pystorm Documentation

Release 3.1.4

Parsely

Oct 01, 2018

Contents

pystorm lets you run Python code against real-time streams of data. Integrates with Apache Storm.

CHAPTER 1

Quickstart

1.1 Dependencies

1.2 Spouts and Bolts

The general flow for creating new spouts and bolts using pystorm is to add them to your `src` folder and update the corresponding topology definition.

Let's create a spout that emits sentences until the end of time:

```
import itertools

from pystorm.spout import Spout


class SentenceSpout(Spout):

    def initialize(self, stormconf, context):
        self.sentences = [
            "She advised him to take a long holiday, so he immediately quit work and␣
↪took a trip around the world",
            "I was very glad to get a present from her",
            "He will be here in half an hour",
            "She saw him eating a sandwich",
        ]
        self.sentences = itertools.cycle(self.sentences)

    def next_tuple(self):
        sentence = next(self.sentences)
        self.emit([sentence])

    def ack(self, tup_id):
        pass  # if a tuple is processed properly, do nothing
```

(continues on next page)

(continued from previous page)

```
    def fail(self, tup_id):
        pass  # if a tuple fails to process, do nothing
```

The magic in the code above happens in the `initialize()` and `next_tuple()` functions. Once the spout enters the main run loop, pystorm will call your spout's `initialize()` method. After initialization is complete, pystorm will continually call the spout's `next_tuple()` method where you're expected to emit tuples that match whatever you've defined in your topology definition.

Now let's create a bolt that takes in sentences, and spits out words:

```
import re

from pystorm.bolt import Bolt

class SentenceSplitterBolt(Bolt):

    def process(self, tup):
        sentence = tup.values[0]  # extract the sentence
        sentence = re.sub(r"[,.;!\?]", "", sentence)  # get rid of punctuation
        words = [[word.strip()] for word in sentence.split(" ") if word.strip()]
        if not words:
            # no words to process in the sentence, fail the tuple
            self.fail(tup)
            return

        for word in words:
            self.emit([word])
        # tuple acknowledgement is handled automatically
```

The bolt implementation is even simpler. We simply override the default `process()` method which pystorm calls when a tuple has been emitted by an incoming spout or bolt. You are welcome to do whatever processing you would like in this method and can further emit tuples or not depending on the purpose of your bolt.

If your `process()` method completes without raising an Exception, pystorm will automatically ensure any emits you have are anchored to the current tuple being processed and acknowledged after `process()` completes.

If an Exception is raised while `process()` is called, pystorm automatically fails the current tuple prior to killing the Python process.

1.2.1 Failed Tuples

In the example above, we added the ability to fail a sentence tuple if it did not provide any words. What happens when we fail a tuple? Storm will send a "fail" message back to the spout where the tuple originated from (in this case `SentenceSpout`) and pystorm calls the spout's *`fail()`* method. It's then up to your spout implementation to decide what to do. A spout could retry a failed tuple, send an error message, or kill the topology.

1.2.2 Bolt Configuration Options

You can disable the automatic acknowleding, anchoring or failing of tuples by adding class variables set to false for: `auto_ack`, `auto_anchor` or `auto_fail`. All three options are documented in *`pystorm.bolt.Bolt`*.

Example:

```
from pystorm.bolt import Bolt

class MyBolt(Bolt):

    auto_ack = False
    auto_fail = False

    def process(self, tup):
        # do stuff...
        if error:
          self.fail(tup)  # perform failure manually
        self.ack(tup)  # perform acknowledgement manually
```

1.2.3 Handling Tick Tuples

Ticks tuples are built into Storm to provide some simple forms of cron-like behaviour without actually having to use cron. You can receive and react to tick tuples as timer events with your python bolts using pystorm too.

The first step is to override `process_tick()` in your custom Bolt class. Once this is overridden, you can set the storm option `topology.tick.tuple.freq.secs=<frequency>` to cause a tick tuple to be emitted every `<frequency>` seconds.

You can see the full docs for `process_tick()` in *pystorm.bolt.Bolt*.

