g.: `float` field getting parsed as a `float`)
- Optional `None` support
- No need to use nested data structures

Performance

Serialization using *`DataPoint`* is about 3x faster than dictionary-like objects. See this notebook and the API reference for details. Regarding object instantiation performance, dictionaries are slightly faster, but the time difference is negligible and 1-2 orders of magnitude smaller than time required for serialization.

2.4 Querying data

Querying data is as simple as passing an InfluxDB query string to *`query()`*:

```
client.query('SELECT myfield FROM mymeasurement')
```

The result (in `blocking` and `async` modes) is a dictionary containing the parsed JSON data returned by the InfluxDB HTTP API:

```
{'results': [{'series': [{'columns': ['time', 'Price', 'Volume'],
     'name': 'mymeasurement',
     'values': [[1491963424224703000, 5783, 100],
      [1491963424375146000, 5783, 200],
      [1491963428374895000, 5783, 100],
      [1491963429645478000, 5783, 1100],
      [1491963429655289000, 5783, 100],
      [1491963437084443000, 5783, 100],
      [1491963442274656000, 5783, 900],
      [1491963442274657000, 5782, 5500],
      [1491963442274658000, 5781, 3200],
      [1491963442314710000, 5782, 100]]}],
   'statement_id': 0}]}
```

2.4.1 Output formats

When querying data, `InfluxDBClient` can return data in one of the following formats:

1) `json`: Default. Returns the a dictionary containing the JSON response received from InfluxDB.
2) `bytes`: Returns raw, non-parsed JSON binary blob as received from InfluxDB. The contents of the returns JSON blob are not checked at all. Useful for response caching.
3) `dataframe`: Parses the result into a Pandas dataframe or a dictionary of dataframes. See *Retrieving DataFrames* for details.
4) `iterable`: Wraps the JSON response in a `InfluxDBResult` or `InfluxDBChunkedResult` object. This object main purpose is to facilitate iteration of data. See *Iterating responses* for details.

The output format for can be switched on-the-fly by changing the `output` attribute:

```
client = InfluxDBClient(output='dataframe')
client.mode = 'json'
```

2.4.2 Retrieving DataFrames

When the client is in `dataframe` mode, *`query()`* will return a `pandas.DataFrame`:

```
                                  Price  Volume
2017-04-12 02:17:04.224703+00:00   5783     100
2017-04-12 02:17:04.375146+00:00   5783     200
2017-04-12 02:17:08.374895+00:00   5783     100
2017-04-12 02:17:09.645478+00:00   5783    1100
2017-04-12 02:17:09.655289+00:00   5783     100
2017-04-12 02:17:17.084443+00:00   5783     100
2017-04-12 02:17:22.274656+00:00   5783     900
2017-04-12 02:17:22.274657+00:00   5782    5500
2017-04-12 02:17:22.274658+00:00   5781    3200
2017-04-12 02:17:22.314710+00:00   5782     100
```

Note: On multi-statement queries and/or statements that return multiple InfluxDB series (such as a *GROUP by "tag"* query), a dictionary of dataframes or a list of dictionaries of dataframes may be returned. Aioinflux generates a dataframe for each series contained in the JSON returned by InfluxDB. See this Github issue for further discussion.

When generating dataframes, InfluxDB types are mapped to the following Numpy/Pandas dtypes:

InfluxDB type	Dataframe column `dtype`
Float	`float64`
Integer	`int64`
String	`object`
Boolean	`bool`
Timestamp	`datetime64`

2.4.3 Chunked responses

Aioinflux supports InfluxDB chunked queries. Passing `chunked=True` when calling *`query()`*, returns an `AsyncGenerator` object, which can asynchronously iterated. Using chunked requests allows response processing to be partially done before the full response is retrieved, reducing overall query time.

```
chunks = await client.query("SELECT * FROM mymeasurement", chunked=True)
async for chunk in chunks:
    # do something
    await process_chunk(...)
```

Chunked responses are not supported when using the `dataframe` output format.

