Nanostream Documentation

Release 0.1

Zachary Ernst

Feb 21, 2019

CONTENTS:

1 Overview **1**
- 1.1 Why is it? . . . 1
- 1.2 What is it? . . . 1
- 1.3 What isn't it? . . . 1
- 1.4 NanoStream pipelines . . . 2
- 1.5 Installing NanoStream . . . 3
- 1.6 Using NanoStream . . . 4
- 1.7 Rolling your own `NanoNode` class . . . 6
- 1.8 Composing and configuring `NanoNode` objects . . . 8

2 The Data Journey **11**
- 2.1 Overview . . . 11
- 2.2 Example: Making a GET request . . . 11

3 Treehorn **13**
- 3.1 Using Treehorn . . . 13
- 3.2 Summing up . . . 17

4 Implementation **19**
- 4.1 The data journey . . . 19

5 API Documentation **21**
- 5.1 Node module . . . 21
- 5.2 Civis-specific node types . . . 28
- 5.3 Data structures module . . . 29
- 5.4 Network nodes module . . . 40
- 5.5 NanoStreamMessage module . . . 41
- 5.6 PoisonPill module . . . 41
- 5.7 Trigger module . . . 41
- 5.8 Batch module . . . 42
- 5.9 NanoStreamQueue module . . . 42

6 License **43**

7 Indices and tables **45**

Python Module Index **47**

Index **49**

CHAPTER

ONE

OVERVIEW

NanoStream is a package of classes and functions that help you write consistent, efficient, configuration-driven ETL pipelines in Python. It is open-source and as simple as possible (but not simpler).

This overview tells you why NanoStream exists, and how it can help you escape from ETL hell.

1.1 Why is it?

Tolstoy said that every happy family is the same, but every unhappy family is unhappy in its own way. ETL pipelines are unhappy families.

Why are they so unhappy? Every engineer who does more than one project involving ETL eventually goes through the same stages of ETL grief. First, they think it's not so bad. Then they do another project and discover that they have to rewrite very similar code. Then they think, "Surely, I could have just written a few library functions and reused that code, saving lots of time." But when they try to do this, they discover that although their ETL projects are very similar, they are just different enough that their code isn't reusable. So they resign themselves to rewriting code over and over again. The code is unreliable, difficult to maintain, and usually poorly tested and documented because it's such a pain to write in the first place. The task of writing ETL pipelines is so lousy that engineering best practices tend to go out the window because the engineer has better things to do.

1.2 What is it?

NanoStream is an ETL framework for the real world. It aims to provide structure and consistency to your ETL pipelines, while still allowing you to write bespoke code for all of the weird little idiosyncratic features of your data. It is opinionated without being bossy.

The overall idea of NanoStream is simple. On the surface, it looks a lot like streaming frameworks such as Spark or Storm. You hook up various tasks in a directed graph called a "pipeline". The pipeline ingests data from or more places, transforms it, and loads the data somewhere else. But it differs from Spark-like systems in important ways:

1. It is agnostic between stream and batch. Batches of data can be turned into streams and vice-versa.
2. It is lightweight, requiring no specialized infrastructure or network configuration.
3. Its built-in functionality is specifically designed for ETL tasks.
4. It is meant to accommodate 90% of your ETL needs entirely by writing configuration files.

1.3 What isn't it?

There are many things that NanoStream is not:

1. It is not a Big Data(tm) tool. If you're handling petabytes of data, you do not want to use NanoStream.
2. It is not suitable for large amounts of computation. If you need to use dataframes to calculate lots of complex statistical information in real-time, this is not the tool for you.

Basically, NanoStream deliberately makes two trade-offs: (1) it gives up Big Data(tm) for simplicity; and (2) it gives up being a general-purpose analytic tool in favor of being very good at ETL.

1.4 NanoStream pipelines

An ETL pipeline in NanoStream is a series of nodes connected by queues. Data is generated or processed in each node, and the output is placed on a queue to be picked up by downstream nodes.

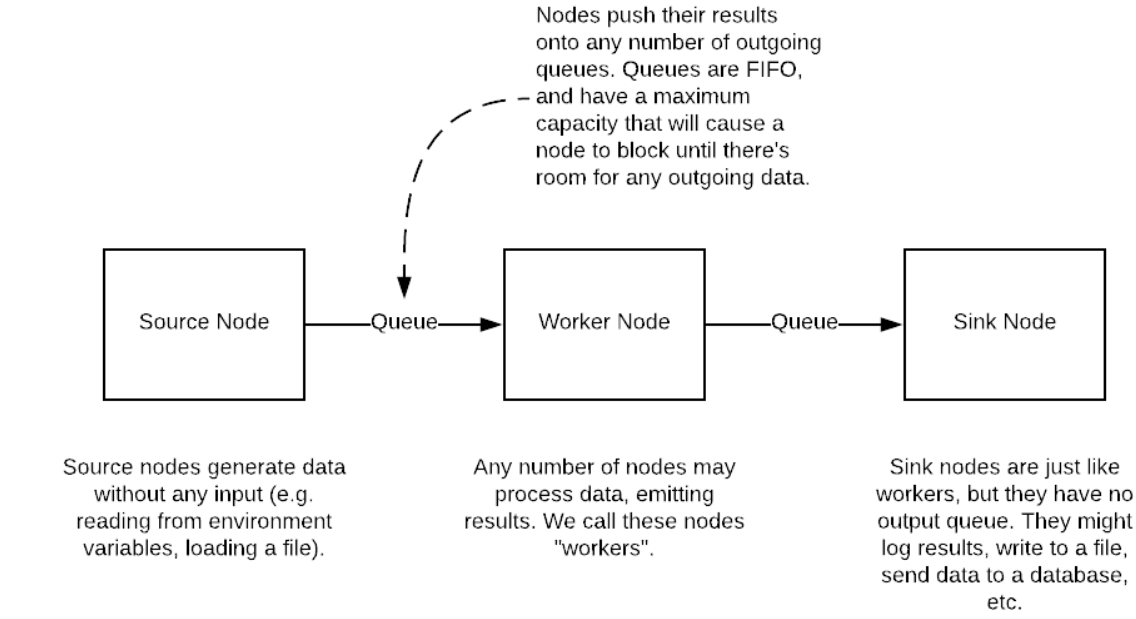

Fig. 1: Very high-level view of a NanoStream pipeline

For the sake of convenience, we distinguish between three types of nodes (although there's no real difference in their use or implementation):

1. Source nodes. These are nodes that generate data and send it to the rest of the pipeline. They might, for example, read data from an external data source such as an API endpoint or a database.
2. Worker nodes. The workers process data by picking up messages from their incoming queues. Their output is placed onto any number of outgoing queues to be further processed by downstream nodes.
3. Sink nodes. These are worker nodes with no outgoing queue. They will typically perform tasks such as inserting data into a database or generating statistics to be sent somewhere outside the pipeline.

All pipelines are implemented in pure Python (version >=3.5). Each node is instantiated from a class that inherits from the `NanoNode` class. Queues are never instantiated directly by the user; they are created automatically whenever two nodes are linked together.

There is a large (and growing) number of specialized `NanoNode` subclasses, each geared toward a specific task. Such tasks include:

1. Querying a table in a SQL database and sending the results downstream.
2. Making a request to a REST API, paging through the responses until there are no more results.
3. Ingesting individual messages from an upstream node and batching them together into a single message, or doing the reverse.
4. Reading environment variables.
5. Watching a directory for new files and sending the names of those files down the pipeline when they appear.
6. Filtering messages, letting them through the pipeline only if a particular test is passed.

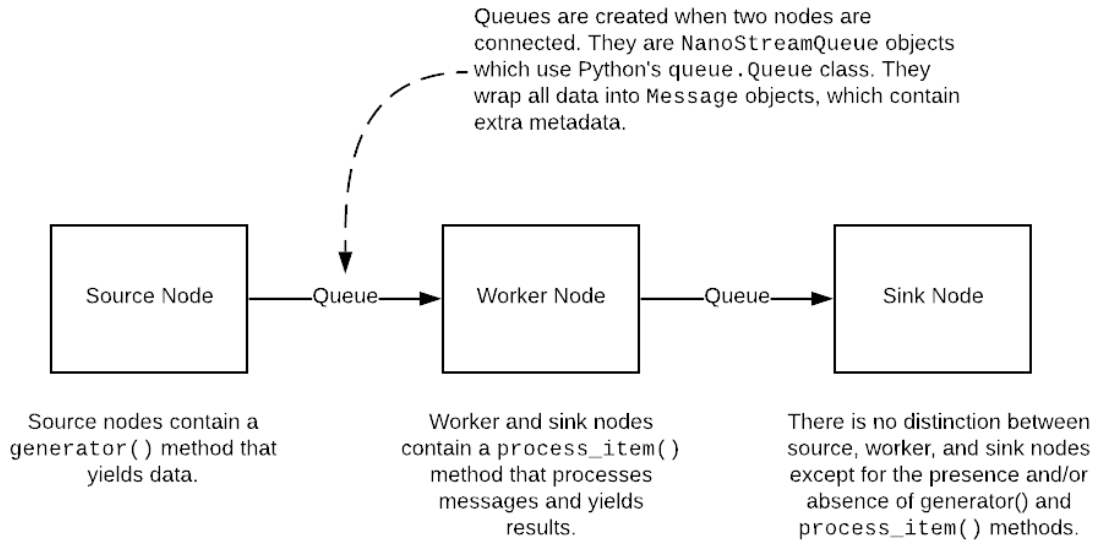

Fig. 2: Somewhat high-level view of a NanoStream pipeline

All results and messages passed among the nodes must be dictionary-like objects. By default, messages retain any keys and values that were created by upstream nodes.

The goal is for NanoStream to be "batteries included", with built-in `NanoNode` subclasses for every common ETL task. But because ETL pipelines generally have something weird going on somewhere, NanoNode makes it easy to roll your own node classes;

1.5 Installing NanoStream

NanoStream is installed in the usual way, with pip:

```
pip install nanostream
```

To test your installation, try typing

```
nanostream --help
```

If NanoStream is installed correctly, you should see a help message.

1.6 Using NanoStream

You use NanoStream by (1) writing a configuration file that describes your pipeline, and (2) running the `nanostream` command, specifying the location of your configuration file. NanoStream will read the configuration, create the pipeline, and run it.

The configuration file is written in YAML. It has three parts:

1. A list of global variables (optional)
2. The nodes and their options (required)
3. A list of edges connecting those nodes to each other.

This is a simple configuration file. If you want to, you can copy it into a file called `sample_config.yaml`:

```
---
pipeline_name: Sample NanoStream configuration
pipeline_description: Reads some environment variables and prints them

nodes:
  get_environment_variables:
    class: GetEnvironmentVariables
    summary: Gets all the necessary environment variables
    options:
      environment_variables:
        - API_KEY
        - API_USER_ID

  print_variables:
    class: PrinterOfThings
    summary: Prints the environment variables to the terminal
    options:
      prepend: "Environment variables: "

paths:
  -
    - get_environment_variables
    - print_variables
```

If you've installed NanoStream and copied this configuration into `sample_config.yaml`, then you can execute the pipeline:

```
nanostream run --filename sample_config.yaml
```

The output should look like this (you might also see some log messages):

```
Environment variables:
{'API_USER_ID': None, 'API_KEY': None}
```

The NanoStream pipeline has found the values of two environment variables (`API_KEY` and `API_USER_ID`) and printed them to the terminal. If those environmet variables have not been set, their values will be `None`. But if you were to set any of them, their values would be printed.