Example:

```
from pystorm.bolt import Bolt

class MyBolt(Bolt):

    def process_tick(self, freq):
        # An action we want to perform at some regular interval...
        self.flush_old_state()
```

Then, for example, to cause `process_tick()` to be called every 2 seconds on all of your bolts that override it, you can launch your topology under `sparse run` by setting the appropriate -o option and value as in the following example:

```
$ sparse run -o "topology.tick.tuple.freq.secs=2" ...
```

CHAPTER 2

API

2.1 Tuples

class pystorm.component.**Tuple**(*id*, *component*, *stream*, *task*, *values*)

Storm's primitive data type passed around via streams.

Variables

- **id** – the ID of the Tuple.
- **component** – component that the Tuple was generated from.
- **stream** – the stream that the Tuple was emitted into.
- **task** – the task the Tuple was generated from.
- **values** – the payload of the Tuple where data is stored.

You should never have to instantiate an instance of a *pystorm.component.Tuple* yourself as pystorm handles this for you prior to, for example, a *pystorm.bolt.Bolt*'s `process()` method being called.

None of the emit methods for bolts or spouts require that you pass a *pystorm.component.Tuple* instance.

2.2 Components

Both *pystorm.bolt.Bolt* and *pystorm.spout.Spout* inherit from a common base-class, *pystorm.component.Component*. It handles the basic Multi-Lang IPC between Storm and Python.

class pystorm.component.**Component**(*input_stream=<open file '<stdin>', mode 'r'>*, *output_stream=<open file '<stdout>', mode 'w'>*, *rdb_signal=u'SIGUSR1'*, *serializer=u'json'*)

Base class for spouts and bolts which contains class methods for logging messages back to the Storm worker process.

Variables

- **input_stream** – The `file`-like object to use to retrieve commands from Storm. Defaults to `sys.stdin`.
- **output_stream** – The `file`-like object to send messages to Storm with. Defaults to `sys.stdout`.
- **topology_name** – The name of the topology sent by Storm in the initial handshake.
- **task_id** – The numerical task ID for this component, as sent by Storm in the initial handshake.
- **component_name** – The name of this component, as sent by Storm in the initial handshake.
- **debug** – A `bool` indicating whether or not Storm is running in debug mode. Specified by the *topology.debug* Storm setting.
- **storm_conf** – A `dict` containing the configuration values sent by Storm in the initial handshake with this component.
- **context** – The context of where this component is in the topology. See the Storm Multi-Lang protocol documentation for details.
- **pid** – An `int` indicating the process ID of this component as retrieved by `os.getpid()`.
- **logger** – A logger to use with this component.

Note: Using `Component.logger` combined with the `pystorm.component.StormHandler` handler is the recommended way for logging messages from your component. If you use `Component.log` instead, the logging messages will *always* be sent to Storm, even if they are `debug` level messages and you are running in production. Using `pystorm.component.StormHandler` ensures that you will instead have your logging messages filtered on the Python side and only have the messages you actually want logged serialized and sent to Storm.

- **serializer** – The `Serializer` that is used to serialize messages between Storm and Python.
- **exit_on_exception** – A `bool` indicating whether or not the process should exit when an exception other than `StormWentAwayError` is raised. Defaults to `True`.

emit(*tup*, *tup_id=None*, *stream=None*, *anchors=None*, *direct_task=None*, *need_task_ids=False*)

Emit a new Tuple to a stream.

Parameters

- **tup** (`list` or *pystorm.component.Tuple*) – the Tuple payload to send to Storm, should contain only JSON-serializable data.
- **tup_id** (*str*) – the ID for the Tuple. If omitted by a *pystorm.spout.Spout*, this emit will be unreliable.
- **stream** (*str*) – the ID of the stream to emit this Tuple to. Specify `None` to emit to default stream.
- **anchors** (*list*) – IDs the Tuples (or *pystorm.component.Tuple* instances) which the emitted Tuples should be anchored to. This is only passed by *pystorm.bolt.Bolt*.
- **direct_task** (*int*) – the task to send the Tuple to.

- **need_task_ids** (*bool*) – indicate whether or not you'd like the task IDs the Tuple was emitted (default: `False`).

Returns `None`, unless `need_task_ids=True`, in which case it will be a `list` of task IDs that the Tuple was sent to if. Note that when specifying direct_task, this will be equal to `[direct_task]`.

initialize(*storm_conf*, *context*)
Called immediately after the initial handshake with Storm and before the main run loop. A good place to initialize connections to data sources.

