pypond Documentation

Release 0.5.0

Monte M. Goode

Apr 13, 2017

Developer Documentation

CHAPTER 1

Overview

PyPond is a Python implementation of the JavaScript Pond timeseries library. At a very high level, both implementations offer classes and structures to collect, manipulate and transmit timeseries data. Time series transmission is done via a JSON-based wire format.

This implementation is available on GitHub and the API documentation is available at software.es.net (and alternately on RTD).

PyPond runs on python 2.7 and 3.3 through 3.6.

CHAPTER 2

Core Documentation

The main project site has extensive documentation on the various structures (Event, TimeRange, TimeSeries, etc) that both implementations use internally. There is no need to duplicate that conceptual documentation here since the python implementation follows the same API and uses the same structures.

The only real difference with pypond is that the method names have been changed to their obvious pythonic corollaries (`obj.toString()` becomes `obj.to_string()`) and any comparison methods named `.is()` in the JavaScript version have been renamed to `.same()` in pypond since `is` is a reserved word in python.

The tests can also be referred to as a fairly complete set of examples as well.

API Documentation

Detailed API documentation. The tests can also be referred to as a fairly complete set of examples as well.

- genindex
- modindex
- search

Notes on time handling

UTC vs. local time

Initializing Event objects

All of the Event variants can be initialized with UTC milliseconds since the epoch, or an **aware** python `datetime` object. If a **naive** `datetime` object is passed in, an exception will be raised. When passing in a `datetime` object, it is encouraged that they be in UTC as well.

Be aware that if an aware non-UTC/local time `datetime` object is passed in, a warning will be issued, it will be converted to a UTC `datetime` object and that's what will be used internally. The conversion will be done like this using the python `datetime` library and the third party `pytz` library thusly:

```
dtime.astimezone(pytz.UTC)
```

And the resulting `datetime` object will be converted to milliseconds (see section on precision).

This is for consistency, for parity with the JavaScript Date library that uses epoch ms at its core, and because the Pond/PyPond wire format relies on epoch ms. One really can't go wrong with initially reporting all of their events using milliseconds since the epoch. *Please consider doing that.*

Rendering in local time

Of course there are cases where is desirable to represent time series data in the user's local time zone. Like in a graphing application. Even though PyPond does only business in UTC internally, this is possible. This can be changed on how you window and aggregate the data.

See the section on Aggregation in the main Pond Pipeline documentation. Note how you specify the `.windowBy()` (`.window_by()` in PyPond) value in the pipeline chain. This can be a fixed value like `1d` where it will aggregate the data into daily buckets. Fixed windows like that can **only** be rendered in UTC. Or it can be a non-fixed value like `daily` which will also aggregate the data into daily buckets, but the user can choose how to render the data in that case.

The default will be to render in UTC - any such choice will always default to UTC, the user will always need to set `utc=False` where appropriate. But when using a non-fixed window, the optional utc boolean can be set:

```
kcol = (
    Pipeline()
    .from_source(timeseries)
    .window_by('daily', utc=False)
    .emit_on('eachEvent')
    .aggregate({'in': Functions.avg(), 'out': Functions.avg()})
    .to_keyed_collections()
)
```

Then the aggregation key/buckets be daily averages in the local time zone.

There is also a trio of helper functions in the `TimeSeries` class that presents a higher level access to this functionality:

```
TimeSeries.daily_rollup()
TimeSeries.monthly_rollup()
TimeSeries.yearly_rollup()
```

They all take a `dict` of a column name and an aggregation function as in the above example:

```
TimeSeries.monthly_rollup({'in': Functions.avg(), 'out': Functions.avg()})
```

And the data will automatically be rendered in local time.

Conversion to local time

When the conversion covered in the previous section happens, the user has no control over **what** time zone it will be rendered to. All conversions will automatically happen using the local time zone as determined by the `tzlocal` library:

```
LOCAL_TZ = tzlocal.get_localzone()
```

This is primarily for parity with the JavaScript library which will be running browser-side and will be localizing as apropos. Moreover, the scope of this library is not to be a time handling swiss army knife.