Although this is a very trivial example, it is enough to show the main functionality of NanoStream. Let's look at the configuration file one part at a time.

The configuration starts with two top-level options, `pipeline_name` and `pipeline_description`. These are optional, and are only used for the user's convenience.

Below those are two sections: `nodes` and `paths`. Each `nodes` section contains one or more blocks that always have this form:

```
do_something:
  class: node class
  summary: optional string describing what this node does
  options:
    option_1: value of this option
    option_2: value of another option
```

Let's go through this one line at a time.

Each node block describes a single node in the NanoStream pipeline. A node must be given a name, which can be any arbitrary string. This should be a short, descriptive string describing its action, such as `get_environment_variables` or `parse_json`, for example. We encourage you to stick to a clear naming convention. We like nodes to have names of the form `verb_noun` (as in `print_name`).

NanoStream contains a number of node classes, each of which is designed for a specific type of ETL task. In the sample configuration, we're used the built-in classes `GetEnvironmentVariables` and `PrinterOfThings`; these are the value following `class`. You can also roll your own node classes (we'll describe how to do this later in the documentation).

Next is a set of keys and values for the various options that are supported by that class. Because each node class does something different, the options are different as well. In the sample configuration, the `GetEnvironmentVariables` node class requires a list of environment variables to retrieve, so as you would expect, we specify that list under the `environment_variables` option. The various options are explained in the documentation for each class. In addition to the options that are specific to each node, there are also options that are common to every type of node. These will be explained later.

The structure of the pipeline is given in the `paths` section, which contains a list of lists. Each list is a set of nodes that are to be linked together in order. In our example, the `paths` value says that `get_environment_variables` will send its output to `print_variables`. Paths can be arbitrarily long.

If you wanted to send the environment variables down two different execution paths, you add another list to the `paths`, like so:

```
paths:
  -
    - get_environment_variables
    - print_variables
  -
    - get_environment_variables
    - do_something_else
    - and_then_do_this
```

With this set of `paths`, the pipeline looks like a very simple tree, with `get_environment_variables` at the root, which branches to `print_variables` and `do_something_else`.

When you have written the configuration file, you're ready to use the NanoStream CLI. It accepts a command, followed by some options. As of now, the commands it accepts are `run`, which executes the pipeline, and `draw`, which generates a diagram of the pipeline. The relevant command(s) are:

```
python nanostream_cli.py [run | draw] --filename my_sample_config.yaml
```

The `nanostream` command can generate a pdf file containing a drawing of the pipeline, showing the flow of data through the various nodes. Just speciy `draw` instead of `run` to generate the diagram. For our simple little pipeline, we get this:

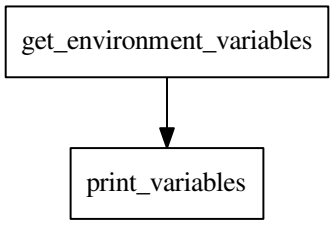

Fig. 3: The pipeline drawing for the simple configuration example

It is also possible to skip using the configuration file and define your pipelines directly in code. In general, it's better to use the configuration file for a variety of reasons, but you always have the option of doing this in Python.

Nodes are defined in code by instantiating classes that inherit from `NanoNode`. Upon instantiation, the constructor takes the same set of keyword arguments as you see in the configuration. Nodes are linked together by the `>` operator, as in `node_1 > node_2`. After the pipeline has been built in this way, it is started by calling `node.global_start()` on any of the nodes in the pipeline.

The code corresponding to the configuration file above would look like this:

```
# Define the nodes using the various subclasses of NanoNode
get_environment_variables =
GetEnvironmentVariables(
    environment_variables=['API_KEY', 'API_USER_ID'])
print_variables = PrinterOfThings(prepend='Environment variables: ')

# The '>' operator can also be chained, as in:
# node_1 > node_2 > node_3 > ...
get_environment_variables > print_variables

# Run the pipeline. This command will not block.
get_environment_variables.global_start()
```

1.7 Rolling your own `NanoNode` class

If there are no built-in `NanoNode` classes suitable for your ETL pipeline, it is easy to write your own.

For example, suppose you want to create a source node for your pipeline that simply emits a user-defined string every few seconds forever. The user would be able to specify the string and the number of seconds to pause after each message has been sent. The class could be defined like so:

```
class FooEmitter(NanoNode):  # inherit from NanoNode
    '''
    Sends ``self.output_string`` every ``self.interval`` seconds.
    '''
    def __init__(self, output_string='', interval=1, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        self.output_string = output_string
        self.interval = interval
        super(FooEmitter, self).__init__()  # Must call the `NanoNode` __init__

    def generator(self):
        while True:
            time.sleep(self.interval)
            yield self.output_string  # Output must be yielded, not returned
```

Let's look at each part of this class.

The first thing to note is that the class inherits from `NanoNode` – this is the mix-in class that gives the node all of its functionality within the NanoStream framework.

The `__init__` method should take only keyword arguments, not positional arguments. This restriction is to guarantee that the configuration files have names for any options that are specified in the pipeline. In the `__init__` function, you should also be sure to accept `**kwargs`, because options that are common to all `NanoNode` objects are expected to be there.

After any attributes have been defined, the `__init__` method **must** invoke the parent class's constructor through the use of the `super` function. Be sure to pass the `**kwargs` argument into the function as shown in the example.

If the node class is intended to be used as a source node, then you need to define a `generator` method. This method can be virtually anything, so long as it sends its output via a `yield` statement.

If you need to define a worker node (that is, a node that accepts input from a queue), you will provide a `process_item` method instead of a `generator`. But the structure of that method is the same, with the single exception that you will have access to a `__message__` attribute which contains the incoming message data. The structure of a typical `process_item` method is shown in the figure.

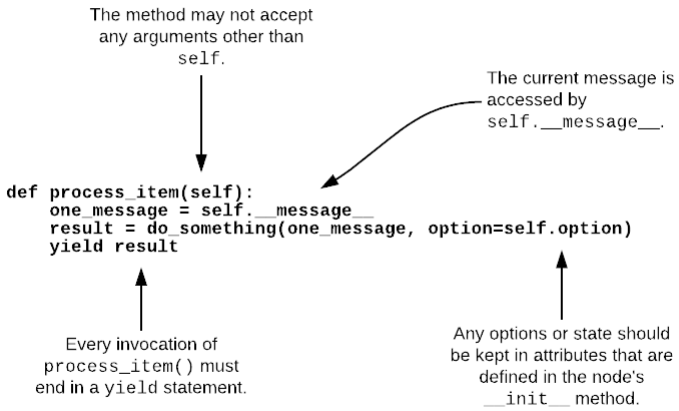

Fig. 4: A typical `process_item` method for `NanoNode` objects

For example, let's suppose you want to create a node that is passed a string as a message, and returns `True` if the message has an even number of characters, `False` otherwise. The class definition would look like this:

```
class MessageLengthTester(NanoNode):
    def __init__(self):
        # No particular initialization required in this example
```

(continues on next page)

(continued from previous page)

```
        super(MessageLengthTester, self).__init__()

    def process_item(self):
        if len(self.__message__) % 2 == 0:
            yield True
        else:
            yield False
```

1.8 Composing and configuring NanoNode objects

Warning: The code described in this section is experimental and very unstable. It would be bad to use it for anything important.

Let's suppose you've worked very hard to create the pipeline from the last example. Now, your boss says that another engineering team wants to use it, but they want to rename parameters and "freeze" the values of certain other parameters to specific values. Once that's done, they want to use it as just one part of a more complicated NanoStream pipeline.

This can be accomplished using a configuration file. When NanoStream parses the configuration file, it will dynamically create the desired class, which can be instantiated and used as if it were a single node in another pipeline.

The configuration file is written in YAML, and it would look like this:

```
name: FooMessageTester

nodes:
  - name: foo_generator
    class FooEmitter
    frozen_arguments:
      message: foobar
    arg_mapping:
      interval: foo_interval
  - name: length_tester
    class: MessageLengthTester
    arg_mapping: null
```

With this file saved as (e.g.) foo_message.yaml, the following code will create a FooMessageTester class and instantiate it:

```
foo_message_config = yaml.load(open('./foo_message.yaml', 'r').read())
class_factory(foo_message_config)
# At this point, there is now a `FooMessageTester` class
foo_node = FooMessageTester(foo_interval=1)
```

You can now use foo_node just as you would any other node. So in order to run it, you just do:

```
foo_node.global_start()
```

Because foo_node is just another node, you can insert it into a larger pipeline and reuse it. For example, suppose that other engineering team wants to add a PrinterOfThings to the end of the pipeline. They'd do this:

```
printer = PrinterOfThings()
foo_node > printer
```

CHAPTER

TWO

THE DATA JOURNEY

2.1 Overview

NanoStream pipelines create dictionary-like objects as messages, and those messages move through the various nodes until they reach a sink. As they move through the nodes, they are modified in one or more of the following ways:

1. New keys and values are added to the dictionary.
2. Keys and values are removed from the dictionary.
3. Values are modified through in-place operations.
4. The structure of the dictionary is changed (e.g. keys are renamed, the dictionary is flattened, and so on).

Which of these operations is used depends on the particular type of node that processes the message, and how the various options are specified for that node.

By default, each time a message is processed, all of its existing keys and values are retained in the message as it's passed to the next node. This behavior is to enable the message to accumulate information over several steps in the pipeline because some nodes require data that is generated by various operations.

Nodes may have access to the entire message, or it is possible to specify which key-value pair is passed to it. This is done by using the `key` option in the node definition. If the node will be generating results to be passed downstream, then we need to either (1) specify the output key for those results; or (2) make sure that the node is generating a dictionary. If (2), then by default the dictionary will be merged into the incoming message, and the combined dictionary will be placed on the node's outgoing queue. If a specific key is specified for the generated data, then we use the `output_key` option in the node definition.

2.2 Example: Making a GET request

Let's consider a very common ETL task. We want to make a GET request to an API endpoint and return the result. The GET request will take a couple of parameters, such as an endpoint name, date and username. For our example, the URL will just be:

```
http://example.api.com/ENDPOINT?date=DATE&username=USERNAME
```

As you would expect, the username and endpoint can be specified in advance; but the date will change each time the pipeline is run. So the date has to be generated at runtime and passed to the node. In our example, the date will be passed to the pipeline as an environment variables `DATE` when the pipeline is executed.

NanoNode contains node classes for reading environment variables and for making GET requests. They are called `GetEnvironmentVariables` and `HttpGetRequest`. They take the following options:

1. `GetEnvironmentVariables`

- `environment_variables`: A list of the names of the environment variables to be fetched. The results will be put in keys named after those environment variables.

2. `HttpGetRequest`

   - `url` (required): The URL for the GET request. Any parameters that will be filled-in at runtime should be put into curly braces. See the example configuration file below.
   - `json` (optional: default `True`) Whether the response should be parsed as JSON.
   - `endpoint_dict` (optional) Keys and values to be substituted into the url.