Parameters

- **storm_conf** (*dict*) – the Storm configuration for this component. This is the configuration provided to the topology, merged in with cluster configuration on the worker node.
- **context** (*dict*) – information about the component's place within the topology such as: task IDs, inputs, outputs etc.

static is_heartbeat(*tup*)

Returns Whether or not the given Tuple is a heartbeat

log(*message*, *level=None*)
Log a message to Storm optionally providing a logging level.

Parameters

- **message** (*str*) – the log message to send to Storm.
- **level** (*str*) – the logging level that Storm should use when writing the `message`. Can be one of: trace, debug, info, warn, or error (default: `info`).

> **Warning:** This will send your message to Storm regardless of what level you specify. In almost all cases, you are better of using `Component.logger` and not setting `pystorm.log.path`, because that will use a `pystorm.component.StormHandler` to do the filtering on the Python side (instead of on the Java side after taking the time to serialize your message and send it to Storm).

raise_exception(*exception*, *tup=None*)
Report an exception back to Storm via logging.

Parameters

- **exception** – a Python exception.
- **tup** – a *Tuple* object.

read_handshake()
Read and process an initial handshake message from Storm.

read_message()
Read a message from Storm via serializer.

report_metric(*name*, *value*)
Report a custom metric back to Storm.

Parameters

- **name** – Name of the metric. This can be anything.
- **value** – Value of the metric. This is usually a number.

Only supported in Storm 0.9.3+.

Note: In order for this to work, the metric must be registered on the Storm side. See example code here.

run()
Main run loop for all components.

Performs initial handshake with Storm and reads Tuples handing them off to subclasses. Any exceptions are caught and logged back to Storm prior to the Python process exiting.

Warning: Subclasses should **not** override this method.

send_message(*message*)
Send a message to Storm via stdout.

2.2.1 Spouts

Spouts are data sources for topologies, they can read from any data source and emit tuples into streams.

class pystorm.spout.**Spout**(*input_stream=<open file '<stdin>', mode 'r'>, output_stream=<open file '<stdout>', mode 'w'>, rdb_signal=u'SIGUSR1', serializer=u'json'*)
Bases: *pystorm.component.Component*

Base class for all pystorm spouts.

For more information on spouts, consult Storm's Concepts documentation.

ack(*tup_id*)
Called when a bolt acknowledges a Tuple in the topology.

Parameters **tup_id** (*str*) – the ID of the Tuple that has been fully acknowledged in the topology.

activate()
Called when the Spout has been activated after being deactivated.

Note: This requires at least Storm 1.1.0.

deactivate()
Called when the Spout has been deactivated.

Note: This requires at least Storm 1.1.0.

emit(*tup*, *tup_id=None*, *stream=None*, *direct_task=None*, *need_task_ids=False*)
Emit a spout Tuple message.

Parameters

- **tup** (*list or tuple*) – the Tuple to send to Storm, should contain only JSON-serializable data.
- **tup_id** (*str*) – the ID for the Tuple. Leave this blank for an unreliable emit.

- **stream** (*str*) – ID of the stream this Tuple should be emitted to. Leave empty to emit to the default stream.
- **direct_task** (*int*) – the task to send the Tuple to if performing a direct emit.
- **need_task_ids** (*bool*) – indicate whether or not you'd like the task IDs the Tuple was emitted (default: `False`).

Returns `None`, unless `need_task_ids=True`, in which case it will be a `list` of task IDs that the Tuple was sent to if. Note that when specifying direct_task, this will be equal to `[direct_task]`.

fail(*tup_id*)
Called when a Tuple fails in the topology

A spout can choose to emit the Tuple again or ignore the fail. The default is to ignore.

Parameters **tup_id** (*str*) – the ID of the Tuple that has failed in the topology either due to a bolt calling `fail()` or a Tuple timing out.

initialize(*storm_conf*, *context*)
Called immediately after the initial handshake with Storm and before the main run loop. A good place to initialize connections to data sources.

Parameters

- **storm_conf** (*dict*) – the Storm configuration for this component. This is the configuration provided to the topology, merged in with cluster configuration on the worker node.
- **context** (*dict*) – information about the component's place within the topology such as: task IDs, inputs, outputs etc.

static is_heartbeat(*tup*)

Returns Whether or not the given Tuple is a heartbeat

log(*message*, *level=None*)
Log a message to Storm optionally providing a logging level.