Local time and the IndexedEvent class

The only `Event` class that explicitly takes a `utc=False` flag is the `IndexedEvent` class. It behaves somewhat differently than the `Event` and `TimeRangeEvent` classes which do not. Rather than being initialized with an epoch ms timestamp or a `datetime` object they are initialized with strings of the following formats:

```
    The index string arg will may be of two forms:

    - 2015-07-14  (day)
    - 2015-07     (month)
    - 2015        (year)

    or:

    - 1d-278      (range, in n x days, hours, minutes or seconds)

    and return a TimeRange for that time. The TimeRange may be considered to be
    local time or UTC time, depending on the utc flag passed in.
```

A UTC conversion will still happen under the hood, just a little differently.

If an `Index` (which is the underlying time-handling structure to `IndexedEvent`) is initialized thusly:

```
utc = Index('2015-07-14')
```

That is a daily index and is internally creating a range spanning that entire day. So looking at the internal timestamps yields this:

```
print(utc.begin(), utc.end())
2015-07-14 00:00:00+00:00 2015-07-14 23:59:59+00:00
```

But doing the same thing with `utc=False` (if you are in Pacific Time) yields this:

```
local = Index('2015-07-14', utc=False)
print(local.begin(), local.end())
2015-07-14 07:00:00+00:00 2015-07-15 06:59:59+00:00
```

The time range is **not** internally held as spanning that day in the local time zone, it is converted and reflected in UTC.

Yet another example of why it is preferred to input and store the data in UTC and view it in a localized way.

Precision

Internal timestamps are precise down to the millisecond even though the python `datetime` object is precise down to the microsecond. This is primarily for parity with the JavaScript library - the JS `Date` object is only accurate down to the millisecond. Unit testing showed that allowing microsecond accuracy exposed discrepancies between times that should have been "the same."

It is perfectly fine to pass in python `datetime` objects that have microsecond accuracy, just be aware that it will be rounded to milliseconds automatically.

Data columns: field_spec and field_path

There are some points to note about the nomenclature that the pypond and pond code bases use to refer to the "columns of data" in the time series event objects. This TimeSeries:

```
DATA = dict(
    name="traffic",
    columns=["time", "value", "status"],
    points=[
        [1400425947000, 52, "ok"],
        [1400425948000, 18, "ok"],
        [1400425949000, 26, "fail"],
        [1400425950000, 93, "offline"]
    ]
)
```

contains two columns: value and status.

However this TimeSeries:

```
DATA_FLOW = dict(
    name="traffic",
    columns=["time", "direction"],
    points=[
        [1400425947000, {'in': 1, 'out': 2}],
        [1400425948000, {'in': 3, 'out': 4}],
        [1400425949000, {'in': 5, 'out': 6}],
        [1400425950000, {'in': 7, 'out': 8}]
    ]
)
```

contains only one column direction, but that column has two more columns - in and out - nested under it. In the following examples, these nested columns will be referred to as "deep paths."

When specifying columns to the methods that set, retrieve and manipulate data, we use two argument types: field_spec and field_path. They are similar yet different enough to warrant this document.

field_path

A field_path refers to a **single** column in a series. Any method that takes a field_path as an argument only acts on one column at a time. The value passed to this argument can be either a string, a list or None.

String variant

When a string is passed, it can be one of the following formats:

- simple path - the name of a single "top level" column. In the DATA example above, this would be either value or status.
- deep path - the path pointing to a single nested columns with each "segment" of the path delimited with a period. In the DATA_FLOW example above, the incoming data could be retrieved with direction.in as the field_path.

List variant

When a `list` (or `tuple`) is passed as a `field_path`, each element in the iterable is **a single segment of the path to a column**. So to compare with the string examples:

- `['value']` would be equivalent to the string `value`.
- `['direction', 'in']` would be equivalent to the string `direction.in`.

This is particularly important to note because **this behavior is different** than passing a list to a `field_spec` arg.

`None`

If no `field_path` is specified (defaulting to `None`), then the default column `value` will be used.

field_spec

A `field_spec` refers to **one or more** columns in a series. When a method takes a `field_spec`, it may act on multiple columns in a `TimeSeries`. The value passed to this argument can be either a string, a list or `None`.

