Dataplicity Lomond Documentation

Release 0.3.2

Dataplicity

Jul 05, 2018

Guide

Lomond is a Pythonic WebSockets client library designed for reliability and ease of use.

CHAPTER 1

Sponsor

Lomond was sponsored by Dataplicity – Control your Raspberry Pi from anywhere!

CHAPTER 2

Support

For questions or comments about this library, either open an issue or contact support@dataplicity.com.

2.1 Introduction

Lomond is a websocket client library designed to make adding websocket support to your application as tranquil as the Scottish Loch it was named after.

2.2 Installing

You can install Lomond with `pip` as follows:

```
pip install lomond
```

Or to upgrade to the most recent version:

```
pip install lomond --upgrade
```

Alternatively, if you would like to install from source, check out the code from Github.

You may wish to install *wsaccel*, which is a C module containing optimizations for some websocket operations. Lomond will use it if available:

```
pip install wsaccel
```

2.3 Example

To whet your appetite, the following is an example of how to connect to a WebSocket endpoint and interact with the server:

```
from lomond import WebSocket
websocket = WebSocket('wss://ws-feed.gdax.com')

for event in websocket:
    if event.name == "ready":
        websocket.send_json(
            type='subscribe',
            product_ids=['BTC-USD'],
            channels=['ticker']
        )
    elif event.name == "text":
        print(event.json)
```

This code connects to a Gdax, a Bitcoin exchange, and subscribes to realtime notifications about the price of Bitcoin–which it writes to the terminal.

This example is in the Lomond library. You can run it with the following:

```
python -m lomond.examples.btcticker
```

2.4 Basic Usage

To connect to a websocket server, first construct a *WebSocket* object, with a `ws://` or `wss://` URL. Here is an example:

```
from lomond.websocket import WebSocket
ws = WebSocket('wss://echo.websocket.org')
```

No socket connection is made by a freshly constructed WebSocket object. To connect and interact with a websocket server, iterate over the WebSocket instance, which will yield a number of *Event* objects. Here's an example:

```
for event in ws:
    print(event)
```

Here is an example of the output you might get from the above code:

```
Connecting(url='wss://echo.websocket.org')
Connected(url='wss://echo.websocket.org')
Ready(<response HTTP/1.1 101 Web Socket Protocol Handshake>, protocol=None,␣
↪extensions=set([]))
```

The *Ready* event indicates a successful connection to a websocket server. You may now use the *send_text()* and *send_binary()* methods to send data to the server.

When you receive data from the server, a *Text* or *Binary* event will be generated.

2.4.1 Connect Method

Iterating over the WebSocket instance calls the *connect()* method with default parameters, i.e. it is equivalent to the following:

```
for event in ws.connect():
    print(event)
```

You may want to call *connect()* explicitly to customize the WebSocket behaviour.

2.5 Events

Events inform your application when data is received from the server or when the websocket state changes.

All events are derived from *Event* and will contain at least 2 attributes; `received_time` is the epoch time the event was received, and `name` is the name of the event. Some events have additional attributes with more information. See the *Events* for details.

When handling events, you can either check the type with *isinstance* or by looking at the *name* attribute.

For example, the following two lines are equivalent:

```
if isinstance(event, events.Ready):
```

or:

```
if event.name == "ready":
```

Note: The `isinstance` method is possibly uglier, but has the advantage that you are less likely to introduce a bug with a typo in the event name.

If an event is generated that you aren't familiar with, then you should simply ignore it. This is important for backwards compatibility; future versions of Lomond may introduce new event types.

Be careful with code that responds to events. Should there be an unhandled exception within the event loop, Lomond will disconnect the socket without sending a close packet. It's up to your application to ensure that programming errors don't prevent the websocket from closing gracefully.

You may wish to adopt an defensive approach to handling WebSocket events, such as the following:

```
for event in websocket:
    try:
        on_event(event)
    except:
        log.exception('error handling %r', event)
        websocket.close()
```

2.6 Compression

Lomond supports the *permessage-deflate* extension to the WebSocket specification to compress WebSocket frames. To enable compression, construct the WebSocket with the `compress` parameter set to `True`:

```
ws = WebSocket('wss://ws.example.org', compress=True)
```

This tells the server in the initial request to enabled compression if it is supported. If the server supports compression then Lomond may receive text or binary messages in compressed form. The decompression happens automatically so you will see the decompressed data in *Text* and *Binary* events.