CHAPTER

THREE

TREEHORN

Treehorn is a set of classes for manipulating dictionary- and list-like objects in a declarative style. It is meant to be useful for the sort of tasks required for ETL, such as extracting structured data from JSON objects.

3.1 Using Treehorn

Treehorn allows you to search for information in a dictionary- or list-like object by specifying conditions. Structures that match those conditions can be returned, or they can be labeled. If they are labeled, you can use those labels to build more complex searches later, or retrieve the data. The style of Treehorn is somewhat like JQuery and similar languages that are good for manipulating tree-like data structures such as web pages.

We'll explain Treehorn by stepping through an example of how we would extract data from the following JSON blob:

```
{
    "source": "users",
    "hash": "Ch8KFgjQj67igOnVto4BELHgwMD7iNfjkQEYlrfjtZAt",
    "events": [
        {
            "appName": "mobileapp",
            "browser": {
                "name": "Google Chrome",
                "version": []
            },
            "duration": 0,
            "created": 1550596005797,
            "location": {
                "country": "United States",
                "state": "Massachusetts",
                "city": "Boston"
            },
            "id": "af6de71b",
            "smtpId": null,
            "portalId": 537105,
            "email": "alice@gmail.com",
            "sentBy": {
                "id": "befa29c9",
                "created": 1550518557458
            },
            "type": "OPEN",
            "filteredEvent": false,
            "deviceType": "COMPUTER"
        },
        {
```

(continues on next page)

(continued from previous page)


```
            "appName": "desktopapp",
            "browser": {
                "name": "Firefox",
                "version": []
            },
            "duration": 0,
            "created": 1550596005389,
            "location": {
                "country": "United States",
                "state": "New York",
                "city": "New York"
            },
            "id": "12aadd80",
            "smtpId": null,
            "portalId": 537105,
            "email": "bob@gmail.com",
            "sentBy": {
                "id": "2cd1e257",
                "created": 1550581974777
            },
            "type": "OPEN",
            "filteredEvent": false,
            "deviceType": "COMPUTER"
        }
    ]
}
```


As you can see, this JSON blob is similar to a typical response from a REST API (in fact, this is actually an example from a real REST API, with all personal information deleted).

Let's suppose you need to extract the email address and corresponding city name for each entry in `events`. This example is simple enough that you might not see the usefulness of Treehorn, but it's complex enough to get a sense of how Treehorn works. Later, we'll look at circumstances where Treehorn's declarative style is especially useful.

There are three kinds of classes that are important for Treehorn:

1. `Conditions` – These are classes that test a particular location in a tree (e.g. a dictionary) for some condition. Examples of useful conditions are being a dictionary with a certain key, being a non-empty list, having an integer value, and so on.
2. `Traversal` – These classes move throughout a tree, recursively applying tests to each node that they visit. Traversals can be upward (toward the root) or downward (toward the leaves).
3. `Label` – These are nothing more than strings that are attached to particular locations in the tree. Typically, we apply a label to locations in the tree that match particular conditions.
4. `Relations` – Finally, this class represents n-tuples of locations in the tree. For example, if an email address is present in the tree, and the user's city is present, a `Relation` can be used to denote that the person with that email address lives in that city.

The workflow for a typical Treehorn query is that we (1) define some conditions (such as being an email address field); (2) traverse the tree, searching for locations that match those conditions; (3) label those locations; and (4) define a relationship from those labels, which we can use to extract the right information. We'll gradually build up a query by adding each of these steps one at a time.

3.1.1 Condition objects

For this example, let's suppose you've loaded the JSON into a dictionary, like so:

```
import json

with open('./sample_api_response.json', 'r') as infile:
    api_response = json.load(infile)
```

Let's extract the email addresses and corresonding cities for each user in the API response. First, we create a couple of `Condition` objects using the built-in class `HasKey`:

```
has_email_key = HasKey('email')
has_city_key = HasKey('city')
```

The `HasKey` class is a subclass of `MeetsCondition`, all of which are callable and return `True` or `False`. For example, you could do the following:

```
d = {'email': 'myemail.com', 'name': 'carol'}
has_email_key(d)  # Returns True
has_city_key(d)   # Returns False
```

What if you want to test for two conditions on a single node? `MeetsCondition` objects can be combined into larger boolean expressions using `&`, `|`, and `~` like so:

```
(has_email_key & has_city_key)(d)     # Returns False
(has_email_key & ~ has_city_key)(d)   # Returns True
(has_email_key | has_city_key)(d)     # Returns True
```

3.1.2 Traversal objects

`MeetsCondition` objects aren't very useful unless they're combined with traversals. There are two types of traversal classes: `GoUp` and `GoDown`. Each takes a `MeetsCondition` object as a parameter. For example, if you want to search from the root of the tree for every location that is a dictionary with the `email` key, the traversal is:

```
find_email = GoDown(condition=has_email_key)  # or GoDown(condition=HasKey('email'))
```

Similarly for finding places with a `city` key:

```
find_city = GoDown(condition=has_city_key)  # or GoDown(condition=HasKey('city'))
```

If you want to retrieve all of `find_city`'s matches, you can use its `matches` method, which will yield each match:

```
for match in has_email_key.matches(api_response):
    print(match)
```

which will yield:

```
{'id': 'af6de71b', 'portalId': 537105, 'location': {'state': 'Massachusetts', 'city':
↪'Boston', 'country': 'United States'}, 'type': 'OPEN', 'sentBy': {'id': 'befa29c9',
↪'created': 1550518557458}, 'appName': 'mobileapp', 'duration': 0,'smtpId': None,
↪'deviceType': 'COMPUTER', 'created': 1550596005797, 'email': 'alice@gmail.com',
↪'browser': {'version': [], 'name': 'Google Chrome'}, 'filteredEvent': False}
{'id': '12aadd80', 'portalId': 537105, 'location': {'state': 'New York', 'city': 'New␣
↪York', 'country': 'United States'}, 'type': 'OPEN', 'sentBy': {'id': '2cd1e257',
↪'created': 1550581974777}, 'appName': 'desktopapp', 'duration': 0, 'smtpId': None,
↪'deviceType': 'COMPUTER', 'created': 1550596005389, 'email': 'bob@gmai
↪'browser': {'version': [], 'name': 'Firefox'}, 'filteredEvent': False}
```

(continues on next page)

(continued from previous page)

Examining each of the dictionaries, we see that they do in fact contain an `email` key. Note that the traversal does **not** return the email string itself – we asked only for the dictionary containing the key. This is by design, as we will see soon.

Because we want to retrieve not only the email addresses but also the cities, we need another traversal. Each of the two dictionaries containing the `email` key also have a subdictionary that contains a `city` key. So we need a second traversal to get that subdictionary. In other words, retrieving the data we need is a two-step process:

1. Starting at the root of the tree, we traverse downward until we find a dictionary with the `email` key.
2. From each of those dictionaries, we go down until we find a dictionary with the `city` key.

Any non-trivial ETL task involving nested dictionary-like objects will require multi-stage traversals like this one. So Treehorn allows you to chain traversals together using the > operator:

```
chained_traversal = find_email > find_city
```

The `chained_traversal` says, in effect, "Go down into the tree and find every node that has an `email` key. Then, from each of those, continue to go down until you find a node that contains a `city` key. In pseudo-code:

```
For each node_1 starting at the root:
    if node_1 has ``email`` key:
        for each node_2 starting at node_1:
            if node_2 has ``city`` key:
                return
```

So far, we have set set up multi-stage searches for nodes in a tree that satisfy various conditions. Next, we have to extract the right data from those searches. This is where the `Label` and `Relation` classes come into play.

3.1.3 Labels

When nodes are identified that satisfy certain conditions, we will want to label those nodes so that we can extract data from them later. The mechanism for doing this is to use a "label".

Continuing the example, let's use the labels "email" and "city" to mark the respective nodes in the two-stage traversal. We do so by adding a label to the traversal chain. Recall that in the previous section, we wrote:

```
chained_traversal = find_email > find_city
```

whereas we now have:

```
chained_traversal = find_email + 'email' > find_city + 'city'
```

We use + to add a label, and the label is just a string. Under the hood, Treehorn is instantiating a `Label` object, but ordinarily, you shouldn't have to do that directly.

3.1.4 Relations

Lastly, we define a `Relation` object to extract the data from our search. In this example, we might think of the search as returning data about people who live in a certain city. So we might name the `Relation` "FROM_CITY".We'll want to extract the value of the `email` key from the node labeled with `email`, and similarly with the `city` node. This is accomplished by adding a little more syntax:

```
Relation('FROM_CITY') == (
    (find_email + 'email')['email'] > (find_city + 'city')['city'])
```

After executing that statement, Treehorn will create an object named `FROM_CITY`, which can be called on a dictionary to yield the information we want, like so:

```
for email_city in FROM_CITY(api_response):
    print(email_city)
```

which will give us:

```
{'city': 'Boston', 'email': 'alice@gmail.com'}
{'city': 'New York', 'email': 'bob@gmail.com'}
```

Voila!

3.2 Summing up

Normally, ETL pipelines that extract data from dictionary-like objects involve a lot of loops and hard-coded keypaths. To accomplish the simple task of extracting emails and city names from our sample JSON blob, we'd probably hard-code paths for each specific key and value, and then we'd loop over various levels in the dictionary. This has several disadvantages:

1. It leads to brittle code. If the JSON blob changes structure in even very small ways, the hard-coded paths become obsolete and have to be rewritten.
2. The code is difficult to understand and debug. Given a whole bunch of nested loops and hard-coded keypaths, it's very difficult to understand the intent of the code. Errors have to be found by painstakingly stepping through the execution.
3. It is very difficult to accommodate JSON blobs with variable structure. Some JSON blobs returned from APIs have unpredictable levels of nesting, for example. Therefore, keypaths cannot be hard-coded and recursive searches have to be written, which are inefficient and difficult to debug.

The approach taken by Treehorn alleviates some of this pain. For example, the `GoDown` traversal doesn't care how many levels down in the tree it must search; so it is often able to cope with inconsistent structures (within reason) without any code changes. It's also much easier to understand. You can tell from glancing at the code that the intention is to search for a dictionary with a key, and then search from there for lower-level dictionaries with another key, and return the results. Treehorn is also more efficient than writing loops and keypaths because all of its evaluations are lazy – it doesn't hold partial results in memory any longer than necessary because everything is yielded by generators.

CHAPTER

FOUR

IMPLEMENTATION

This section describes what's happening under the hood in a `NanoStream` data pipeline. Most people won't need to read this section.

4.1 The data journey

NanoStream pipelines are sets of `NanoNode` objects connected by `NanoStreamQueue` objects. Think of each `NanoNode` as a vertex in a directed graph, and each `NanoStreamQueue` as a directed edge.

There are two types of `NanoNode` objects. A "source" is a `NanoNode` that does not accept incoming data from another `NanoNode`. A "processor" is any `NanoNode` that is not a "source". Note that there is nothing in the class definition or object that distinguishes between these two – the only difference is that processors have a `process_item` method, and sources have a `generator` method. Other than that, they are identical.