Parameters

- **message** (*str*) – the log message to send to Storm.
- **level** (*str*) – the logging level that Storm should use when writing the `message`. Can be one of: trace, debug, info, warn, or error (default: `info`).

Warning: This will send your message to Storm regardless of what level you specify. In almost all cases, you are better of using `Component.logger` and not setting `pystorm.log.path`, because that will use a `pystorm.component.StormHandler` to do the filtering on the Python side (instead of on the Java side after taking the time to serialize your message and send it to Storm).

next_tuple()
Implement this function to emit Tuples as necessary.

This function should not block, or Storm will think the spout is dead. Instead, let it return and pystorm will send a noop to storm, which lets it know the spout is functioning.

raise_exception(*exception*, *tup=None*)
Report an exception back to Storm via logging.

Parameters

- **exception** – a Python exception.
- **tup** – a `Tuple` object.

read_handshake()
Read and process an initial handshake message from Storm.

read_message()
Read a message from Storm via serializer.

report_metric(*name*, *value*)
Report a custom metric back to Storm.

Parameters

- **name** – Name of the metric. This can be anything.
- **value** – Value of the metric. This is usually a number.

Only supported in Storm 0.9.3+.

Note: In order for this to work, the metric must be registered on the Storm side. See example code here.

---

run()
Main run loop for all components.

Performs initial handshake with Storm and reads Tuples handing them off to subclasses. Any exceptions are caught and logged back to Storm prior to the Python process exiting.

Warning: Subclasses should **not** override this method.

send_message(*message*)
Send a message to Storm via stdout.

class pystorm.spout.**ReliableSpout**(**args*, ***kwargs*)
Bases: *pystorm.spout.Spout*

Reliable spout that will automatically replay failed tuples.

Failed tuples will be replayed up to `max_fails` times.

For more information on spouts, consult Storm's Concepts documentation.

ack(*tup_id*)
Called when a bolt acknowledges a Tuple in the topology.

Parameters **tup_id** (*str*) – the ID of the Tuple that has been fully acknowledged in the topology.

activate()
Called when the Spout has been activated after being deactivated.

Note: This requires at least Storm 1.1.0.

---

deactivate()
Called when the Spout has been deactivated.

Note: This requires at least Storm 1.1.0.

emit(*tup*, *tup_id=None*, *stream=None*, *direct_task=None*, *need_task_ids=False*)
Emit a spout Tuple & add metadata about it to *unacked_tuples*.

In order for this to work, *tup_id* is a required parameter.

See `Bolt.emit()`.

fail(*tup_id*)
Called when a Tuple fails in the topology

A reliable spout will replay a failed tuple up to `max_fails` times.

Parameters **tup_id** (*str*) – the ID of the Tuple that has failed in the topology either due to a bolt calling `fail()` or a Tuple timing out.

initialize(*storm_conf*, *context*)
Called immediately after the initial handshake with Storm and before the main run loop. A good place to initialize connections to data sources.

Parameters

- **storm_conf** (*dict*) – the Storm configuration for this component. This is the configuration provided to the topology, merged in with cluster configuration on the worker node.
- **context** (*dict*) – information about the component's place within the topology such as: task IDs, inputs, outputs etc.

static is_heartbeat(*tup*)

Returns Whether or not the given Tuple is a heartbeat

log(*message*, *level=None*)
Log a message to Storm optionally providing a logging level.

Parameters

- **message** (*str*) – the log message to send to Storm.
- **level** (*str*) – the logging level that Storm should use when writing the `message`. Can be one of: trace, debug, info, warn, or error (default: `info`).

Warning: This will send your message to Storm regardless of what level you specify. In almost all cases, you are better of using `Component.logger` and not setting `pystorm.log.path`, because that will use a `pystorm.component.StormHandler` to do the filtering on the Python side (instead of on the Java side after taking the time to serialize your message and send it to Storm).

next_tuple()
Implement this function to emit Tuples as necessary.

This function should not block, or Storm will think the spout is dead. Instead, let it return and pystorm will send a noop to storm, which lets it know the spout is functioning.

raise_exception(*exception*, *tup=None*)
Report an exception back to Storm via logging.

Parameters

- **exception** – a Python exception.

