# Python StatsD Documentation

*Release 3.2.1*

**James Socol**

**Apr 11, 2017**

# Contents

statsd is a friendly front-end to Graphite. This is a Python client for the statsd daemon.

**Code** https://github.com/jsocol/pystatsd

**License** MIT; see LICENSE file

**Issues** https://github.com/jsocol/pystatsd/issues

**Documentation** https://statsd.readthedocs.io/

Quickly, to use:

```
>>> import statsd
>>> c = statsd.StatsClient('localhost', 8125)
>>> c.incr('foo')  # Increment the 'foo' counter.
>>> c.timing('stats.timed', 320)  # Record a 320ms 'stats.timed'.
```

You can also add a prefix to all your stats:

```
>>> import statsd
>>> c = statsd.StatsClient('localhost', 8125, prefix='foo')
>>> c.incr('bar')  # Will be 'foo.bar' in statsd/graphite.
```

CHAPTER 1

# Installing

The easiest way to install statsd is with pip!

You can install from PyPI:

```
$ pip install statsd
```

Or GitHub:

```
$ pip install -e git+https://github.com/jsocol/pystatsd#egg=statsd
```

Or from source:

```
$ git clone https://github.com/jsocol/pystatsd
$ cd statsd
$ python setup.py install
```

CHAPTER 2

Contents

## Configuring Statsd

It's easy to configure and use Statsd at runtime, but there are also two shortcuts available.

### Runtime

If you are running the statsd server locally and on the default port, it's extremely easy:

```
from statsd import StatsClient

statsd = StatsClient()
statsd.incr('foo')
```

There are three arguments to configure your `StatsClient` instance. They, and their defaults, are:

```
from statsd import StatsClient

statsd = StatsClient(host='localhost',
                     port=8125,
                     prefix=None,
                     maxudpsize=512,
                     ipv6=False)
```

`host` is the host running the statsd server. It will support any kind of name or IP address you might use.

`port` is the statsd server port. The default for both server and client is `8125`.

`prefix` helps distinguish multiple applications or environments using the same statsd server. It will be prepended to all stats, automatically. For example:

```
from statsd import StatsClient

foo_stats = StatsClient(prefix='foo')
```

```
bar_stats = StatsClient(prefix='bar')

foo_stats.incr('baz')
bar_stats.incr('baz')
```

will produce two different stats, `foo.baz` and `bar.baz`. Without the `prefix` argument, or with the same `prefix`, two `StatsClient` instances will update the same stats.

New in version 2.0.3.

`maxudpsize` specifies the maximum packet size statsd will use. This is an advanced options and should not be changed unless you know what you are doing. Larger values then the default of 512 are generally deemed unsafe for use on the internet. On a controlled local network or when the statsd server is running on 127.0.0.1 larger values can decrease the number of UDP packets when pipelining many metrics. Use with care!

New in version 3.2.

`ipv6` tells the client explicitly to look up the host using IPv6 (`True`) or IPv4 (`False`).

---

**Note:** Python will will inherently bind to an ephemeral port on all interfaces (*0.0.0.0*) for each configured client. This is due to the underlying Sockets API in the operating system/kernel. It is safe to block incoming traffic on your firewall if you wish.

---

### TCP Clients

*TCP-based clients* have an additional `timeout` argument, which defaults to `None`, and is passed to *settimeout <https://docs.python.org/2/library/socket.html#socket.socket.settimeout>*.

## In Django

If you are using Statsd in a Django application, you can configure a default `StatsClient` in the Django settings. All of these settings are optional.

Here are the settings and their defaults:

```
STATSD_HOST = 'localhost'
STATSD_PORT = 8125
STATSD_PREFIX = None
STATSD_MAXUDPSIZE = 512
```

You can use the default `StatsClient` simply:

```
from statsd.defaults.django import statsd

statsd.incr('foo')
```

## From the Environment

Statsd isn't only useful in Django or on the web. A default instance can also be configured via environment variables.

Here are the environment variables and their defaults:

```
STATSD_HOST=localhost
STATSD_PORT=8125
STATSD_PREFIX=None
STATSD_MAXUDPSIZE=512
```

and then in your Python application, you can simply do:

```
from statsd.defaults.env import statsd

statsd.incr('foo')
```

**Note:** As of version 3.0, this default instance is always available, configured with the default values, unless overridden by the environment.