2.4.4 Iterating responses

By default, *`query()`* returns a parsed JSON response from InfluxDB. In order to easily iterate over that JSON response point by point, Aioinflux provides the `iterpoints` function, which returns a generator object:

```
from aioinflux import iterpoints

r = client.query('SELECT * from h2o_quality LIMIT 10')
for i in iterpoints(r):
    print(i)
```

```
[1439856000000000000, 41, 'coyote_creek', '1']
[1439856000000000000, 99, 'santa_monica', '2']
[1439856360000000000, 11, 'coyote_creek', '3']
[1439856360000000000, 56, 'santa_monica', '2']
[1439856720000000000, 65, 'santa_monica', '3']
```

`iterpoints` can also be used with chunked responses:

```
chunks = await client.query('SELECT * from h2o_quality', chunked=True)
async for chunk in chunks:
    for point in iterpoints(chunk):
        # do something
```

By default, the generator returned by `iterpoints` yields a plain list of values without doing any expensive parsing. However, in case a specific format is needed, an optional `parser` argument can be passed. `parser` is a function that takes the raw value list for each data point and an additional metadata dictionary containing all or a subset of the following: `{'columns', 'name', 'tags', 'statement_id'}`.

```
r = await client.query('SELECT * from h2o_quality LIMIT 5')
for i in iterpoints(r, lambda x, meta: dict(zip(meta['columns'], x))):
    print(i)
```

```
{'time': 1439856000000000000, 'index': 41, 'location': 'coyote_creek', 'randtag': '1'}
{'time': 1439856000000000000, 'index': 99, 'location': 'santa_monica', 'randtag': '2'}
{'time': 1439856360000000000, 'index': 11, 'location': 'coyote_creek', 'randtag': '3'}
{'time': 1439856360000000000, 'index': 56, 'location': 'santa_monica', 'randtag': '2'}
{'time': 1439856720000000000, 'index': 65, 'location': 'santa_monica', 'randtag': '3'}
```

Besides being explicitly with a raw response, `iterpoints` is also be used "automatically" by `InfluxDBResult` and `InfluxDBChunkedResult` when using `iterable` mode:

```
client.output = 'iterable'
# Returns InfluxDBResult object
r = client.query('SELECT * from h2o_quality LIMIT 10')
for i in r:
    # do something

# Returns InfluxDBChunkedResult object
r = await client.query('SELECT * from h2o_quality', chunked=True)
async for i in r:
    # do something

# Returns InfluxDBChunkedResult object
r = await client.query('SELECT * from h2o_quality', chunked=True)
async for chunk in r.iterchunks():
    # do something with JSON chunk
```

2.4.5 Query patterns

Aioinflux provides a wrapping mechanism around `InfluxDBClient.query` in order to provide convenient access to commonly used query patterns.

Query patterns are query strings containing optional named "replacement fields" surrounded by curly braces `{}`, just as in `str_format()`. Replacement field values are defined by keyword arguments when calling the method associated with the query pattern. Differently from plain `str_format()`, positional arguments are also supported and can be mixed with keyword arguments.

Aioinflux built-in query patterns are defined here. Users can also dynamically define additional query patterns by using the `InfluxDBClient.set_query_pattern` helper function. User-defined query patterns have the disadvantage of not being shown for auto-completion in IDEs such as Pycharm. However, they do show up in dynamic environments such as Jupyter. If you have a query pattern that you think will used by many people and should be built-in, please submit a PR.

Built-in query pattern examples:

```
client.create_database(db='foo')    # CREATE DATABASE {db}
client.drop_measurement('bar')      # DROP MEASUREMENT {measurement}'
client.show_users()                 # SHOW USERS

# Positional and keyword arguments can be mixed
client.show_tag_values_from('bar', key='spam')  # SHOW TAG VALUES FROM {measurement}␣
↪WITH key = "{key}"
```

Please refer to InfluxDB documentation for further query-related information.