- **tup** – a `Tuple` object.

read_handshake()
Read and process an initial handshake message from Storm.

read_message()
Read a message from Storm via serializer.

report_metric(*name*, *value*)
Report a custom metric back to Storm.

Parameters

- **name** – Name of the metric. This can be anything.
- **value** – Value of the metric. This is usually a number.

Only supported in Storm 0.9.3+.

Note: In order for this to work, the metric must be registered on the Storm side. See example code here.

---

run()
Main run loop for all components.

Performs initial handshake with Storm and reads Tuples handing them off to subclasses. Any exceptions are caught and logged back to Storm prior to the Python process exiting.

Warning: Subclasses should **not** override this method.

send_message(*message*)
Send a message to Storm via stdout.

2.2.2 Bolts

class pystorm.bolt.**Bolt**(**args*, ***kwargs*)
Bases: *pystorm.component.Component*

The base class for all pystorm bolts.

For more information on bolts, consult Storm's Concepts documentation.

Variables

- **auto_anchor** – A `bool` indicating whether or not the bolt should automatically anchor emits to the incoming Tuple ID. Tuple anchoring is how Storm provides reliability, you can read more about Tuple anchoring in Storm's docs. Default is `True`.
- **auto_ack** – A `bool` indicating whether or not the bolt should automatically acknowledge Tuples after `process()` is called. Default is `True`.
- **auto_fail** – A `bool` indicating whether or not the bolt should automatically fail Tuples when an exception occurs when the `process()` method is called. Default is `True`.

Example:

```
from pystorm.bolt import Bolt

class SentenceSplitterBolt(Bolt):
```

(continues on next page)

(continued from previous page)

```
    def process(self, tup):
        sentence = tup.values[0]
        for word in sentence.split(" "):
            self.emit([word])
```

ack(*tup*)
Indicate that processing of a Tuple has succeeded.

Parameters **tup** (`str` or *pystorm.component.Tuple*) – the Tuple to acknowledge.

emit(*tup*, *stream=None*, *anchors=None*, *direct_task=None*, *need_task_ids=False*)
Emit a new Tuple to a stream.

Parameters

- **tup** (`list` or *pystorm.component.Tuple*) – the Tuple payload to send to Storm, should contain only JSON-serializable data.
- **stream** (*str*) – the ID of the stream to emit this Tuple to. Specify `None` to emit to default stream.
- **anchors** (*list*) – IDs the Tuples (or *pystorm.component.Tuple* instances) which the emitted Tuples should be anchored to. If `auto_anchor` is set to `True` and you have not specified `anchors`, `anchors` will be set to the incoming/most recent Tuple ID(s).
- **direct_task** (*int*) – the task to send the Tuple to.
- **need_task_ids** (*bool*) – indicate whether or not you'd like the task IDs the Tuple was emitted (default: `False`).

Returns `None`, unless `need_task_ids=True`, in which case it will be a `list` of task IDs that the Tuple was sent to if. Note that when specifying direct_task, this will be equal to `[direct_task]`.

fail(*tup*)
Indicate that processing of a Tuple has failed.

Parameters **tup** (`str` or *pystorm.component.Tuple*) – the Tuple to fail (its `id` if `str`).

initialize(*storm_conf*, *context*)
Called immediately after the initial handshake with Storm and before the main run loop. A good place to initialize connections to data sources.

Parameters

- **storm_conf** (*dict*) – the Storm configuration for this component. This is the configuration provided to the topology, merged in with cluster configuration on the worker node.
- **context** (*dict*) – information about the component's place within the topology such as: task IDs, inputs, outputs etc.

static is_heartbeat(*tup*)

Returns Whether or not the given Tuple is a heartbeat

static is_tick(*tup*)

Returns Whether or not the given Tuple is a tick Tuple

log(*message*, *level=None*)
Log a message to Storm optionally providing a logging level.

Parameters

- **message** (*str*) – the log message to send to Storm.
- **level** (*str*) – the logging level that Storm should use when writing the `message`. Can be one of: trace, debug, info, warn, or error (default: `info`).