String variant

The string variant is essentially identical to the `field_path` string variant - it is a path to a single column of one of the following formats:

- simple path - the name of a single “top level” column. In the `DATA` example above, this would be either `value` or `status`.
- deep path - the path pointing to a single nested columns with each “segment” of the path delimited with a period. In the `DATA_FLOW` example above, the incoming data could be retrieved with `direction.in` as the `field_path`.

List variant

Passing a `list` (or `tuple`) to `field_spec` is different than the aforementioned behavior in that it is explicitly referring to **one or more columns**. Rather than each element being segments of a path, **each element is a full path to a single column**.

Using the previous examples:

- `['in', 'out']` would act on both the `in` and `out` columns from the `DATA` example.
- `['direction.in', 'direction.out']` - here each element is a fully formed “deep path” to the two data columns in the `DATA_FLOW` example.

The lists do not have to have more than one element: `['value'] == 'value'`.

NOTE: accidentally passing this style of list to an arg that is actually a `field_path` will most likely result in an `EventException` being raised. Passing something like `['in', 'out']` as a `field_path` will attempt to retrieve the nested column `in.out` which probably doesn’t exist.

`None`

If no `field_spec` is specified (defaulting to `None`), then the default column `value` will be used.

field_spec_list

This is a less common variant of the `field_spec`. It is used in the case where the method is going to **specifically act on multiple columns**. Otherwise, it's usage is identical to the list variant of `field_spec`.

Fill and other sanitizing methods

Real world data can have gaps, bad names, or occur at irregular intervals. The pypond toolkit contains some methods to adjust or sanitize a series of less than optimal data. As with all other mutation operations in pypond, these methods will return new `Event` objects, new `Collections` and new `TimeSeries` as apropos.

Fill

Data might contain missing or otherwise invalid values. `TimeSeries.fill()` can perform a variety of fill operations to smooth or make sure that the data can be processed in math operations without blowing up.

In pypond, a value is considered "invalid" if it is python `None`, a `NaN` (not a number) value, or an empty string.

Usage

The method prototype looks like this:

```
def fill(self, field_spec=None, method='zero', limit=None)
```

- the `field_spec` argument is the same as it is in the rest of the code - a string or list of strings denoting "columns" in the data. It can point `to.deep.values` using the usual dot notation.
- the `method` arg denotes the fill method to use. Valid values are **zero**, **pad** and **linear**.
- the `limit` arg places a limit on the number of events that will be filled and returned in the new `TimeSeries`. The default is to fill all the events with no limit.

Complete sample usage could look like this:

```
ts = TimeSeries(simple_missing_data)

new_ts = ts.fill(field_spec=['direction.in', 'direction.out'],
                 method='linear', limit=6)
```

Fill methods

There are three fill options:

- `zero` - the default - will transform any invalid value to a zero.
- `pad` - replaces an invalid value with the the previous good value: `[1, None, None, 3]` becomes `[1, 1, 1, 3]`.
- `linear` - interpolate the gaps based on the surrounding good values: `[1, None, None, None, 3]` becomes `[1, 1.5, 2, 2.5, 3]`.

Neither `pad` or `linear` can fill the first value in a series if it is invalid, and they can't start filling until good value has been seen: `[None, None, None, 1, 2, 3]` would remain unchanged. Similarly, `linear` can not fill the last value in a series.

The `fill_limit` arg

The optional arg `fill_limit` controls how many values will be filled before it gives up and starts returning the invalid data until a valid value is seen again.

There might be a situation where it makes sense to fill in a couple of missing values, but no sense to pad out long spans of missing data. This arg sets the limit of the number of missing values that will be filled - or in the case of `linear` *attempt* to be filled - before it just starts returning invalid data until the next valid value is seen.