The data journey begins with one or more source nodes. When a source node is started (by calling its `start` method), a new thread is created and the node's `generator` method is executed inside the thread. As results from the `generator` method are yielded, they are placed on each outgoing `NanoStreamQueue` to be picked up by one or more processors downstream.

The data from the source's `generator` is handled by the `NanoStreamQueue` object. At its heart, the `NanoStreamQueue` is simply a class which has a Python `Queue.queue` object as an attribute. The reason we don't simply use Python `Queue` objects is because the `NanoStreamQueue` contains some logic that's useful. In particular:

1. It wraps the data into a `NanoStreamMessage` object, which also holds useful metadata including a UUID, the ID of the node that generated the data, and a timestamp.
2. If the `NanoStreamQueue` receives data that is simply a `None` object, then it is skipped.

CHAPTER

FIVE

API DOCUMENTATION

5.1 Node module

The `node` module contains the `NanoNode` class, which is the foundation for NanoStream.

class nanostream.node.**AggregateValues**(*values=False*, *tail_path=None*, ***kwargs*)

Bases: *nanostream.node.NanoNode*

Does that.

process_item()

Default no-op for nodes.

class nanostream.node.**BatchMessages**(*batch_size=None*, *batch_list=None*, *counter=0*, *timeout=5*, ***kwargs*)

Bases: *nanostream.node.NanoNode*

cleanup()

If there is any cleanup (closing files, shutting down database connections), necessary when the node is stopped, then the node's class should provide a `cleanup` method. By default, the method is just a logging statement.

process_item()

Default no-op for nodes.

class nanostream.node.**CSVReader**(**args*, ***kwargs*)

Bases: *nanostream.node.NanoNode*

process_item()

Default no-op for nodes.

class nanostream.node.**CSVToDictionaryList**(***kwargs*)

Bases: *nanostream.node.NanoNode*

process_item()

Default no-op for nodes.

class nanostream.node.**ConstantEmitter**(*thing=None*, *delay=2*, ***kwargs*)

Bases: *nanostream.node.NanoNode*

Send a thing every n seconds

generator()

class nanostream.node.**CounterOfThings**(**args*, *batch=False*, *get_runtime_attrs=<function no_op>*, *get_runtime_attrs_args=None*, *get_runtime_attrs_kwargs=None*, *runtime_attrs_destinations=None*, *input_mapping=None*, *retain_input=True*, *throttle=0*, *keep_alive=True*, *max_errors=0*, *name=None*, *input_message_keypath=None*, *key=None*, *messages_received_counter=0*, *prefer_existing_value=False*, *messages_sent_counter=0*, *post_process_function=None*, *post_process_keypath=None*, *summary=''*, *post_process_function_kwargs=None*, *output_key=None*, ***kwargs*)

Bases: *nanostream.node.NanoNode*

foo__init__(**args*, *start=0*, *end=None*, ***kwargs*)

generator()

Just start counting integers

class nanostream.node.**DynamicClassMediator**(**args*, ***kwargs*)

Bases: *nanostream.node.NanoNode*

get_sink()

get_source()

hi()

sink_list()

source_list()

class nanostream.node.**Filter**(*test=None*, *test_keypath=None*, *value=True*, **args*, ***kwargs*)

Bases: *nanostream.node.NanoNode*

Applies tests to each message and filters out messages that don't pass

Built-in tests: key_exists value_is_true value_is_not_none

Example:

{'test': 'key_exists', 'key': mykey}

process_item()

Default no-op for nodes.

class nanostream.node.**GetEnvironmentVariables**(*mappings=None*, *environment_variables=None*, ***kwargs*)

Bases: *nanostream.node.NanoNode*

generator()

process_item()

Default no-op for nodes.

class nanostream.node.**InsertData**(*overwrite=True*, *overwrite_if_null=True*, *value_dict=None*, ***kwargs*)

Bases: *nanostream.node.NanoNode*

process_item()

Default no-op for nodes.

class nanostream.node.**LocalDirectoryWatchdog**(*directory='.'*, *check_interval=3*, ***kwargs*)
Bases: *nanostream.node.NanoNode*

generator()

class nanostream.node.**LocalFileReader**(**args*, ***kwargs*)
Bases: *nanostream.node.NanoNode*

process_item()
Default no-op for nodes.

class nanostream.node.**NanoNode**(**args*, *batch=False*, *get_runtime_attrs=<function no_op>*, *get_runtime_attrs_args=None*, *get_runtime_attrs_kwargs=None*, *runtime_attrs_destinations=None*, *input_mapping=None*, *retain_input=True*, *throttle=0*, *keep_alive=True*, *max_errors=0*, *name=None*, *input_message_keypath=None*, *key=None*, *messages_received_counter=0*, *prefer_existing_value=False*, *messages_sent_counter=0*, *post_process_function=None*, *post_process_keypath=None*, *summary=''*, *post_process_function_kwargs=None*, *output_key=None*, ***kwargs*)
Bases: `object`

The foundational class of *NanoStream*. This class is inherited by all nodes in a computation graph.

Order of operations: 1. Child class `__init__` function 2. `NanoNode` `__init__` function 3. `preflight_function` (Specified in initialization params) 4. `setup` 5. start

These methods have the following intended uses:

1. `__init__` Sets attribute values and calls the `NanoNode` `__init__` method.
2. `get_runtime_attrs` Sets any attribute values that are to be determined at runtime, e.g. by checking environment variables or reading values from a database. The `get_runtime_attrs` should return a dictionary of attributes -> values, or else `None`.
3. `setup` Sets the state of the `NanoNode` and/or creates any attributes that require information available only at runtime.

Variables

- **send_batch_markers** – If `True`, then a `BatchStart` marker will be sent when a new input is received, and a `BatchEnd` will be sent after the input has been processed. The intention is that a number of items will be emitted for each input received. For example, we might emit a table row-by-row for each input.
- **get_runtime_attrs** – A function that returns a dictionary-like object. The keys and values will be saved to this `NanoNode` object's attributes. The function is executed one time, upon starting the node.
- **get_runtime_attrs_args** – A tuple of arguments to be passed to the `get_runtime_attrs` function upon starting the node.
- **get_runtime_attrs_kwargs** – A dictionary of kwargs passed to the `get_runtime_attrs` function.
- **runtime_attrs_destinations** – If set, this is a dictionary mapping the keys returned from the `get_runtime_attrs` function to the names of the attributes to which the values will be saved.
- **throttle** – For each input received, a delay of `throttle` seconds will be added.

- **keep_alive** – If `True`, keep the node's thread alive after everything has been processed.
- **name** – The name of the node. Defaults to a randomly generated hash. Note that this hash is not consistent from one run to the next.
- **input_mapping** – When the node receives a dictionary-like object, this dictionary will cause the keys of the dictionary to be remapped to new keys.
- **retain_input** – If `True`, then combine the dictionary-like input with the output. If keys clash, the output value will be kept.
- **input_message_keypath** – Read the value in this keypath as the content of the incoming message.

add_edge(*target*, ***kwargs*)
Create an edge connecting *self* to *target*.

This method instantiates the `NanoStreamQueue` object that connects the nodes. Connecting the nodes together consists in (1) adding the queue to the other's `input_queue_list` or `output_queue_list` and (2) setting the queue's `source_node` and `target_node` attributes.

Args: target (`NanoNode`): The node to which `self` will be connected.

Returns: None

all_connected(*seen=None*)
Returns all the nodes connected (directly or indirectly) to `self`. This allows us to loop over all the nodes in a pipeline even if we have a handle on only one. This is used by `global_start`, for example.

Args:

seen (set): A set of all the nodes that have been identified as connected to `self`.

Returns:

(set of `NanoNode`): All the nodes connected to `self`. This includes `self`.

broadcast(*broadcast_message*)
Puts the message into all the input queues for all connected nodes.

cleanup()
If there is any cleanup (closing files, shutting down database connections), necessary when the node is stopped, then the node's class should provide a `cleanup` method. By default, the method is just a logging statement.

draw_pipeline()
Draw the pipeline structure using graphviz.

global_start(*datadog=False*, *prometheus=False*, *pipeline_name=None*)
Starts every node connected to `self`. Mainly, it:

1. calls `start()` on each node
2. sets some global variables
3. optionally starts some experimental code for monitoring

input_queue_size
Return the total number of items in all of the queues that are inputs to this node.

is_sink
Tests whether the node is a sink or not, i.e. whether there are no outputs from the node.

Returns: (bool): `True` if the node has no output nodes, `False` otherwise.

is_source
Tests whether the node is a source or not, i.e. whether there are no inputs to the node.

Returns: (bool): `True` if the node has no inputs, `False` otherwise.

kill_pipeline()

log_info(*message=''*)

logjam
Returns the logjam score, which measures the degree to which the node is holding up progress in downstream nodes.

We're defining a logjam as a node whose input queue is full, but whose output queue(s) is not. More specifically, we poll each node in the `monitor_thread`, and increment a counter if the node is a logjam at that time. This property returns the percentage of samples in which the node is a logjam. Our intention is that if this score exceeds a threshold, the user is alerted, or the load is rebalanced somehow (not yet implemented).

Returns: (float): Logjam score

process_item(**args*, ***kwargs*)
Default no-op for nodes.

processor_bak()
This calls the user's `process_item` with just the message content, and then returns the full message.

I think this is deprecated, which is why it's been renamed to `processor_bak`.

setup()
For classes that require initialization at runtime, which can't be done when the class's `__init__` function is called. The `NanoNode` base class's setup function is just a logging call.

It should be unusual to have to make use of `setup` because in practice, initialization can be done in the `__init__` function.

start()
Starts the node. This is called by `NanoNode.global_start()`.

The node's main loop is contained in this method. The main loop does the following:

1. records the timestamp to the node's `started_at` attribute.
2. calls `get_runtime_attrs` (TODO: check if we can deprecate this)
3. calls the `setup` method for the class (which is a no-op by default)
4. if the node is a source, then successively yield all the results of the node's `generator` method, then exit.
5. if the node is not a source, then loop over the input queues, getting the next message. Note that when the message is pulled from the queue, the `NanoStreamQueue` yields it as a dictionary.
6. gets either the content of the entire message if the node has no `key` attribute, or the value of `message[self.key]`.
7. remaps the message content if a `remapping` dictionary has been given in the node's configuration
8. calls the node's `process_item` method, yielding back the results. (Note that a single input message may cause the node to yield zero, one, or more than one output message.)
9. places the results into each of the node's output queues.

stream()
Called in each `NanoNode` thread.

terminate_pipeline(*error=False*)
This method can be called on any node in a pipeline, and it will cause all of the nodes to terminate if they haven't stopped already.