> **Warning:** This will send your message to Storm regardless of what level you specify. In almost all cases, you are better of using `Component.logger` and not setting `pystorm.log.path`, because that will use a `pystorm.component.StormHandler` to do the filtering on the Python side (instead of on the Java side after taking the time to serialize your message and send it to Storm).

process(*tup*)
Process a single Tuple *pystorm.component.Tuple* of input

This should be overridden by subclasses. *pystorm.component.Tuple* objects contain metadata about which component, stream and task it came from. The actual values of the Tuple can be accessed by calling `tup.values`.

Parameters **tup** (*pystorm.component.Tuple*) – the Tuple to be processed.

process_tick(*tup*)
Process special 'tick Tuples' which allow time-based behaviour to be included in bolts.

Default behaviour is to ignore time ticks. This should be overridden by subclasses who wish to react to timer events via tick Tuples.

Tick Tuples will be sent to all bolts in a toplogy when the storm configuration option 'topology.tick.tuple.freq.secs' is set to an integer value, the number of seconds.

Parameters **tup** (*pystorm.component.Tuple*) – the Tuple to be processed.

raise_exception(*exception*, *tup=None*)
Report an exception back to Storm via logging.

Parameters

- **exception** – a Python exception.
- **tup** – a `Tuple` object.

read_handshake()
Read and process an initial handshake message from Storm.

read_message()
Read a message from Storm via serializer.

read_tuple()
Read a tuple from the pipe to Storm.

report_metric(*name*, *value*)
Report a custom metric back to Storm.

Parameters

- **name** – Name of the metric. This can be anything.
- **value** – Value of the metric. This is usually a number.

Only supported in Storm 0.9.3+.

Note: In order for this to work, the metric must be registered on the Storm side. See example code here.

---

run()
Main run loop for all components.

Performs initial handshake with Storm and reads Tuples handing them off to subclasses. Any exceptions are caught and logged back to Storm prior to the Python process exiting.

Warning: Subclasses should **not** override this method.

send_message(*message*)
Send a message to Storm via stdout.

class pystorm.bolt.**BatchingBolt**(**args*, ***kwargs*)
Bases: *pystorm.bolt.Bolt*

A bolt which batches Tuples for processing.

Batching Tuples is unexpectedly complex to do correctly. The main problem is that all bolts are single-threaded. The difficult comes when the topology is shutting down because Storm stops feeding the bolt Tuples. If the bolt is blocked waiting on stdin, then it can't process any waiting Tuples, or even ack ones that were asynchronously written to a data store.

This bolt helps with that by grouping Tuples received between tick Tuples into batches.

To use this class, you must implement `process_batch`. `group_key` can be optionally implemented so that Tuples are grouped before `process_batch` is even called.

Variables

- **auto_anchor** – A `bool` indicating whether or not the bolt should automatically anchor emits to the incoming Tuple ID. Tuple anchoring is how Storm provides reliability, you can read more about Tuple anchoring in Storm's docs. Default is `True`.
- **auto_ack** – A `bool` indicating whether or not the bolt should automatically acknowledge Tuples after `process_batch()` is called. Default is `True`.
- **auto_fail** – A `bool` indicating whether or not the bolt should automatically fail Tuples when an exception occurs when the `process_batch()` method is called. Default is `True`.
- **ticks_between_batches** – The number of tick Tuples to wait before processing a batch.

Example:

```
from pystorm.bolt import BatchingBolt

class WordCounterBolt(BatchingBolt):

    ticks_between_batches = 5

    def group_key(self, tup):
        word = tup.values[0]
        return word  # collect batches of words

```

(continues on next page)

(continued from previous page)

```
    def process_batch(self, key, tups):
        # emit the count of words we had per 5s batch
        self.emit([key, len(tups)])
```

ack(*tup*)
Indicate that processing of a Tuple has succeeded.

Parameters **tup** (`str` or *pystorm.component.Tuple*) – the Tuple to acknowledge.

emit(*tup*, ***kwargs*)
Modified emit that will not return task IDs after emitting.

See `pystorm.component.Bolt` for more information.

Returns `None`.

fail(*tup*)
Indicate that processing of a Tuple has failed.

Parameters **tup** (`str` or *pystorm.component.Tuple*) – the Tuple to fail (its `id` if `str`).

group_key(*tup*)
Return the group key used to group Tuples within a batch.

By default, returns None, which put all Tuples in a single batch, effectively just time-based batching. Override this to create multiple batches based on a key.