2.5 Other functionality

2.5.1 Authentication

Aioinflux supports basic HTTP authentication provided by `aiohttp.BasicAuth`. Simply pass `username` and `password` when instantiating `InfluxDBClient`:

```
client = InfluxDBClient(username='user', password='pass)
```

2.5.2 Unix domain sockets

If your InfluxDB server uses UNIX domain sockets you can use `unix_socket` when instantiating `InfluxDBClient`:

```
client = InfluxDBClient(unix_socket='/path/to/socket')
```

See `aiohttp.UnixConnector` for details.

2.5.3 HTTPS/SSL

Aioinflux/InfluxDB uses HTTP by default, but HTTPS can be used by passing `ssl=True` when instantiating `InfluxDBClient`. If you are acessing your your InfluxDB instance over the public internet, setting up HTTPS is strongly recommended.

```
client = InfluxDBClient(host='my.host.io', ssl=True)
```

2.5.4 Database selection

After the instantiation of the `InfluxDBClient` object, database can be switched by changing the `db` attribute:

```
client = InfluxDBClient(db='db1')
client.db = 'db2'
```

Beware that differently from some NoSQL databases (such as MongoDB), InfluxDB requires that a databases is explicitly created (by using the `CREATE DATABASE` query) before doing any operations on it.

2.5.5 Debugging

If you are having problems while using Aioinflux, enabling logging might be useful.

Below is a simple way to setup logging from your application:

```
import logging

logging.basicConfig()
logging.getLogger('aioinflux').setLevel(logging.DEBUG)
```

For further information about logging, please refer to the official documentation.

CHAPTER 3

Implementation details

Since InfluxDB exposes all its functionality through an HTTP API, `InfluxDBClient` tries to be nothing more than a thin and simple wrapper around that API.

The InfluxDB HTTP API exposes exactly three endpoints/functions: `ping`, `write` and `query`.

`InfluxDBClient` merely wraps these three functions and provides some parsing functionality for generating line protocol data (when writing) and parsing JSON responses (when querying).

Additionally, partials are used in order to provide convenient access to commonly used query patterns. See the *Query patterns* section for details.

CHAPTER 4

API Reference

This part of the documentation covers all the interfaces of Aioinflux

Note: This section of the documentation is under writing and may be wrong/incomplete

4.1 Client Interface

class aioinflux.client.**InfluxDBClient**(*host='localhost'*, *port=8086*, *mode='async'*, *output='json'*, *db=None*, ***, *ssl=False*, *unix_socket=None*, *username=None*, *password=None*, *database=None*, *loop=None*)

ping()
Pings InfluxDB. Returns a dictionary containing the headers of the response from *influxd*.

Return type dict

query(*q*, **args*, *epoch='ns'*, *chunked=False*, *chunk_size=None*, *db=None*, *parser=None*, ***kwargs*)
Sends a query to InfluxDB. Please refer to the InfluxDB documentation for all the possible queries: https://docs.influxdata.com/influxdb/latest/query_language/

Parameters

- **q** (AnyStr) – Raw query string
- **args** – Positional arguments for query patterns
- **db** (Optional[str]) – Database to be queried. Defaults to *self.db*.
- **epoch** (str) – Precision level of response timestamps. Valid values: {'ns', 'u', 'µ', 'ms', 's', 'm', 'h'}.

- **chunked** (bool) – If True, makes InfluxDB return results in streamed batches rather than as a single response. Returns an AsyncGenerator which yields responses in the same format as non-chunked queries.
- **chunk_size** (Optional[int]) – Max number of points for each chunk. By default, InfluxDB chunks responses by series or by every 10,000 points, whichever occurs first.
- **kwargs** – Keyword arguments for query patterns
- **parser** (Optional[Callable]) – Optional parser function for 'iterable' mode

Return type Union[Asyncgenerator[+T_co, -T_contra], dict, bytes, InfluxDBResult, InfluxDBChunkedResult]

Returns Returns an async generator if chunked is True, otherwise returns a dictionary containing the parsed JSON response.

classmethod set_query_pattern(*name*, *qp*)

Defines custom methods to provide quick access to commonly used query patterns. Query patterns are plain strings, with optional the named placed holders. Named placed holders are processed as keyword arguments in str.format. Positional arguments are also supported.