Args: error (bool): Not yet implemented.

thread_monitor()
This function loops over all of the threads in the pipeline, checking that they are either `finished` or `running`. If any have had an abnormal exit, terminate the entire pipeline.

time_running
Return the number of wall-clock seconds elapsed since the node was started.

class nanostream.node.**NothingToSeeHere**
Bases: `object`

Vacuous class used as a no-op message type.

class nanostream.node.**PrinterOfThings**(**args*, ***kwargs*)
Bases: *nanostream.node.NanoNode*

process_item()
Default no-op for nodes.

class nanostream.node.**RandomSample**(*sample=0.1*)
Bases: *nanostream.node.NanoNode*

Lets through only a random sample of incoming messages. Might be useful for testing, or when only approximate results are necessary.

process_item()
Default no-op for nodes.

class nanostream.node.**Remapper**(*mapping=None*, ***kwargs*)
Bases: *nanostream.node.NanoNode*

process_item()
Default no-op for nodes.

class nanostream.node.**SequenceEmitter**(*sequence*, **args*, *max_sequences=1*, ***kwargs*)
Bases: *nanostream.node.NanoNode*

Emits `sequence` `max_sequences` times, or forever if `max_sequences` is `None`.

generator()
Emit the sequence `max_sequences` times.

process_item()
Emit the sequence `max_sequences` times.

class nanostream.node.**Serializer**(*values=False*, **args*, ***kwargs*)
Bases: *nanostream.node.NanoNode*

Takes an iterable thing as input, and successively yields its items.

process_item()
Default no-op for nodes.

class nanostream.node.**SimpleTransforms**(*missing_keypath_action='ignore'*, *starting_path=None*, *transform_mapping=None*, *target_value=None*, *keypath=None*, ***kwargs*)
Bases: *nanostream.node.NanoNode*

process_item()
Default no-op for nodes.

class nanostream.node.**StreamMySQLTable**(**args*, *host='localhost'*, *user=None*, *table=None*, *password=None*, *database=None*, *port=3306*, *to_row_obj=False*, *send_batch_markers=True*, ***kwargs*)

Bases: *nanostream.node.NanoNode*

generator()

get_schema()

setup()

For classes that require initialization at runtime, which can't be done when the class's `__init__` function is called. The `NanoNode` base class's setup function is just a logging call.

It should be unusual to have to make use of `setup` because in practice, initialization can be done in the `__init__` function.

class nanostream.node.**StreamingJoin**(*window=30*, *streams=None*, **args*, ***kwargs*)

Bases: *nanostream.node.NanoNode*

Joins two streams on a key, using exact match only. MVP.

process_item()

class nanostream.node.**SubstituteRegex**(*match_regex=None*, *substitute_string=None*, **args*, ***kwargs*)

Bases: *nanostream.node.NanoNode*

process_item()

Default no-op for nodes.

class nanostream.node.**TimeWindowAccumulator**(**args*, ***kwargs*)

Bases: *nanostream.node.NanoNode*

Every N seconds, put the latest M seconds data on the queue.

class nanostream.node.**bcolors**

Bases: `object`

This class holds the values for the various colors that are used in the tables that monitor the status of the nodes.

BOLD = '\x1b[1m'

ENDC = '\x1b[0m'

FAIL = '\x1b[91m'

HEADER = '\x1b[95m'

OKBLUE = '\x1b[94m'

OKGREEN = '\x1b[92m'

UNDERLINE = '\x1b[4m'

WARNING = '\x1b[93m'

nanostream.node.**class_factory**(*raw_config*)

nanostream.node.**get_environment_variables**(**args*)

Retrieves the environment variables listed in `*args`.

Args: args (list of str): List of environment variables.

Returns:

dict: Dictionary of environment variables to values. If the environment variable is not defined, the value is `None`.

`nanostream.node.`**`get_node_dict`**(*node_config*)

`nanostream.node.`**`kwarg_remapper`**(*f*, ***kwarg_mapping*)

`nanostream.node.`**`no_op`**(**args*, ***kwargs*)
No-op function to serve as default `get_runtime_attrs`.

`nanostream.node.`**`template_class`**(*class_name*, *parent_class*, *kwargs_remapping*, *frozen_arguments_mapping*)

5.2 Civis-specific node types

This is where any classes specific to the Civis API live.

class `nanostream.civis_nodes.`**`CivisSQLExecute`**(**args*, *sql=None*, *civis_api_key=None*, *civis_api_key_env_var='CIVIS_API_KEY'*, *database=None*, *dummy_run=False*, *query_dict=None*, *returned_columns=None*, ***kwargs*)

Bases: *nanostream.node.NanoNode*

Execute a SQL statement and return the results.

`process_item`()
Execute a SQL statement and return the result.

class `nanostream.civis_nodes.`**`CivisToCSV`**(**args*, *sql=None*, *civis_api_key=None*, *civis_api_key_env_var='CIVIS_API_KEY'*, *database=None*, *dummy_run=False*, *query_dict=None*, *returned_columns=None*, *include_headers=True*, *delimiter=','*, ***kwargs*)

Bases: *nanostream.node.NanoNode*

Execute a SQL statement and return the results via a CSV file.

`process_item`()
Execute a SQL statement and return the result.

class `nanostream.civis_nodes.`**`EnsureCivisRedshiftTableExists`**(*on_failure='exit'*, *table=None*, *schema=None*, *columns=None*, *block=True*, ***kwargs*)

Bases: *nanostream.node.NanoNode*

`generator`()

`process_item`()
Default no-op for nodes.

class `nanostream.civis_nodes.`**`FindValueInRedshiftColumn`**(*on_failure='exit'*, *table=None*, *database=None*, *schema=None*, *column=None*, *choice='max'*, ***kwargs*)

Bases: *nanostream.node.NanoNode*

generator()

process_item()
Default no-op for nodes.

class nanostream.civis_nodes.**SendToCivis**(**args*, *civis_api_key=None*, *civis_api_key_env_var='CIVIS_API_KEY'*, *database=None*, *schema=None*, *existing_table_rows='append'*, *include_columns=None*, *dummy_run=False*, *block=False*, *max_errors=0*, *table=None*, *columns=None*, *remap=None*, *recorded_tables={}*, ***kwargs*)
Bases: *nanostream.node.NanoNode*

monitor_futures()

process_item()
Accept a bunch of dictionaries mapping column names to values.

setup()
Not sure if we'll need this. We could get a client and pass it around.

5.3 Data structures module

Data types (e.g. Rows, Records) for ETL.

class nanostream.utils.data_structures.**BOOL**(*value*, *original_type=None*, *name=None*)
Bases: *nanostream.utils.data_structures.DataType*, *nanostream.utils.data_structures.IntermediateTypeSystem*

python_cast_function
alias of builtins.bool

class nanostream.utils.data_structures.**DATETIME**(*value*, *original_type=None*, *name=None*)
Bases: *nanostream.utils.data_structures.DataType*, *nanostream.utils.data_structures.IntermediateTypeSystem*

python_cast_function()

class nanostream.utils.data_structures.**DataSourceTypeSystem**
Bases: object

Information about mapping one type system onto another contained in the children of this class.

static convert(*obj*)
Override this method if something more complicated is necessary.

static type_mapping(**args*, ***kwargs*)

class nanostream.utils.data_structures.**DataType**(*value*, *original_type=None*, *name=None*)
Bases: object

Each DataType gets a python_cast_function, which is a function.

intermediate_type = None

python_cast_function = None

to_intermediate_type()
Convert the DataType to an IntermediateDataType using its class's intermediate_type attribute.

to_python()

type_system
Just for convenience to make the type system an attribute.

class nanostream.utils.data_structures.**FLOAT**(*value*, *original_type=None*, *name=None*)
Bases: *nanostream.utils.data_structures.DataType*, *nanostream.utils.data_structures.IntermediateTypeSystem*

python_cast_function
alias of builtins.float

class nanostream.utils.data_structures.**INTEGER**(*value*, *original_type=None*, *name=None*)
Bases: *nanostream.utils.data_structures.DataType*, *nanostream.utils.data_structures.IntermediateTypeSystem*

python_cast_function
alias of builtins.int

exception nanostream.utils.data_structures.**IncompatibleTypesException**
Bases: Exception

class nanostream.utils.data_structures.**IntermediateTypeSystem**
Bases: *nanostream.utils.data_structures.DataSourceTypeSystem*

Never instantiate this by hand.

class nanostream.utils.data_structures.**MYSQL_BOOL**(*value*, *original_type=None*, *name=None*)
Bases: *nanostream.utils.data_structures.DataType*, *nanostream.utils.data_structures.MySQLTypeSystem*

intermediate_type
alias of *BOOL*

python_cast_function
alias of builtins.bool

class nanostream.utils.data_structures.**MYSQL_DATE**(*value*, *original_type=None*, *name=None*)
Bases: *nanostream.utils.data_structures.DataType*, *nanostream.utils.data_structures.MySQLTypeSystem*

intermediate_type
alias of *DATETIME*

python_cast_function()

class nanostream.utils.data_structures.**MYSQL_ENUM**(*value*, *original_type=None*, *name=None*)
Bases: *nanostream.utils.data_structures.DataType*, *nanostream.utils.data_structures.MySQLTypeSystem*

intermediate_type
alias of *STRING*

python_cast_function
alias of builtins.str

class nanostream.utils.data_structures.**MYSQL_INTEGER**
Bases: type

class nanostream.utils.data_structures.**MYSQL_INTEGER0**(*value*, *original_type=None*, *name=None*)
Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 0

class nanostream.utils.data_structures.**MYSQL_INTEGER1**(*value*, *original_type=None*, *name=None*)
Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 1

class nanostream.utils.data_structures.**MYSQL_INTEGER10**(*value*, *original_type=None*, *name=None*)
Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 10

class nanostream.utils.data_structures.**MYSQL_INTEGER1024**(*value*, *original_type=None*, *name=None*)
Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 1024

class nanostream.utils.data_structures.**MYSQL_INTEGER11**(*value*, *original_type=None*, *name=None*)
Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 11

class nanostream.utils.data_structures.**MYSQL_INTEGER12**(*value*, *original_type=None*, *name=None*)
Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 12

class nanostream.utils.data_structures.**MYSQL_INTEGER128**(*value*, *original_type=None*, *name=None*)
Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 128

class nanostream.utils.data_structures.**MYSQL_INTEGER13**(*value*, *original_type=None*, *name=None*)
Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 13

class nanostream.utils.data_structures.**MYSQL_INTEGER14**(*value*, *original_type=None*, *name=None*)
Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 14

class nanostream.utils.data_structures.**MYSQL_INTEGER15**(*value*, *original_type=None*, *name=None*)
Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 15

class nanostream.utils.data_structures.**MYSQL_INTEGER16**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 16

class nanostream.utils.data_structures.**MYSQL_INTEGER16384**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 16384

class nanostream.utils.data_structures.**MYSQL_INTEGER17**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 17

class nanostream.utils.data_structures.**MYSQL_INTEGER18**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 18

class nanostream.utils.data_structures.**MYSQL_INTEGER19**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 19

class nanostream.utils.data_structures.**MYSQL_INTEGER2**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 2

class nanostream.utils.data_structures.**MYSQL_INTEGER20**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 20

class nanostream.utils.data_structures.**MYSQL_INTEGER2048**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 2048

class nanostream.utils.data_structures.**MYSQL_INTEGER21**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 21

class nanostream.utils.data_structures.**MYSQL_INTEGER22**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 22

class nanostream.utils.data_structures.**MYSQL_INTEGER23**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 23

class nanostream.utils.data_structures.**MYSQL_INTEGER24**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 24

class nanostream.utils.data_structures.**MYSQL_INTEGER25**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 25

class nanostream.utils.data_structures.**MYSQL_INTEGER256**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 256