# Data Types

The statsd server supports a number of different data types, and performs different aggregation on each of them. The three main types are *counters*, *timers*, and *gauges*.

The statsd server collects and aggregates in 30 second intervals before flushing to Graphite. Graphite usually stores the most recent data in 1-minute averaged buckets, so when you're looking at a graph, for each stat you are typically seeing the average value over that minute.

## Counters

*Counters* are the most basic and default type. They are treated as a count of a type of event per second, and are, in Graphite, typically averaged over one minute. That is, when looking at a graph, you are usually seeing the average number of events per second during a one-minute period.

The statsd server collects counters under the `stats` prefix.

Counters are managed with the *incr* and *decr* methods of `StatsClient`:

```
from statsd import StatsClient

statsd = StatsClient()

statsd.incr('some.event')
```

You can increment a counter by more than one by passing a second parameter:

```
statsd.incr('some.other.event', 10)
```

You can also use the `rate` parameter to produce sampled data. The statsd server will take the sample rate into account, and the `StatsClient` will only send data `rate` percent of the time. This can help the statsd server stay responsive with extremely busy applications.

`rate` is a float between 0 and 1:

```
# Increment this counter 10% of the time.
statsd.incr('some.third.event', rate=0.1)
```

Because the statsd server is aware of the sampling, it will still show you the true average rate per second.

You can also decrement counters. The `decr` method takes the same arguments as `incr`:

```
statsd.decr('some.other.event')
# Decrease the counter by 5, 15% sample.
statsd.decr('some.third.event', 5, rate=0.15)
```

## Timers

*Timers* are meant to track how long something took. They are an invaluable tool for tracking application performance.

The statsd server collects all timers under the `stats.timers` prefix, and will calculate the lower bound, mean, 90th percentile, upper bound, and count of each timer for each period (by the time you see it in Graphite, that's usually per minute).

- The *lower bound* is the lowest value statsd saw for that stat during that time period.
- The *mean* is the average of all values statsd saw for that stat during that time period.
- The *90th percentile* is a value *x* such that 90% of all the values statsd saw for that stat during that time period are below *x*, and 10% are above. This is a great number to try to optimize.
- The *upper bound* is the highest value statsd saw for that stat during that time period.
- The *count* is the number of timings statsd saw for that stat during that time period. It is not averaged.

The statsd server only operates in millisecond timings. Everything should be converted to milliseconds.

The `rate` parameter will sample the data being sent to the statsd server, but in this case it doesn't make sense for the statsd server to take it into account (except possibly for the *count* value, but then it would be lying about how much data it averaged).

See the *timing documentation* for more detail on using timers with Statsd.

## Gauges

*Gauges* are a constant data type. They are not subject to averaging, and they don't change unless you change them. That is, once you set a gauge value, it will be a flat line on the graph until you change it again.

Gauges are useful for things that are already averaged, or don't need to reset periodically. System load, for example, could be graphed with a gauge. You might use `incr` to count the number of logins to a system, but a gauge to track how many active WebSocket connections you have.

The statsd server collects gauges under the `stats.gauges` prefix.

The *gauge* method also support the `rate` parameter to sample data back to the statsd server, but use it with care, especially with gauges that may not be updated very often.

### Gauge Deltas

Gauges may be *updated* (as opposed to *set*) by setting the `delta` keyword argument to `True`. For example:

```
statsd.gauge('foo', 70)  # Set the 'foo' gauge to 70.
statsd.gauge('foo', 1, delta=True)  # Set 'foo' to 71.
statsd.gauge('foo', -3, delta=True)  # Set 'foo' to 68.
```

**Note:** Support for gauge deltas was added to the server in 3eecd18. You will need to be running at least that version for the `delta` kwarg to have any effect.

### Sets

*Sets* count the number of unique values passed to a key.

For example, you could count the number of users accessing your system using:

statsd.set('users', userid)

If that method is called multiple times with the same userid in the same sample period, that userid will only be counted once.

## Using Timers

*Timers* are an incredibly powerful tool for tracking application performance. Statsd provides a number of ways to use them to instrument your code.

There are four ways to use timers.