Parameters **tup** (*pystorm.component.Tuple*) – the Tuple used to extract a group key

Returns Any `hashable` value.

initialize(*storm_conf*, *context*)
Called immediately after the initial handshake with Storm and before the main run loop. A good place to initialize connections to data sources.

Parameters

- **storm_conf** (*dict*) – the Storm configuration for this component. This is the configuration provided to the topology, merged in with cluster configuration on the worker node.
- **context** (*dict*) – information about the component's place within the topology such as: task IDs, inputs, outputs etc.

static is_heartbeat(*tup*)

Returns Whether or not the given Tuple is a heartbeat

static is_tick(*tup*)

Returns Whether or not the given Tuple is a tick Tuple

log(*message*, *level=None*)
Log a message to Storm optionally providing a logging level.

Parameters

- **message** (*str*) – the log message to send to Storm.
- **level** (*str*) – the logging level that Storm should use when writing the `message`. Can be one of: trace, debug, info, warn, or error (default: `info`).

> **Warning:** This will send your message to Storm regardless of what level you specify. In almost all cases, you are better of using `Component.logger` and not setting `pystorm.log.path`, because that will use a `pystorm.component.StormHandler` to do the filtering on the Python side (instead of on the Java side after taking the time to serialize your message and send it to Storm).

process(*tup*)
Group non-tick Tuples into batches by `group_key`.

> **Warning:** This method should **not** be overriden. If you want to tweak how Tuples are grouped into batches, override `group_key`.

process_batch(*key*, *tups*)
Process a batch of Tuples. Should be overridden by subclasses.

Parameters

- **key** (*hashable*) – the group key for the list of batches.
- **tups** (*list*) – a *list* of *pystorm.component.Tuple* s for the group.

process_batches()
Iterate through all batches, call process_batch on them, and ack.

Separated out for the rare instances when we want to subclass BatchingBolt and customize what mechanism causes batches to be processed.

process_tick(*tick_tup*)
Increment tick counter, and call `process_batch` for all current batches if tick counter exceeds `ticks_between_batches`.

See `pystorm.component.Bolt` for more information.

> **Warning:** This method should **not** be overriden. If you want to tweak how Tuples are grouped into batches, override `group_key`.

raise_exception(*exception*, *tup=None*)
Report an exception back to Storm via logging.

Parameters

- **exception** – a Python exception.
- **tup** – a `Tuple` object.

read_handshake()
Read and process an initial handshake message from Storm.

read_message()
Read a message from Storm via serializer.

read_tuple()
Read a tuple from the pipe to Storm.

report_metric(*name*, *value*)
Report a custom metric back to Storm.

Parameters

- **name** – Name of the metric. This can be anything.
- **value** – Value of the metric. This is usually a number.

Only supported in Storm 0.9.3+.

Note: In order for this to work, the metric must be registered on the Storm side. See example code here.

---

run()
Main run loop for all components.

Performs initial handshake with Storm and reads Tuples handing them off to subclasses. Any exceptions are caught and logged back to Storm prior to the Python process exiting.

Warning: Subclasses should **not** override this method.

send_message(*message*)
Send a message to Storm via stdout.

class pystorm.bolt.**TicklessBatchingBolt**(**args*, ***kwargs*)
Bases: *pystorm.bolt.BatchingBolt*

A BatchingBolt which uses a timer thread instead of tick tuples.

Batching tuples is unexpectedly complex to do correctly. The main problem is that all bolts are single-threaded. The difficult comes when the topology is shutting down because Storm stops feeding the bolt tuples. If the bolt is blocked waiting on stdin, then it can't process any waiting tuples, or even ack ones that were asynchronously written to a data store.

This bolt helps with that grouping tuples based on a time interval and then processing them on a worker thread.

To use this class, you must implement `process_batch`. `group_key` can be optionally implemented so that tuples are grouped before `process_batch` is even called.

Variables

- **auto_anchor** – A `bool` indicating whether or not the bolt should automatically anchor emits to the incoming tuple ID. Tuple anchoring is how Storm provides reliability, you can read more about tuple anchoring in Storm's docs. Default is `True`.
- **auto_ack** – A `bool` indicating whether or not the bolt should automatically acknowledge tuples after `process_batch()` is called. Default is `True`.
- **auto_fail** – A `bool` indicating whether or not the bolt should automatically fail tuples when an exception occurs when the `process_batch()` method is called. Default is `True`.
- **secs_between_batches** – The time (in seconds) between calls to `process_batch()`. Note that if there are no tuples in any batch, the TicklessBatchingBolt will continue to sleep.