class nanostream.utils.data_structures.**MYSQL_INTEGER26**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 26

class nanostream.utils.data_structures.**MYSQL_INTEGER27**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 27

class nanostream.utils.data_structures.**MYSQL_INTEGER28**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 28

class nanostream.utils.data_structures.**MYSQL_INTEGER29**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 29

class nanostream.utils.data_structures.**MYSQL_INTEGER3**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 3

class nanostream.utils.data_structures.**MYSQL_INTEGER30**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 30

class nanostream.utils.data_structures.**MYSQL_INTEGER31**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 31

class nanostream.utils.data_structures.**MYSQL_INTEGER32**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 32

class nanostream.utils.data_structures.**MYSQL_INTEGER32768**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 32768

class nanostream.utils.data_structures.**MYSQL_INTEGER4**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 4

class nanostream.utils.data_structures.**MYSQL_INTEGER4096**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 4096

class nanostream.utils.data_structures.**MYSQL_INTEGER5**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 5

class nanostream.utils.data_structures.**MYSQL_INTEGER512**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 512

class nanostream.utils.data_structures.**MYSQL_INTEGER6**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 6

class nanostream.utils.data_structures.**MYSQL_INTEGER64**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 64

class nanostream.utils.data_structures.**MYSQL_INTEGER7**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 7

class nanostream.utils.data_structures.**MYSQL_INTEGER8**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 8

class nanostream.utils.data_structures.**MYSQL_INTEGER8192**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 8192

class nanostream.utils.data_structures.**MYSQL_INTEGER9**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 9

class nanostream.utils.data_structures.**MYSQL_INTEGER_BASE**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.DataType*, *nanostream.utils.data_structures.MySQLTypeSystem*

intermediate_type
alias of *INTEGER*

python_cast_function
alias of builtins.int

class nanostream.utils.data_structures.**MYSQL_VARCHAR**

Bases: type

class nanostream.utils.data_structures.**MYSQL_VARCHAR0**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 0

class nanostream.utils.data_structures.**MYSQL_VARCHAR1**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 1

class nanostream.utils.data_structures.**MYSQL_VARCHAR10**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 10

class nanostream.utils.data_structures.**MYSQL_VARCHAR1024**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 1024

class nanostream.utils.data_structures.**MYSQL_VARCHAR11**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 11

class nanostream.utils.data_structures.**MYSQL_VARCHAR12**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 12

class nanostream.utils.data_structures.**MYSQL_VARCHAR128**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 128

class nanostream.utils.data_structures.**MYSQL_VARCHAR13**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 13

class nanostream.utils.data_structures.**MYSQL_VARCHAR14**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 14

class nanostream.utils.data_structures.**MYSQL_VARCHAR15**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 15

class nanostream.utils.data_structures.**MYSQL_VARCHAR16**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 16

class nanostream.utils.data_structures.**MYSQL_VARCHAR16384**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 16384

class nanostream.utils.data_structures.**MYSQL_VARCHAR17**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 17

class nanostream.utils.data_structures.**MYSQL_VARCHAR18**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 18

class nanostream.utils.data_structures.**MYSQL_VARCHAR19**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 19

class nanostream.utils.data_structures.**MYSQL_VARCHAR2**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 2

class nanostream.utils.data_structures.**MYSQL_VARCHAR20**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 20

class nanostream.utils.data_structures.**MYSQL_VARCHAR2048**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 2048

class nanostream.utils.data_structures.**MYSQL_VARCHAR21**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 21

class nanostream.utils.data_structures.**MYSQL_VARCHAR22**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 22

class nanostream.utils.data_structures.**MYSQL_VARCHAR23**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 23

class nanostream.utils.data_structures.**MYSQL_VARCHAR24**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 24

class nanostream.utils.data_structures.**MYSQL_VARCHAR25**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 25

class nanostream.utils.data_structures.**MYSQL_VARCHAR256**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 256

class nanostream.utils.data_structures.**MYSQL_VARCHAR26**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 26

class nanostream.utils.data_structures.**MYSQL_VARCHAR27**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 27

class nanostream.utils.data_structures.**MYSQL_VARCHAR28**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 28

class nanostream.utils.data_structures.**MYSQL_VARCHAR29**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 29

class nanostream.utils.data_structures.**MYSQL_VARCHAR3**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 3

class nanostream.utils.data_structures.**MYSQL_VARCHAR30**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 30

class nanostream.utils.data_structures.**MYSQL_VARCHAR31**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 31

class nanostream.utils.data_structures.**MYSQL_VARCHAR32**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 32

class nanostream.utils.data_structures.**MYSQL_VARCHAR32768**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 32768

class nanostream.utils.data_structures.**MYSQL_VARCHAR4**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 4

class nanostream.utils.data_structures.**MYSQL_VARCHAR4096**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 4096

class nanostream.utils.data_structures.**MYSQL_VARCHAR5**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 5

class nanostream.utils.data_structures.**MYSQL_VARCHAR512**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 512

class nanostream.utils.data_structures.**MYSQL_VARCHAR6**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 6

class nanostream.utils.data_structures.**MYSQL_VARCHAR64**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 64

class nanostream.utils.data_structures.**MYSQL_VARCHAR7**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 7

class nanostream.utils.data_structures.**MYSQL_VARCHAR8**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 8

class nanostream.utils.data_structures.**MYSQL_VARCHAR8192**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 8192

class nanostream.utils.data_structures.**MYSQL_VARCHAR9**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 9

class nanostream.utils.data_structures.**MYSQL_VARCHAR_BASE**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.DataType*, *nanostream.utils.data_structures.MySQLTypeSystem*

intermediate_type
alias of *STRING*

python_cast_function
alias of builtins.str

class nanostream.utils.data_structures.**MySQLTypeSystem**

Bases: *nanostream.utils.data_structures.DataSourceTypeSystem*

Each TypeSystem gets a type_mapping static method that takes a string and returns the class in the type system named by that string. For example, int(8) in a MySQL schema should return the MYSQL_INTEGER8 class.

static type_mapping(*string*)
Parses the schema strings from MySQL and returns the appropriate class.

class nanostream.utils.data_structures.**PrimitiveTypeSystem**

Bases: *nanostream.utils.data_structures.DataSourceTypeSystem*

class nanostream.utils.data_structures.**PythonTypeSystem**

Bases: *nanostream.utils.data_structures.DataSourceTypeSystem*

class nanostream.utils.data_structures.**Row**(**records*, *type_system=None*)

Bases: object

A collection of DataType objects (typed values). They are dictionaries mapping the names of the values to the DataType objects.

concat(*other*, *fail_on_duplicate=True*)

static from_dict(*row_dictionary*, ***kwargs*)
Creates a Row object form a dictionary mapping names to values.

is_empty()

keys()
For implementing the mapping protocol.

class nanostream.utils.data_structures.**STRING**(*value*, *original_type=None*, *name=None*)

Bases: *nanostream.utils.data_structures.DataType*, *nanostream.utils.data_structures.IntermediateTypeSystem*

python_cast_function
alias of builtins.str

nanostream.utils.data_structures.**all_bases**(*obj*)
Return all the class to which obj belongs.

nanostream.utils.data_structures.**convert_to_type_system**(*obj*, *cls*)

nanostream.utils.data_structures.**get_type_system**(*obj*)

nanostream.utils.data_structures.**make_types**()

nanostream.utils.data_structures.**mysql_type**(*string*)
Parses the schema strings from MySQL and returns the appropriate class.

nanostream.utils.data_structures.**primitive_to_intermediate_type**(*thing*, *name=None*)

5.4 Network nodes module

Classes that deal with sending and receiving data across the interwebs.

class nanostream.node_classes.network_nodes.**HttpGetRequest**(*endpoint_template=None*, *endpoint_dict=None*, *protocol='http'*, *retries=5*, *json=True*, ***kwargs*)

Bases: *nanostream.node.NanoNode*

Node class for making simple GET requests.

process_item()
The input to this function will be a dictionary-like object with parameters to be substituted into the endpoint string and a dictionary with keys and values to be passed in the GET request.

Three use-cases: 1. Endpoint and parameters set initially and never changed. 2. Endpoint and parameters set once at runtime 3. Endpoint and parameters set by upstream messages

class nanostream.node_classes.network_nodes.**HttpGetRequestPaginator**(*endpoint_dict=None*, *json=True*, *pagination_get_request_key=None*, *endpoint_template=None*, *additional_data_key=None*, *pagination_key=None*, *pagination_template_key=None*, *default_offset_value=''*, ***kwargs*)

Bases: *nanostream.node.NanoNode*

Node class for HTTP API requests that require paging through sets of results.

process_item()
Default no-op for nodes.

class nanostream.node_classes.network_nodes.**PaginatedHttpGetRequest**(*endpoint_template=None*, *additional_data_key=None*, *pagination_key=None*, *pagination_get_request_key=None*, *protocol='http'*, *retries=5*, *default_offset_value=''*, *additional_data_test=<class 'bool'>*)

Bases: object

For handling requests in a semi-general way that require paging through lists of results and repeatedly making GET requests.

responses()

Generator. Yields each response until empty.

5.5 NanoStreamMessage module

The NanoStreamMesaage encapsulates the content of each piece of data, along with some useful metadata.

class nanostream.message.message.**NanoStreamMessage**(*message_content*)

Bases: object

A class that contains the message payloads that are queued for each NanoStreamProcessor. It holds the messages and lots of metadata used for logging, monitoring, etc.

5.6 PoisonPill module

A simple class that is sent in a message to signal that the node should be terminated.

class nanostream.message.poison_pill.**PoisonPill**

Bases: object

5.7 Trigger module

A simple class containing no data, which is intended merely as a trigger, signaling that the downstream node should do something.

class nanostream.message.trigger.**Trigger**(*previous_trigger_time=None*, *trigger_name=None*)

Bases: object

nanostream.message.trigger.**hello_world**()

5.8 Batch module

We'll use markers to delimit batches of things, such as serialized files and that kind of thing.

class nanostream.message.batch.**BatchEnd**(**args*, ***kwargs*)
Bases: object

class nanostream.message.batch.**BatchStart**(**args*, ***kwargs*)
Bases: object

class nanostream.message.canary.**Canary**
Bases: object

5.9 NanoStreamQueue module

These are queues that form the directed edges between nodes.

class nanostream.node_queue.queue.**NanoStreamQueue**(*max_queue_size*, *name=None*)
Bases: object

approximately_full(*error=0.95*)

empty

get()

put(*message*, **args*, *previous_message=None*, ***kwargs*)
Places a message on the output queues. If the message is None, then the queue is skipped.