You can send compressed messages by setting the `compress` parameter on the *send_text()* and *send_binary()* methods. This parameter is `True` by default, but you might want to set it to `False` if you know the data is already compressed.

If the server does not support compression, then setting the `compress` parameter will have no effect. The *supports_compression* property will be set to `True` if compression is enabled or `False` if the server does not support compression.

2.7 Closing the WebSocket

The websocket protocol specifies how to close the websocket cleanly. The procedure for closing depends on whether the close is initiated by the client or the server.

2.7.1 Client

To close a websocket, call the *close()* method to initiate a *websocket close handshake*. You may call this method from within the websocket loop, or from another thread.

When you call *close()*, Lomond sends a close packet to the server. The server will respond by sending a close packet of its own. Only when this echoed close packet is received will the WebSocket close the underlaying tcp/ip socket. This allows both ends of the connection to finish what they are doing without worrying the remote end has stopped responding to messages.

Note: When you call the `close()` method, you will no longer be able to *send* data, but you may still *receive* packets from the server until the close has completed.

When the websocket has been closed, you will receive a *Closed* event, followed by a *Disconnected* event, and the event loop will exit.

It's possible a malfunctioning server may not respond to a close packet, which would leave a WebSocket in a permanent *closing* state. As a precaution, Lomond will force close the socket after 30 seconds, if the server doesn't respond to a close packet. You can change or disable this timeout with the *close_timeout* parameter, on *connect()*.

2.7.2 Server

The websocket server will send a close packet when it wished to close. When Lomond receives that packet, a *Closing* event will be generated. You may send text or binary messages in response to the Closing event, but afterwards Lomond echos the close packet and no further data may be sent. The server will then close the socket, and you will receive a *Disconnected* event, followed by the event loop ending.

2.7.3 Non-graceful Closes

A non-graceful close is when a the tcp/ip connection is closed *without* completing the closing handshake. This can occur if the server is misbehaving or if connectivity has been interrupted.

The *Disconnected* event contains a boolean attribute `graceful`, which will be `False` if the closing handshake was not completed.

2.8 Pings and Pongs

Both the websocket server and client may send 'ping' packets, which should be responded to with a 'pong' packet. This allows both ends of the connection to know if the other end is really listening.

By default, Lomond will send pings packets every 30 seconds. If you wish to change this rate or disable ping packets entirely, you may use the *connect()* method.

Here's how you would disable pings:

```
websocket = Websocket('wss://ws.example.org')
for event in WebSocket.connect(ping_rate=0):
    on_event(event)
```

Lomond will also automatically respond to ping requests. Since this is a requirement of the websocket specification, you probably don't want to change this behaviour. But it may be disabled with the *auto_pong* flag in *connect()*.

When Lomond receives a ping packet from the server, a *Ping* event will be generated. When the server sends you a pong packet, a *Pong* event will be generated.

You can send a ping / pong packet at any time with *send_ping()* and *send_pong()*.

Note: The server may send pong packets *not* in response to a ping packet (see https://tools.ietf.org/html/rfc6455#section-5.5.3 for details).

2.9 Polling

Lomond checks for automatic pings and performs other housekeeping tasks at a regular intervals. This *polling* is exposed as *Poll* events. Your application can use these events to do any processing that needs to be invoked at regular intervals.

The default poll rate of 5 seconds is granular enough for Lomond's polling needs, while having negligible impact on CPU. If your application needs to process at a faster rate, you may set the `poll` parameter of *connect()*.

Note: If your application needs to be more realtime than polling once a second, you should probably use threads in tandem with the event loop.

2.10 Proxies

Lomond can work with WebSockets over HTTP proxy. By default, Lomond will autodetect the proxy from `HTTP_PROXY` and `HTTPS_PROXY` environment variables, used for the `ws` and `wss` protocols respectively.

You may set the proxy manually by supplying a dictionary with the keys `http` and `https` (which may contain the same value). Here's an example:

```
ws = Websocket(
    'wss://echo.example.org',
    proxies = {
        'http': 'http://127.0.0.1:8888',
        'https': 'http://127.0.0.1:8888'
    }
)
```

Note: If you want to disable automatic proxy detection, then set the `proxies` parameter to an empty dictionary.