### Calling `timing` manually

The simplest way to use a timer is to record the time yourself and send it manually, using the *timing* method:

```
import time
from statsd import StatsClient

statsd = StatsClient()

start = time.time()
time.sleep(3)

# You must convert to milliseconds:
dt = int((time.time() - start) * 1000)
statsd.timing('slept', dt)
```

### Using a context manager

Each `StatsClient` instance contains a *timer* attribute that can be used as a context manager or a decorator. When used as a context manager, it will automatically report the time taken for the inner block:

```
from statsd import StatsClient

statsd = StatsClient()

with statsd.timer('foo'):
    # This block will be timed.
    for i in xrange(0, 100000):
        i ** 2
# The timing is sent immediately when the managed block exits.
```

## Using a decorator

The `timer` attribute decorates your methods in a thread-safe manner. Every time the decorated function is called, the time it took to execute will be sent to the statsd server.

```
from statsd import StatsClient

statsd = StatsClient()

@statsd.timer('myfunc')
def myfunc(a, b):
    """Calculate the most complicated thing a and b can do."""

# Timing information will be sent every time the function is called.
myfunc(1, 2)
myfunc(3, 7)
```

## Using a Timer object directly

New in version 2.1.

`statsd.client.Timer` objects function as context managers and as decorators, but they can also be used directly. (Flat is, after all, better than nested.)

```
from statsd import StatsClient

statsd = StatsClient()

foo_timer = statsd.timer('foo')
foo_timer.start()
# Do something fun.
foo_timer.stop()
```

When `statsd.client.Timer.stop()` is called, a **:ref:'timing stat <timer-type>'_** will automatically be sent to StatsD. You can over ride this behavior with the `send=False` keyword argument to `stop()`:

```
foo_timer.stop(send=False)
foo_timer.send()
```

Use `statsd.client.Timer.send()` to send the stat when you're ready.

---

**Note:** This use of timers is compatible with **:ref:'Pipelines <pipeline-chapter>'_** but be careful with the `send()` method. It *must* be called for the stat to be included when the Pipeline finally sends data, but `send()` will *not* immediately cause data to be sent in the context of a Pipeline. For example:

```
with statsd.pipeline() as pipe:
    foo_timer = pipe.timer('foo').start()
    # Do something...
    pipe.incr('bar')
    foo_timer.stop()  # Will be sent when the managed block exits.

with statsd.pipeline() as pipe:
    foo_timer = pipe.timer('foo').start()
    # Do something...
    pipe.incr('bar')
    foo_timer.stop(send=False)  # Will not be sent.
```

```
    foo_timer.send()  # Will be sent when the managed block exits.
    # Do something else...
```

# Pipelines

The `Pipeline` class is a subclass of `StatsClient` that batches together several stats before sending. It implements the entire client interface, plus a `send()` method.

`Pipeline` objects should be created with `StatsClient().pipeline()`:

```
client = StatsClient()

pipe = client.pipeline()
pipe.incr('foo')
pipe.decr('bar')
pipe.timing('baz', 520)
pipe.send()
```

No stats will be sent until `send()` is called, at which point they will be packed into as few UDP packets as possible.

## As a Context Manager

`Pipeline` objects can also be used as context managers:

```
with StatsClient().pipeline() as pipe:
    pipe.incr('foo')
    pipe.decr('bar')
```

`pipe.send()` will be called automatically when the managed block exits.

## Thread Safety

While `StatsClient` instances are considered thread-safe (or at least as thread-safe as the standard library's `socket.send` is), `Pipeline` instances **are not thread-safe**. Storing stats for later creates at least two important race conditions in a multi-threaded environment. You should create one `Pipeline` per-thread, if necessary.

# TCPStatsClient

The `TCPStatsClient` class has a very similar interface to `StatsClient`, but internally it uses TCP connections instead of UDP. These are the main differencies when using `TCPStatsClient` compared to `StatsClient`:

- The constructor supports a `timeout` parameter to set a timeout on all socket actions.
- `connect()` and all methods that send data can potentially raise socket exceptions.
- **It is not thread-safe**, so it is recommended to not share it across threads unless a lot of attention is paid to make sure that no two threads ever use it at once.