Sample query pattern: "SELECT mean(load) FROM cpu_stats WHERE host = '{host}' AND time > now() - {days}d"

Parameters

- **name** (str) – Name of the query pattern class method. Must be a valid Python identifier.
- **qp** (str) – Query pattern string

Return type None

write(*data*, *measurement=None*, *db=None*, *precision=None*, *rp=None*, *tag_columns=None*, ***extra_tags*)

Writes data to InfluxDB. Input can be:

1) a string properly formatted in InfluxDB's line protocol
2) a dictionary-like object containing four keys: measurement, time, tags, fields
3) a Pandas DataFrame with a DatetimeIndex
4) an iterable of one of above

Input data in formats 2-4 are parsed to the line protocol before being written to InfluxDB. See the InfluxDB docs for more details.

Parameters

- **data** (Union[AnyStr, Mapping[~KT, +VT_co], Iterable[Union[AnyStr, Mapping[~KT, +VT_co]]]]) – Input data (see description above).
- **measurement** (Optional[str]) – Measurement name. Mandatory when when writing DataFrames only. When writing dictionary-like data, this field is treated as the default value for points that do not contain a *measurement* field.
- **db** (Optional[str]) – Database to be written to. Defaults to *self.db*.
- **precision** (Optional[str]) – Sets the precision for the supplied Unix time values. Ignored if input timestamp data is of non-integer type. Valid values: {'ns', 'u', 'µ', 'ms', 's', 'm', 'h'}
- **rp** (Optional[str]) – Sets the target retention policy for the write. If unspecified, data is written to the default retention policy.

- **tag_columns** (Optional[Iterable[+T_co]]) – Columns to be treated as tags (used when writing DataFrames only)
- **extra_tags** – Additional tags to be added to all points passed.

Return type bool

Returns Returns *True* if insert is successful. Raises *ValueError* exception otherwise.

exception aioinflux.client.**InfluxDBWriteError**(*resp*)

4.2 Serialization

aioinflux.serialization.common.**escape**(*string*, *escape_pattern*)
Assistant function for string escaping

class aioinflux.serialization.datapoint.**DataPoint**
Base class for dynamically generated datapoint class

items()
Returns an iterator over pair of keys and values

to_dict()
Converts datapoint to a regular dictionary

Return type dict

to_lineprotocol()
Returns InfluxDB line protocol representation of datapoint

Return type bytes

class aioinflux.serialization.datapoint.**InfluxType**
An enumeration.

aioinflux.serialization.datapoint.**datapoint**(*schema=None*, *name='DataPoint'*, ***, *rm_none=False*, *fill_none=False*, *extra_tags=None*)
Dynamic datapoint class factory

Can be used as a decorator (similar to Python 3.7 dataclasses) or as a function (similar to namedtuple(), but mutable).

Main characteristics:

- Supports accessing field values by attribute or subscription
- Support dict-like iteration via items method
- Built-in serialization to InfluxDB line protocol through the to_lineprotocol method.
- About 2-3x faster serialization than the serialization.mapping module.
  - Difference gets smaller (1x-1.5x) when rm_none=True or when the number of fields/tags is very large (20+).

Parameters

- **schema** – Dictionary-based (functional namedtuple style) or @dataclass decorator-based (dataclass style) measurement schema
- **name** – Class name (used when passing schema dictionaries only)

- **rm_none** – Whether apply a regex to remove `None` values from. If `False`, passing `None` values to boolean, integer or float or time fields will result in write errors. Setting to `True` is "safer" but impacts performance.
- **fill_none** – Whether or not to set missing fields to `None`. Likely best used together with `rm_none=True`.
- **extra_tags** – Hard coded tags to be added to every point generated.

CHAPTER 5

Contributing

To contribute, fork the repository on GitHub, make your changes and submit a pull request.
Aioinflux is not a mature project yet, so just simply raising issues is also greatly appreciated :)

CHAPTER 6

Alternatives

- InfluxDB-Python: The official blocking-only client. Based on Requests.
- influx-sansio: Fork of aioinflux using curio/trio and asks as a backend.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index