2.11 WebSockets and Threading

WebSocket objects are *thread safe*, but Lomond does not need to launch any threads to run a websocket. For many applications, responding to data and poll events is all you will need. However, if your application needs to do more than communicate with a websocket server, you may want to run a websocket in a thread of its own.

2.12 Persistent Connections

Lomond supports a simple mechanism for persistent connections – you can tell Lomond to continually retry a websocket connection if it is dropped for any reason. This allows an application to maintain a websocket connection even if there are any outages in connectivity.

To run a persistent connection, wrap a WebSocket instance with `persist()`. Here is an example:

```
from lomond.persist import persist
websocket = WebSocket('wss://ws.example.org')
for event in persist(websocket):
    # handle event
```

You will receive events as normal with the above loop.

If the connection is dropped for any reason, you will receive `Disconnected` as usual, followed by `Connecting` when Lomond retries the connection. Lomond will keep retrying the connection until it is successful, and a `Ready` event is generated.

The `persist()` function implements *exponential backoff*. If the websocket object fails to connect, it will wait for a random period between zero seconds and an upper limit. Every time the connection fails, it will double the upper limit until it connects, or a maximum delay is reached.

The exponential backoff prevents a client from hammering a server that may already be overloaded. It also prevents the client from being stuck in a cpu intensive spin loop.

2.13 Exceptions

Lomond takes great care not to leak any socket or system related exceptions. During normal usage of `WebSocket` objects you can expect only the following exception hierarchy to be thrown.

exception lomond.errors.**CompressionParameterError**(*msg*, **args*, ***kwargs*)
Raised when the headers contain invalid compression parameters.

exception lomond.errors.**ConnectFail**(*msg*, **args*, ***kwargs*)
An error connecting to a socket.

exception lomond.errors.**CriticalProtocolError**(*msg*, **args*, ***kwargs*)
Critical protocol error. An egregious error in the protocol resulting in an immediate disconnect.

exception lomond.errors.**FrameBuildError**(*msg*, **args*, ***kwargs*)
Raised when trying to build an invalid websocket frame.

exception lomond.errors.**HandshakeError**(*msg*, **args*, ***kwargs*)
Raised when the server doesn't respond correctly to the websocket handshake.

exception lomond.errors.**PayloadTooLarge**(*msg*, **args*, ***kwargs*)
The payload length field is too large.

Websocket messages have a maximum payload of 2**63 bytes. In practice it may be impossible to generate such a packet for real, but its feasible a corrupt packet header could make it appear that such a packet was being sent.

exception lomond.errors.**ProtocolError**(*msg*, **args*, ***kwargs*)
Raised in response to a protocol violation.

exception lomond.errors.**TransportFail**(*msg*, **args*, ***kwargs*)
The transport (socket) failed when sending.

Likely indicating connectivity issues.

exception lomond.errors.**WebSocketClosed**(*msg*, **args*, ***kwargs*)
Raised when attempting to send over a closed websocket.

exception lomond.errors.**WebSocketClosing**(*msg*, **args*, ***kwargs*)
Raised when attempting to send over a closing websocket.

exception lomond.errors.**WebSocketError**(*msg*, **args*, ***kwargs*)
Base exception.

exception lomond.errors.**WebSocketUnavailable**(*msg*, **args*, ***kwargs*)
The websocket can not be used.

2.14 Events

class lomond.events.**BackOff**(*delay*)
Generated when a persistent connection has to wait before re- attempting a connection.

Parameters **delay** (*float*) – The delay (in seconds) before Lomond will re- attempt to connect.

class lomond.events.**Binary**(*data*)
Generated when Lomond receives a binary message from the server.

Parameters **data** (*bytes*) – The binary payload.

class lomond.events.**Closed**(*code*, *reason*)
Generated when the websocket was closed. The websocket may no longer send packets after this event has been received. This event will be followed by *Disconnected*.

Parameters

- **code** – The closed code returned from the server.
- **reason** (*str*) – An optional description why the websocket was closed, as returned from the server.

class lomond.events.**Closing**(*code*, *reason*)
Generated when the server is closing the connection.

No more messages will be received from the server, but you may still send messages while handling this event. A *Disconnected* event should be generated shortly after this event.