Note: Can be fractional to specify greater precision (e.g. 2.5).

Example:

```
from pystorm.bolt import TicklessBatchingBolt

class WordCounterBolt(TicklessBatchingBolt):

    secs_between_batches = 5

    def group_key(self, tup):
        word = tup.values[0]
        return word  # collect batches of words

    def process_batch(self, key, tups):
        # emit the count of words we had per 5s batch
        self.emit([key, len(tups)])
```

ack(*tup*)
Indicate that processing of a Tuple has succeeded.

Parameters **tup** (`str` or *pystorm.component.Tuple*) – the Tuple to acknowledge.

emit(*tup*, ***kwargs*)
Modified emit that will not return task IDs after emitting.

See `pystorm.component.Bolt` for more information.

Returns `None`.

fail(*tup*)
Indicate that processing of a Tuple has failed.

Parameters **tup** (`str` or *pystorm.component.Tuple*) – the Tuple to fail (its `id` if `str`).

group_key(*tup*)
Return the group key used to group Tuples within a batch.

By default, returns None, which put all Tuples in a single batch, effectively just time-based batching. Override this to create multiple batches based on a key.

Parameters **tup** (*pystorm.component.Tuple*) – the Tuple used to extract a group key

Returns Any `hashable` value.

initialize(*storm_conf*, *context*)
Called immediately after the initial handshake with Storm and before the main run loop. A good place to initialize connections to data sources.

Parameters

- **storm_conf** (*dict*) – the Storm configuration for this component. This is the configuration provided to the topology, merged in with cluster configuration on the worker node.
- **context** (*dict*) – information about the component's place within the topology such as: task IDs, inputs, outputs etc.

static is_heartbeat(*tup*)

Returns Whether or not the given Tuple is a heartbeat

static is_tick(*tup*)

Returns Whether or not the given Tuple is a tick Tuple

log(*message*, *level=None*)
Log a message to Storm optionally providing a logging level.

Parameters

- **message** (*str*) – the log message to send to Storm.
- **level** (*str*) – the logging level that Storm should use when writing the `message`. Can be one of: trace, debug, info, warn, or error (default: `info`).

> **Warning:** This will send your message to Storm regardless of what level you specify. In almost all cases, you are better of using `Component.logger` and not setting `pystorm.log.path`, because that will use a `pystorm.component.StormHandler` to do the filtering on the Python side (instead of on the Java side after taking the time to serialize your message and send it to Storm).

process(*tup*)
Group non-tick Tuples into batches by `group_key`.

> **Warning:** This method should **not** be overriden. If you want to tweak how Tuples are grouped into batches, override `group_key`.

process_batch(*key*, *tups*)
Process a batch of Tuples. Should be overridden by subclasses.

Parameters

- **key** (*hashable*) – the group key for the list of batches.
- **tups** (*list*) – a *list* of *pystorm.component.Tuple* s for the group.

process_batches()
Iterate through all batches, call process_batch on them, and ack.

Separated out for the rare instances when we want to subclass BatchingBolt and customize what mechanism causes batches to be processed.

process_tick(*tick_tup*)
Just ack tick tuples and ignore them.

raise_exception(*exception*, *tup=None*)
Report an exception back to Storm via logging.

Parameters

- **exception** – a Python exception.
- **tup** – a `Tuple` object.

read_handshake()
Read and process an initial handshake message from Storm.

read_message()
Read a message from Storm via serializer.

read_tuple()
Read a tuple from the pipe to Storm.

report_metric(*name*, *value*)
Report a custom metric back to Storm.

Parameters

- **name** – Name of the metric. This can be anything.

- **value** – Value of the metric. This is usually a number.

Only supported in Storm 0.9.3+.

Note: In order for this to work, the metric must be registered on the Storm side. See example code here.

---

run()
Main run loop for all components.

Performs initial handshake with Storm and reads Tuples handing them off to subclasses. Any exceptions are caught and logged back to Storm prior to the Python process exiting.

> **Warning:** Subclasses should **not** override this method.

send_message(*message*)
Send a message to Storm via stdout.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index

R

S

T