Messages are NanoStreamMessage objects; the payload of the message is message.message_content.

size()

CHAPTER

SIX

LICENSE

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

n

nanostream.civis_nodes, 28
nanostream.message.batch, 41
nanostream.message.canary, 42
nanostream.message.message, 41
nanostream.message.poison_pill, 41
nanostream.message.trigger, 41
nanostream.node, 21
nanostream.node_classes.network_nodes, 40
nanostream.node_queue.queue, 42
nanostream.utils.data_structures, 29

INDEX

A

add_edge() (nanostream.node.NanoNode method), 24
AggregateValues (class in nanostream.node), 21
all_bases() (in module nanostream.utils.data_structures), 39
all_connected() (nanostream.node.NanoNode method), 24
approximately_full() (nanostream.node_queue.queue.NanoStreamQueue method), 42

B

BatchEnd (class in nanostream.message.batch), 42
BatchMessages (class in nanostream.node), 21
BatchStart (class in nanostream.message.batch), 42
bcolors (class in nanostream.node), 27
BOLD (nanostream.node.bcolors attribute), 27
BOOL (class in nanostream.utils.data_structures), 29
broadcast() (nanostream.node.NanoNode method), 24

C

Canary (class in nanostream.message.canary), 42
CivisSQLExecute (class in nanostream.civis_nodes), 28
CivisToCSV (class in nanostream.civis_nodes), 28
class_factory() (in module nanostream.node), 27
cleanup() (nanostream.node.BatchMessages method), 21
cleanup() (nanostream.node.NanoNode method), 24
concat() (nanostream.utils.data_structures.Row method), 39
ConstantEmitter (class in nanostream.node), 21
convert() (nanostream.utils.data_structures.DataSourceTypeSystem static method), 29
convert_to_type_system() (in module nanostream.utils.data_structures), 40
CounterOfThings (class in nanostream.node), 21
CSVReader (class in nanostream.node), 21
CSVToDictionaryList (class in nanostream.node), 21

D

DataSourceTypeSystem (class in nanostream.utils.data_structures), 29
DataType (class in nanostream.utils.data_structures), 29
DATETIME (class in nanostream.utils.data_structures), 29
draw_pipeline() (nanostream.node.NanoNode method), 24
DynamicClassMediator (class in nanostream.node), 22

E

empty (nanostream.node_queue.queue.NanoStreamQueue attribute), 42
ENDC (nanostream.node.bcolors attribute), 27
EnsureCivisRedshiftTableExists (class in nanostream.civis_nodes), 28

F

FAIL (nanostream.node.bcolors attribute), 27
Filter (class in nanostream.node), 22
FindValueInRedshiftColumn (class in nanostream.civis_nodes), 28
FLOAT (class in nanostream.utils.data_structures), 30
foo__init__() (nanostream.node.CounterOfThings method), 22
from_dict() (nanostream.utils.data_structures.Row static method), 39

G

generator() (nanostream.civis_nodes.EnsureCivisRedshiftTableExists method), 28
generator() (nanostream.civis_nodes.FindValueInRedshiftColumn method), 28
generator() (nanostream.node.ConstantEmitter method), 21
generator() (nanostream.node.CounterOfThings method), 22
generator() (nanostream.node.GetEnvironmentVariables method), 22
generator() (nanostream.node.LocalDirectoryWatchdog method), 23
generator() (nanostream.node.SequenceEmitter method), 26
generator() (nanostream.node.StreamMySQLTable method), 27

get() (nanostream.node_queue.queue.NanoStreamQueue method), 42
get_environment_variables() (in module nanostream.node), 27
get_node_dict() (in module nanostream.node), 28
get_schema() (nanostream.node.StreamMySQLTable method), 27
get_sink() (nanostream.node.DynamicClassMediator method), 22
get_source() (nanostream.node.DynamicClassMediator method), 22
get_type_system() (in module nanostream.utils.data_structures), 40
GetEnvironmentVariables (class in nanostream.node), 22
global_start() (nanostream.node.NanoNode method), 24

H

HEADER (nanostream.node.bcolors attribute), 27
hello_world() (in module nanostream.message.trigger), 41
hi() (nanostream.node.DynamicClassMediator method), 22
HttpGetRequest (class in nanostream.node_classes.network_nodes), 40
HttpGetRequestPaginator (class in nanostream.node_classes.network_nodes), 40

I

IncompatibleTypesException, 30
input_queue_size (nanostream.node.NanoNode attribute), 24
InsertData (class in nanostream.node), 22
INTEGER (class in nanostream.utils.data_structures), 30
intermediate_type (nanostream.utils.data_structures.DataType attribute), 29
intermediate_type (nanostream.utils.data_structures.MYSQL_BOOL attribute), 30
intermediate_type (nanostream.utils.data_structures.MYSQL_DATE attribute), 30
intermediate_type (nanostream.utils.data_structures.MYSQL_ENUM attribute), 30
intermediate_type (nanostream.utils.data_structures.MYSQL_INTEGER_BASE attribute), 35
intermediate_type (nanostream.utils.data_structures.MYSQL_VARCHAR_BASE attribute), 39
IntermediateTypeSystem (class in nanostream.utils.data_structures), 30
is_empty() (nanostream.utils.data_structures.Row method), 39
is_sink (nanostream.node.NanoNode attribute), 24
is_source (nanostream.node.NanoNode attribute), 24

K

keys() (nanostream.utils.data_structures.Row method), 39
kill_pipeline() (nanostream.node.NanoNode method), 25
kwarg_remapper() (in module nanostream.node), 28

L

LocalDirectoryWatchdog (class in nanostream.node), 22
LocalFileReader (class in nanostream.node), 23
log_info() (nanostream.node.NanoNode method), 25
logjam (nanostream.node.NanoNode attribute), 25

M

make_types() (in module nanostream.utils.data_structures), 40
max_length (nanostream.utils.data_structures.MYSQL_INTEGER0 attribute), 31
max_length (nanostream.utils.data_structures.MYSQL_INTEGER1 attribute), 31
max_length (nanostream.utils.data_structures.MYSQL_INTEGER10 attribute), 31
max_length (nanostream.utils.data_structures.MYSQL_INTEGER1024 attribute), 31
max_length (nanostream.utils.data_structures.MYSQL_INTEGER11 attribute), 31
max_length (nanostream.utils.data_structures.MYSQL_INTEGER12 attribute), 31
max_length (nanostream.utils.data_structures.MYSQL_INTEGER128 attribute), 31
max_length (nanostream.utils.data_structures.MYSQL_INTEGER13 attribute), 31
max_length (nanostream.utils.data_structures.MYSQL_INTEGER14 attribute), 31
max_length (nanostream.utils.data_structures.MYSQL_INTEGER15 attribute), 31
max_length (nanostream.utils.data_structures.MYSQL_INTEGER16 attribute), 32
max_length (nanostream.utils.data_structures.MYSQL_INTEGER16384 attribute), 32
max_length (nanostream.utils.data_structures.MYSQL_INTEGER17 attribute), 32
max_length (nanostream.utils.data_structures.MYSQL_INTEGER18 attribute), 32
max_length (nanostream.utils.data_structures.MYSQL_INTEGER19 attribute), 32
max_length (nanostream.utils.data_structures.MYSQL_INTEGER2 attribute), 32
max_length (nanostream.utils.data_structures.MYSQL_INTEGER20 attribute), 32

max_length (nanostream.utils.data_structures.MYSQL_INTEGER_2048 attribute), 32
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_21 attribute), 32
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_22 attribute), 32
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_23 attribute), 32
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_24 attribute), 33
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_25 attribute), 33
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_256 attribute), 33
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_26 attribute), 33
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_27 attribute), 33
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_28 attribute), 33
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_29 attribute), 33
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_3 attribute), 33
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_30 attribute), 33
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_31 attribute), 33
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_32 attribute), 33
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_32768 attribute), 34
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_4 attribute), 34
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_4096 attribute), 34
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_5 attribute), 34
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_512 attribute), 34
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_6 attribute), 34
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_64 attribute), 34
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_7 attribute), 34
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_8 attribute), 34
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_8192 attribute), 34
max_length (nanostream.utils.data_structures.MYSQL_INTEGER_9 attribute), 34
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR_1 attribute), 35
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR1 attribute), 35
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR10 attribute), 35
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR1024 attribute), 35
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR11 attribute), 35
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR12 attribute), 35
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR128 attribute), 35
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR13 attribute), 35
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR14 attribute), 36
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR15 attribute), 36
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR16 attribute), 36
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR16384 attribute), 36
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR17 attribute), 36
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR18 attribute), 36
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR19 attribute), 36
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR2 attribute), 36
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR20 attribute), 36
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR2048 attribute), 36
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR21 attribute), 36
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR22 attribute), 37
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR23 attribute), 37
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR24 attribute), 37
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR25 attribute), 37
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR256 attribute), 37
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR26 attribute), 37
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR27 attribute), 37
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR28 attribute), 37
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR29 attribute), 37

max_length (nanostream.utils.data_structures.MYSQL_VARCHAR attribute), 37
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR30 attribute), 37
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR31 attribute), 38
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR32 attribute), 38
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR32768 attribute), 38
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR4 attribute), 38
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR4096 attribute), 38
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR5 attribute), 38
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR512 attribute), 38
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR6 attribute), 38
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR64 attribute), 38
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR7 attribute), 38
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR8 attribute), 38
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR8192 attribute), 39
max_length (nanostream.utils.data_structures.MYSQL_VARCHAR9 attribute), 39
monitor_futures() (nanostream.civis_nodes.SendToCivis method), 29
MYSQL_BOOL (class in nanostream.utils.data_structures), 30
MYSQL_DATE (class in nanostream.utils.data_structures), 30
MYSQL_ENUM (class in nanostream.utils.data_structures), 30
MYSQL_INTEGER (class in nanostream.utils.data_structures), 30
MYSQL_INTEGER0 (class in nanostream.utils.data_structures), 31
MYSQL_INTEGER1 (class in nanostream.utils.data_structures), 31
MYSQL_INTEGER10 (class in nanostream.utils.data_structures), 31
MYSQL_INTEGER1024 (class in nanostream.utils.data_structures), 31
MYSQL_INTEGER11 (class in nanostream.utils.data_structures), 31
MYSQL_INTEGER12 (class in nanostream.utils.data_structures), 31
MYSQL_INTEGER128 (class in nanostream.utils.data_structures), 31
MYSQL_INTEGER13 (class in nanostream.utils.data_structures), 31
MYSQL_INTEGER14 (class in nanostream.utils.data_structures), 31
MYSQL_INTEGER15 (class in nanostream.utils.data_structures), 31
MYSQL_INTEGER16 (class in nanostream.utils.data_structures), 31
MYSQL_INTEGER16384 (class in nanostream.utils.data_structures), 32
MYSQL_INTEGER17 (class in nanostream.utils.data_structures), 32
MYSQL_INTEGER18 (class in nanostream.utils.data_structures), 32
MYSQL_INTEGER19 (class in nanostream.utils.data_structures), 32
MYSQL_INTEGER2 (class in nanostream.utils.data_structures), 32
MYSQL_INTEGER20 (class in nanostream.utils.data_structures), 32
MYSQL_INTEGER2048 (class in nanostream.utils.data_structures), 32
MYSQL_INTEGER21 (class in nanostream.utils.data_structures), 32
MYSQL_INTEGER22 (class in nanostream.utils.data_structures), 32
MYSQL_INTEGER23 (class in nanostream.utils.data_structures), 32
MYSQL_INTEGER24 (class in nanostream.utils.data_structures), 32
MYSQL_INTEGER25 (class in nanostream.utils.data_structures), 33
MYSQL_INTEGER256 (class in nanostream.utils.data_structures), 33
MYSQL_INTEGER26 (class in nanostream.utils.data_structures), 33
MYSQL_INTEGER27 (class in nanostream.utils.data_structures), 33
MYSQL_INTEGER28 (class in nanostream.utils.data_structures), 33
MYSQL_INTEGER29 (class in nanostream.utils.data_structures), 33
MYSQL_INTEGER3 (class in nanostream.utils.data_structures), 33
MYSQL_INTEGER30 (class in nanostream.utils.data_structures), 33
MYSQL_INTEGER31 (class in nanostream.utils.data_structures), 33
MYSQL_INTEGER32 (class in nanostream.utils.data_structures), 33
MYSQL_INTEGER32768 (class in nanostream.utils.data_structures), 33
MYSQL_INTEGER4 (class in nanostream.utils.data_structures), 34