# API Reference

The `StatsClient` provides accessors for all the types of data the statsd server supports.

**Note:** Each public stats API method supports a `rate` parameter, but statsd doesn't always use it the same way. See the *Data Types* for more information.

## StatsClient

```
StatsClient(host='localhost', port=8125, prefix=None, maxudpsize=512)
```

Create a new `StatsClient` instance with the appropriate connection and prefix information.

- `host`: the hostname or IPv4 address of the statsd server.
- `port`: the port of the statsd server.
- `prefix`: a prefix to distinguish and group stats from an application or environment.
- `maxudpsize`: the largest safe UDP packet to save. 512 is generally considered safe for the public internet, but private networks may support larger packet sizes.

### incr

```
StatsClient().incr(stat, count=1, rate=1)
```

Increment a *counter*.

- `stat`: the name of the counter to increment.
- `count`: the amount to increment by. Typically an integer. May be negative, but see also *decr*.
- `rate`: a sample rate, a float between 0 and 1. Will only send data this percentage of the time. The statsd server will take the sample rate into account for counters.

### decr

```
StatsClient().decr(stat, count=1, rate=1)
```

Decrement a *counter*.

- `stat`: the name of the counter to decrement.
- `count`: the amount to decrement by. Typically an integer. May be negative but that will have the impact of incrementing the counter. See also *incr*.
- `rate`: a sample rate, a float between 0 and 1. Will only send data this percentage of the time. The statsd server will take the sample rate into account for counters.

### gauge

```
StatsClient().gauge(stat, value, rate=1, delta=False)
```

Set a *gauge* value.

- `stat`: the name of the gauge to set.
- `value`: the current value of the gauge.
- `rate`: a sample rate, a float between 0 and 1. Will only send data this percentage of the time. The statsd server does *not* take the sample rate into account for gauges. Use with care.
- `delta`: whether or not to consider this a delta value or an absolute value. See the *gauge* type for more detail.

---

**Note:** Gauges were added to the statsd server in commit 0ed78be. If you try to use this method with an older version of the server, the data will not be recorded.

---

### set

```
StatsClient().set(stat, value, rate=1)
```

Increment a *set* value.

- `stat`: the name of the set to update.
- `value`: the unique value to count.
- `rate`: a sample rate, a float between 0 and 1. Will only send data this percentage of the time. The statsd server does *not* take the sample rate into account for sets. Use with care.

---

**Note:** Sets were added to the statsd server in commit 1c10cfc0ac. If you try to use this method with an older version of the server, the data will not be recorded.

---

### timing

```
StatsClient().timing(stat, delta, rate=1)
```

Record *timer* information.

- `stat`: the name of the timer to use.
- `delta`: the number of milliseconds whatever action took. Should always be milliseconds.
- `rate`: a sample rate, a float between 0 and 1. Will only send data this percentage of the time. The statsd server does *not* take the sample rate into account for timers.

## timer

```
with StatsClient().timer(stat, rate=1):
    pass
```

```
@StatsClient().timer(stat, rate=1)
def foo():
    pass
```

```
timer = StatsClient().timer('foo', rate=1)
```

Automatically record timing information for a managed block or function call. See also the *chapter on timing*.

- `stat`: the name of the timer to use.
- `rate`: a sample rate, a float between 0 and 1. Will only send data this percentage of the time. The statsd server does *not* take the sample rate into account for timers.

#### start

```
StatsClient().timer('foo').start()
```

Causes a timer object to start counting. Called automatically when the object is used as a decorator or context manager. Returns the timer object for simplicity.

#### stop

```
timer = StatsClient().timer('foo').start()
timer.stop()
```

Causes the timer object to stop timing and send the results to statsd. Can be called with `send=False` to prevent immediate sending immediately, and use `send()`. Called automatically when the object is used as a decorator or context manager. Returns the timer object.

If `stop()` is called before `start()`, a `RuntimeError` is raised.

#### send

```
timer = StatsClient().timer('foo').start()
timer.stop(send=False)
timer.send()
```

Causes the timer to send any unsent data. If the data has already been sent, or has not yet been recorded, a `RuntimeError` is raised.