So given `fill_limit=2` the following values will be filled in the following ways:

```
Original:
    [1, None, None, None, 5, 6, 7]

Zero:
    [1, 0, 0, None, 5, 6, 7]

Pad:
    [1, 1, 1, None, 5, 6, 7]

Linear:
    [1, None, None, None, 5, 6, 7]
```

Using methods `zero` and `pad` the first two missing values are filled and the third is skipped. When using the `linear` method, nothing gets filled because a valid value has not been seen before the limit has been reached, so it just gives up and returns the missing data.

When filling multiple columns, the count is maintained on a per-column basis. So given the following data:

```
    simple_missing_data = dict(
        name="traffic",
        columns=["time", "direction"],
        points=[
            [1400425947000, {'in': 1, 'out': None}],
            [1400425948000, {'in': None, 'out': None}],
            [1400425949000, {'in': None, 'out': None}],
            [1400425950000, {'in': 3, 'out': 8}],
            [1400425960000, {'in': None, 'out': None}],
            [1400425970000, {'in': None, 'out': 12}],
            [1400425980000, {'in': None, 'out': 13}],
            [1400425990000, {'in': 7, 'out': None}],
            [1400426000000, {'in': 8, 'out': None}],
            [1400426010000, {'in': 9, 'out': None}],
            [1400426020000, {'in': 10, 'out': None}],
        ]
    )
```

The `in` and `out` sub-columns will be counted and filled independently of each other.

If `fill_limit` is not set, no limits will be placed on the fill and all values will be filled as apropos to the selected method.

Constructing `linear` fill `Pipeline` chains

`TimeSeries.fill()` will be the common entry point for the `Filler`, but a `Pipeline` can be constructed as well. Even though the default behavior of `TimeSeries.fill()` applies to all fill methods, the `linear` fill logic is somewhat different than the `zero` and `pad` methods. Note the following points when creating your own `method='linear'` processing chain.

- When constructing a `Pipeline` to do a `linear` fill on multiple columns, chain them together like this rather than passing in a `field_spec` that is a list of columns:

```
Pipeline()
.from_source(ts)
.fill(field_spec='direction.in', method='linear')
.fill(field_spec='direction.out', method='linear')
.to_keyed_collections()
```

- If a non numeric value (as determined by `isinstance(val, numbers.Number)`) is encountered when doing a `linear` fill, a warning will be issued and that column will not be processed.
- When using streaming input like `Stream`, it is a best practice to set a limit using the optional arg `fill_limit`. This will ensure events will continue being emitted if the data hits a long run of invalid values.
- When using an unbounded source, make sure to shut it down "cleanly" using `.stop()`. This will ensure `.flush()` is called so any unfilled cached events are emitted.

Rename

It might be necessary to rename the columns/data keys in the events in a `TimeSeries`. It is preferable to just give the columns/keys the desired names when the `Event` objects are being instantiated. This is because using `TimeSeries.rename()` will create all new `Event` objects and a new `TimeSeries` as well. But if that is necessary, use this method.

Usage

This method takes a python dict of strings in the format `{'key': 'new_key'}`. This example:

```
ts = TimeSeries(TICKET_RANGE)

renamed = ts.rename_columns({'title': 'event', 'esnet_ticket': 'ticket'})
```

will rename the existing column `title` to `event`, etc.

Limitations

Unlike other uses of a `field_spec` to point at a `deep.nested.value` in pypond, `.rename()` only allows renaming a 'top level' column/key. If the data payload looks like this:

```
{'direction': {'in': 5, 'out': 7}}
```

The top level key `direction` can be renamed but the nested keys `in` and `out` can not.

Align

The align processor takes a `TimeSeries` of events that might come in with timestamps at uneven intervals and produces a new series of those points aligned on precise time window boundaries. A series containing four events with following timestamps:

```
0:40
1:05
1:45
2:10
```

Given a window of `1m` (one minute), a new series with two events at the following times will be produced:

```
1:00
2:00
```

Only a series of `Event` objects can be aligned. `IndexedEvent` objects are basically already aligned and it makes no sense in the case of a `TimeRangeEvent`.

It should also be noted that the emitted/aligned event will only contain the fields that alignment was requested on. Which is to say if you have two columns, `in` and `out`, and only request to align the `in` column, the `out` value will not be contained in the emitted event.