MYSQL_INTEGER4096 (class in nanostream.utils.data_structures), 34
MYSQL_INTEGER5 (class in nanostream.utils.data_structures), 34
MYSQL_INTEGER512 (class in nanostream.utils.data_structures), 34
MYSQL_INTEGER6 (class in nanostream.utils.data_structures), 34
MYSQL_INTEGER64 (class in nanostream.utils.data_structures), 34
MYSQL_INTEGER7 (class in nanostream.utils.data_structures), 34
MYSQL_INTEGER8 (class in nanostream.utils.data_structures), 34
MYSQL_INTEGER8192 (class in nanostream.utils.data_structures), 34
MYSQL_INTEGER9 (class in nanostream.utils.data_structures), 34
MYSQL_INTEGER_BASE (class in nanostream.utils.data_structures), 35
mysql_type() (in module nanostream.utils.data_structures), 40
MYSQL_VARCHAR (class in nanostream.utils.data_structures), 35
MYSQL_VARCHAR0 (class in nanostream.utils.data_structures), 35
MYSQL_VARCHAR1 (class in nanostream.utils.data_structures), 35
MYSQL_VARCHAR10 (class in nanostream.utils.data_structures), 35
MYSQL_VARCHAR1024 (class in nanostream.utils.data_structures), 35
MYSQL_VARCHAR11 (class in nanostream.utils.data_structures), 35
MYSQL_VARCHAR12 (class in nanostream.utils.data_structures), 35
MYSQL_VARCHAR128 (class in nanostream.utils.data_structures), 35
MYSQL_VARCHAR13 (class in nanostream.utils.data_structures), 35
MYSQL_VARCHAR14 (class in nanostream.utils.data_structures), 35
MYSQL_VARCHAR15 (class in nanostream.utils.data_structures), 36
MYSQL_VARCHAR16 (class in nanostream.utils.data_structures), 36
MYSQL_VARCHAR16384 (class in nanostream.utils.data_structures), 36
MYSQL_VARCHAR17 (class in nanostream.utils.data_structures), 36
MYSQL_VARCHAR18 (class in nanostream.utils.data_structures), 36
MYSQL_VARCHAR19 (class in nanostream.utils.data_structures), 36
MYSQL_VARCHAR2 (class in nanostream.utils.data_structures), 36
MYSQL_VARCHAR20 (class in nanostream.utils.data_structures), 36
MYSQL_VARCHAR2048 (class in nanostream.utils.data_structures), 36
MYSQL_VARCHAR21 (class in nanostream.utils.data_structures), 36
MYSQL_VARCHAR22 (class in nanostream.utils.data_structures), 36
MYSQL_VARCHAR23 (class in nanostream.utils.data_structures), 37
MYSQL_VARCHAR24 (class in nanostream.utils.data_structures), 37
MYSQL_VARCHAR25 (class in nanostream.utils.data_structures), 37
MYSQL_VARCHAR256 (class in nanostream.utils.data_structures), 37
MYSQL_VARCHAR26 (class in nanostream.utils.data_structures), 37
MYSQL_VARCHAR27 (class in nanostream.utils.data_structures), 37
MYSQL_VARCHAR28 (class in nanostream.utils.data_structures), 37
MYSQL_VARCHAR29 (class in nanostream.utils.data_structures), 37
MYSQL_VARCHAR3 (class in nanostream.utils.data_structures), 37
MYSQL_VARCHAR30 (class in nanostream.utils.data_structures), 37
MYSQL_VARCHAR31 (class in nanostream.utils.data_structures), 37
MYSQL_VARCHAR32 (class in nanostream.utils.data_structures), 38
MYSQL_VARCHAR32768 (class in nanostream.utils.data_structures), 38
MYSQL_VARCHAR4 (class in nanostream.utils.data_structures), 38
MYSQL_VARCHAR4096 (class in nanostream.utils.data_structures), 38
MYSQL_VARCHAR5 (class in nanostream.utils.data_structures), 38
MYSQL_VARCHAR512 (class in nanostream.utils.data_structures), 38
MYSQL_VARCHAR6 (class in nanostream.utils.data_structures), 38
MYSQL_VARCHAR64 (class in nanostream.utils.data_structures), 38
MYSQL_VARCHAR7 (class in nanostream.utils.data_structures), 38
MYSQL_VARCHAR8 (class in nanostream.utils.data_structures), 38
MYSQL_VARCHAR8192 (class in nanostream.utils.data_structures), 39

MYSQL_VARCHAR9 (class in nanostream.utils.data_structures), 39
MYSQL_VARCHAR_BASE (class in nanostream.utils.data_structures), 39
MySQLTypeSystem (class in nanostream.utils.data_structures), 39

N

NanoNode (class in nanostream.node), 23
nanostream.civis_nodes (module), 28
nanostream.message.batch (module), 41
nanostream.message.canary (module), 42
nanostream.message.message (module), 41
nanostream.message.poison_pill (module), 41
nanostream.message.trigger (module), 41
nanostream.node (module), 21
nanostream.node_classes.network_nodes (module), 40
nanostream.node_queue.queue (module), 42
nanostream.utils.data_structures (module), 29
NanoStreamMessage (class in nanostream.message.message), 41
NanoStreamQueue (class in nanostream.node_queue.queue), 42
no_op() (in module nanostream.node), 28
NothingToSeeHere (class in nanostream.node), 26

O

OKBLUE (nanostream.node.bcolors attribute), 27
OKGREEN (nanostream.node.bcolors attribute), 27

P

PaginatedHttpGetRequest (class in nanostream.node_classes.network_nodes), 40
PoisonPill (class in nanostream.message.poison_pill), 41
primitive_to_intermediate_type() (in module nanostream.utils.data_structures), 40
PrimitiveTypeSystem (class in nanostream.utils.data_structures), 39
PrinterOfThings (class in nanostream.node), 26
process_item() (nanostream.civis_nodes.CivisSQLExecute method), 28
process_item() (nanostream.civis_nodes.CivisToCSV method), 28
process_item() (nanostream.civis_nodes.EnsureCivisRedshiftTableExists method), 28
process_item() (nanostream.civis_nodes.FindValueInRedshiftColumn method), 29
process_item() (nanostream.civis_nodes.SendToCivis method), 29
process_item() (nanostream.node.AggregateValues method), 21
process_item() (nanostream.node.BatchMessages method), 21
process_item() (nanostream.node.CSVReader method), 21
process_item() (nanostream.node.CSVToDictionaryList method), 21
process_item() (nanostream.node.Filter method), 22
process_item() (nanostream.node.GetEnvironmentVariables method), 22
process_item() (nanostream.node.InsertData method), 22
process_item() (nanostream.node.LocalFileReader method), 23
process_item() (nanostream.node.NanoNode method), 25
process_item() (nanostream.node.PrinterOfThings method), 26
process_item() (nanostream.node.RandomSample method), 26
process_item() (nanostream.node.Remapper method), 26
process_item() (nanostream.node.SequenceEmitter method), 26
process_item() (nanostream.node.Serializer method), 26
process_item() (nanostream.node.SimpleTransforms method), 26
process_item() (nanostream.node.StreamingJoin method), 27
process_item() (nanostream.node.SubstituteRegex method), 27
process_item() (nanostream.node_classes.network_nodes.HttpGetRequest method), 40
process_item() (nanostream.node_classes.network_nodes.HttpGetRequestP method), 40
processor_bak() (nanostream.node.NanoNode method), 25
put() (nanostream.node_queue.queue.NanoStreamQueue method), 42
python_cast_function (nanostream.utils.data_structures.BOOL attribute), 29
python_cast_function (nanostream.utils.data_structures.DataType attribute), 29
python_cast_function (nanostream.utils.data_structures.FLOAT attribute), 30
python_cast_function (nanostream.utils.data_structures.INTEGER attribute), 30
python_cast_function (nanostream.utils.data_structures.MYSQL_BOOL attribute), 30
python_cast_function (nanostream.utils.data_structures.MYSQL_ENUM attribute), 30
python_cast_function (nanostream.utils.data_structures.MYSQL_INTEGER_BASE attribute), 35

python_cast_function (nanostream.utils.data_structures.MYSQL_VARCHAR_BASE attribute), 39
python_cast_function (nanostream.utils.data_structures.STRING attribute), 39
python_cast_function() (nanostream.utils.data_structures.DATETIME method), 29
python_cast_function() (nanostream.utils.data_structures.MYSQL_DATE method), 30
PythonTypeSystem (class in nanostream.utils.data_structures), 39

R

RandomSample (class in nanostream.node), 26
Remapper (class in nanostream.node), 26
responses() (nanostream.node_classes.network_nodes.PaginatedHttpGetRequest method), 41
Row (class in nanostream.utils.data_structures), 39

S

SendToCivis (class in nanostream.civis_nodes), 29
SequenceEmitter (class in nanostream.node), 26
Serializer (class in nanostream.node), 26
setup() (nanostream.civis_nodes.SendToCivis method), 29
setup() (nanostream.node.NanoNode method), 25
setup() (nanostream.node.StreamMySQLTable method), 27
SimpleTransforms (class in nanostream.node), 26
sink_list() (nanostream.node.DynamicClassMediator method), 22
size() (nanostream.node_queue.queue.NanoStreamQueue method), 42
source_list() (nanostream.node.DynamicClassMediator method), 22
start() (nanostream.node.NanoNode method), 25
stream() (nanostream.node.NanoNode method), 25
StreamingJoin (class in nanostream.node), 27
StreamMySQLTable (class in nanostream.node), 27
STRING (class in nanostream.utils.data_structures), 39
SubstituteRegex (class in nanostream.node), 27

T

template_class() (in module nanostream.node), 28
terminate_pipeline() (nanostream.node.NanoNode method), 25
thread_monitor() (nanostream.node.NanoNode method), 26
time_running (nanostream.node.NanoNode attribute), 26
TimeWindowAccumulator (class in nanostream.node), 27
to_intermediate_type() (nanostream.utils.data_structures.DataType method), 29
to_python() (nanostream.utils.data_structures.DataType method), 30
Trigger (class in nanostream.message.trigger), 41
type_mapping() (nanostream.utils.data_structures.DataSourceTypeSystem static method), 29
type_mapping() (nanostream.utils.data_structures.MySQLTypeSystem static method), 39
type_system (nanostream.utils.data_structures.DataType attribute), 30

U

UNDERLINE (nanostream.node.bcolors attribute), 27

W

WARNING (nanostream.node.bcolors attribute), 27