---

**Note:** See the note abbout *timer objects and pipelines*.

---

### pipeline

```
StatsClient().pipeline()
```

Returns a *Pipeline* object for collecting several stats. Can also be used as a context manager:

```
with StatsClient().pipeline() as pipe:
    pipe.incr('foo')
```

#### send

```
pipe = StatsClient().pipeline()
pipe.incr('foo')
pipe.send()
```

Causes a *Pipeline* object to send all batched stats.

**Note:** This method is not implemented on the base StatsClient class.

### TCPStatsClient

```
TCPStatsClient(host='localhost', port=8125, prefix=None, timeout=None)
```

Create a new `TCPStatsClient` instance with the appropriate connection and prefix information.

- `host`: the hostname or IPv4 address of the statsd server.
- `port`: the port of the statsd server.
- `prefix`: a prefix to distinguish and group stats from an application or environment.
- `timeout`: socket timeout for any actions on the connection socket.

`TCPStatsClient` implements all methods of `StatsClient`, including `pipeline()`, with the difference that it is not thread safe and it can raise exceptions on connection errors. Unlike `StatsClient` it uses a TCP connection to communicate with StatsD.

In addition to the stats methods, `TCPStatsClient` supports the following TCP-specific methods.

#### close

```
from statsd import TCPStatsClient

statsd = TCPStatsClient()
statsd.incr('some.event')
statsd.close()
```

Closes a connection that's currently open and deletes it's socket. If this is called on a `TCPStatsClient` which currently has no open connection it is a non-action.

#### connect

```
from statsd import TCPStatsClient

statsd = TCPStatsClient()
statsd.incr('some.event')  # calls connect() internally
statsd.close()
statsd.connect()  # creates new connection
```

Creates a connection to StatsD. If there are errors like connection timed out or connection refused, the according exceptions will be raised. It is usually not necessary to call this method because sending data to StatsD will call `connect` implicitely if the current instance of `TCPStatsClient` does not already hold an open connection.

#### reconnect

```
from statsd import TCPStatsClient

statsd = TCPStatsClient()
statsd.incr('some.event')
statsd.reconnect()  # closes open connection and creates new one
```

Closes a currently existing connection and replaces it with a new one. If no connection exists already it will simply create a new one. Internally this does nothing else than calling `close()` and `connect()`.

## Contributing

I happily accept patches if they make sense for the project and work well. If you aren't sure if I'll merge a patch upstream, please open an issue and describe it.

Patches should meet the following criteria before I'll merge them:

- All existing tests must pass.
- Bugfixes and new features must include new tests or asserts.
- Must not introduce any PEP8 or PyFlakes violations.

I recommend doing all development in a virtualenv, though this is really up to you.

It would be great if new or changed features had documentation and included updates to the `CHANGES` file, but it's not totally necessary.

### Running Tests

To run the tests, you just need `nose` and `mock`. These can be installed with `pip`:

```
$ mkvirtualenv statsd
$ pip install -r requirements.txt
$ nosetests
```

You can also run the tests with tox:

```
$ tox
```

Tox will run the tests in Pythons 2.5, 2.6, 2.7, 3.2, 3.3, 3.4, and PyPy, if they're available.

### Writing Tests

New features or bug fixes should include tests that fail without the relevant code changes and pass with them.

For example, if there is a bug in the `StatsClient._send` method, a new test should demonstrate the incorrect behavior by failing, and the associated changes should fix it. The failure can be a FAILURE or an ERROR.

Tests and the code to fix them should be in the same commit. Bisecting should not stumble over any otherwise known failures.

**Note:** Pull requests that only contain tests to demonstrate bugs are welcome, but they will be squashed with code changes to fix them.

## PEP8 and PyFlakes

The development requirements (`requirements.txt`) include the `flake8` tool. It is easy to run:

```
$ flake8 statsd/
```

`flake8` should not raise any issues or warnings.

**Note:** The docs directory includes a Sphinx-generated conf.py that has several violations. That's fine, don't worry about it.

## Documentation

The documentation lives in the `docs/` directory and is automatically built and pushed to ReadTheDocs.

If you change or add a feature, and want to update the docs, that would be great. New features may need a new chapter. You can follow the examples already there, and be sure to add a reference to `docs/index.rst`. Changes or very small additions may just need a new heading in an existing chapter.

CHAPTER 3

# Indices and tables

- search