Usage

The full argument usage of the align method:

```
ts = TimeSeries(DATA_WITH_GAPS)
aligned = ts.align(field_spec='value', window='1m', method='linear', limit=2)
```

- `field_spec` - indicates which fields should be interpolated by the selected `method`. Typical usage of this arg type. If not supplied, then the default field `value` will be used.
- `window` - an integer and the usual `s/m/h/d` notation like `1m`, `30s`, `6h`, etc. The emitted events will be emitted on the indicated window boundaries. Due to the nature of the interpolation, one would want to use a window close to the frequency of the events. It would make little sense to set a window of `5h` on hourly data, etc.
- `method` - the interpolation method to be used: `linear` (the default) and `hold`.
- `limit` - sets a limit on the number of boundary interpolated events will be produced. If `limit=2`, `window='1m'` and two events come in at the following times:

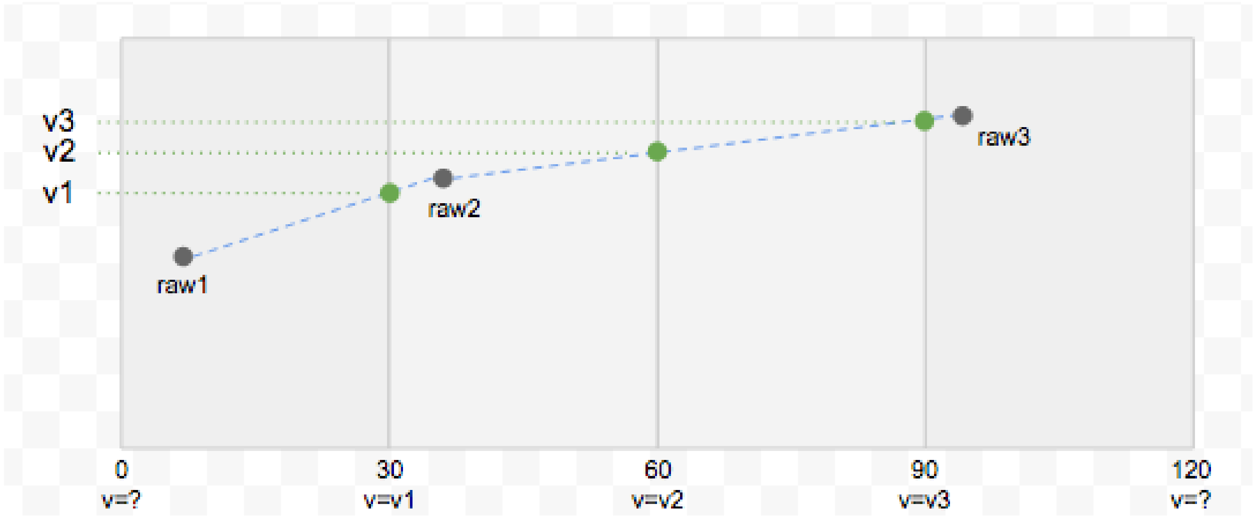
```
0:45
3:15
```

That would normally produce events on three window boundaries `1:00, 2:00 and 3:00` and that exceeds the `limit` so those events will have `None` as a value instead of an interpolated value.

Fill methods

Linear

This is the default method. It interpolates differential values in `Event` objects on the window boundaries using a strategy like this:



The green points are the events that will be produced by the `linear` fill method by interpolating the raw points. It also shows why it makes little sense to use a window significantly larger than the frequency of the events. When the window is set too wide for the data, many of the points in the middle of the window will be disregarded since the generated points are interpolated from the last event in the previous window and the first one in the current window.

Hold

This is a much simpler method. It just fills the selected field(s) with the corresponding value from the previous event.

Rate (derivative)

This generates a new `TimeSeries` of `TimeRangeEvent` objects which contain the derivative between columns in two consecutive `Event` objects. The start and end time of the time range events correspond to the timestamps of the two events the calculation was derived from.

The primary use case for this was to generate rate data from monotonically increasing SNMP counter values like this:

```
TimeSeries(RAW_COUNTERS).align(field_spec='in', window='30s').rate('in')
```

This would take the raw counter data, do a linear alignment on them on 30 second window boundaries, and then calculate the rates by calculating the derivative between the aligned boundaries.

However it is not necessary to align your events first, just calling `.rate()` will generate time range events with the derivative between the consecutive events.

Usage

The method prototype:

```
def rate(self, field_spec=None, allow_negative=True)
```

- `field_spec` - indicates which fields should be interpolated by the selected `method`. Typical usage of this arg type. If not supplied, then the default field `value` will be used.
- `allow_negative` - if left defaulting to `True`, then if a negative derivative is calculated, that will be used as the value in the new event. If set to `False` a negative derivative will be set to `None` instead. There are certain use cases - like if a monotonically increasing counter gets reset - that this is the desired outcome.

Release notes

Notes about releases, API changes, etc.

0.4

First stable release with full feature parity with the Pond JS code base.

0.5

0.5.0

NOTE: `key` means "the specific timestamp, index or time range an event object exists at."

Major changes:

- `Event.merge()` has been changed and is not backwards compatible with the 0.4 version. Previously it took a list of Event objects at the same key and returns a single, merged Event. Now it takes a list of Event objects that can be of differing keys and returns a list of Events where the events at the same key have their values merged into a single event. To wit: `[e(1, {'a': 1}), e(1, {'b': 2}), e(2, {'a': 3}), e(2, {'b': 4})] -> [e(1, {'a': 1, 'b': 2}), e(2, {'a': 3, 'b': 4})]`
- `Event.combine()` has been re-worked to accommodate this, but this is mostly for performance and should be transparent to the user.
- `Event.avg()` and `Event.sum()` (which are helper functions that use `Event.combine()`) now behave like `Event.merge()` and return a list of summed/averaged events, rather than a single event at one key.

Additions:

- `Collection.at_key()` retrieves all the events in a `Collection` at a specified key.
- `Collection.dedup()` removes duplicate (events at the same key) Event objects from a `Collection`.
- `Collection.event_list_as_map()` returns the Event objects in a `Collection` as a `dict` of `list` where the key is the `key` and the list contains the events at that `key`.
- `Event.key()` and `Event.type()` have been added but are mostly used internally. Have been added to all three event variants.
- `Event.is_duplicate()` compares two events and returns `True` if they are of the same time and exist at the same key. Can also be used to compare payload values as well with an optional flag.

Various:

- Added a boolean flag to allow `TimeSeries.daily_rollup()` `.monthly_rollup()` and `.yearly_rollup()` to render results in UTC rather than localtime. They default to rendering in localtime due to client-side concerns (like rendering a chart), but can now render in UTC since it is being used in server-side applications.
- Fixed a bug that impacted `TimeSeries/Collection.at_time()` when the first event should be returned.

Running the tests

Running the unit tests will probably only be of interest to other developers. There is a test module that tests interoperability with the JavaScript library (`pypond/tests/interop_test.py`) that will require some additional setup.

1. It will check to find the `node` executable somewhere in the path. If it is not found, those tests will fail.
2. The Pond source will need to be checked out at the same level "alongside" of the pypond source and then execute `npm install` followed by `npm run build` at the root level of the pond source. (Running `npm run build` should be a formality, but is included just in case the `pond/lib` directory was not properly regenerated from `pond/src`)
3. Execute `pip install -r dev-requirements.txt` and then run `nosetests` from either the source root or test directory (`pypond/` or `pypond/tests/`). The `pip` command will also install pypond in "develop" mode.

That particular test sends the data on a round trip by:

1. generating the wire format using the Python structures
2. sends it to an external script run by `node` as an arg
3. which reconstitutes the wire format as a JS structure
4. then the JS structure is used to generate the wire format again
5. wire format is returned to the calling unit test over stdout
6. a new Python structure is created with the incoming wire format
7. that structure is checked against the original data the first TimeSeries was created from.

All of the other tests are just standard-issue Python unit tests.

The tests can also be referred to as a fairly complete set of examples as well.