Parameters

- **code** – The closed code returned from the server.
- **reason** (*str*) – An optional description why the websocket was closed, as returned from the server.

class lomond.events.**ConnectFail**(*reason*)
Generate when Lomond was unable to connect to a Websocket server.

Parameters **reason** (*str*) – A short description of the reason for the failure.

class lomond.events.**Connected**(*url*, *proxy=None*)
Generated when Lomond has connected to a server but not yet negotiated the websocket upgrade.

Parameters

- **url** (*str*) – The websocket URL connected to.
- **proxy** (*str*) – The proxy URL connected to (or None).

class lomond.events.**Connecting**(*url*)
Generated prior to establishing a websocket connection to a server.

Parameters **url** – The websocket URL the websocket is connecting to.

class lomond.events.**Disconnected**(*reason=u'closed'*, *graceful=False*)
Generated when a websocket connection has been dropped.

Parameters

- **reason** (*str*) – A description of why the websocket was closed.
- **graceful** (*bool*) – Flag indicating if the connection was dropped gracefully (*True*), or disconnected due to a socket failure (*False*) or other problem.

class lomond.events.**Event**
Base class for a websocket 'event'.

class lomond.events.**Ping**(*data*)
Generated when Lomond received a ping packet from the server.

Parameters **data** (*bytes*) – Ping payload data.

class lomond.events.**Poll**
A generated poll event.

class lomond.events.**Pong**(*data*)
Generated when Lomond receives a pong packet from the server.

Parameters **data** (*bytes*) – The pong payload data.

class lomond.events.**ProtocolError**(*error*, *critical*)
Generated when the server deviates from the protocol.

Parameters

- **error** (*str*) – A description of the error.
- **critical** (*bool*) – Indicates if the error is considered 'critical'. If `True`, Lomond will disconnect immediately. If `False`, Lomond will send a close message to the server.

class lomond.events.**Ready**(*response*, *protocol*, *extensions*)
Generated when Lomond has connected to the server, and successfully negotiated the websocket upgrade.

Parameters

- **response** – A *Response* object.
- **protocol** (*str*) – A websocket protocol or `None` if no protocol was supplied.
- **extensions** (*set*) – A set of negotiated websocket extensions. Currently only the `'permessage-deflate'` extension is supported.

class lomond.events.**Rejected**(*response*, *reason*)
Server rejected WS connection.

class lomond.events.**Text**(*text*)
Generated when Lomond receives a text message from the server.

Parameters **text** (*str*) – The text payload.

json
Text decoded as JSON.

Calls `json.loads` to decode the `text` attribute, and may throw the same exceptions if the text is not valid json.

class lomond.events.**UnknownMessage**(*message*)
An application message was received, with an unknown opcode.

class lomond.events.**Unresponsive**
The server has not responding to pings within *ping_timeout* seconds.

Will be followed by a *Disconnected* event.

2.15 Persist

Maintains a persistent websocket connection.

lomond.persist.**persist**(*websocket*, *poll=5*, *min_wait=5*, *max_wait=30*, *ping_rate=30*, *ping_timeout=None*, *exit_event=None*)
Run a websocket, with a retry mechanism and exponential back-off.

Parameters

- **websocket** – A `Websocket` instance.
- **poll** (*float*) – The websocket poll rate, in seconds.
- **min_wait** (*float*) – The minimum time to wait between reconnect attempts (seconds).
- **max_wait** (*float*) – The maximum time to wait between reconnect attempts (seconds).
- **ping_rate** (*float*) – Delay between pings (seconds), or *0* for no auto ping.
- **ping_timeout** (*float*) – Maximum time in seconds to wait for a pong response before disconnecting. Set to *None* (default) to disable. If set, double *ping_rate* would be a good starting point.
- **exit_event** – A threading event object, which can be used to exit the persist loop if it is set. Set to *None* to use an internal event object.

lomond.persist.**random**() → x in the interval [0, 1).

2.16 Status

class lomond.status.**Status**

2.17 Response

A simple abstraction for an HTTP response.

A response object is supplied in the *Ready* event.

class lomond.response.**Response**(*header_data*)
A HTTP response.

Parameters **header_data** (*bytes*) – Raw response.

get(*name*, *default=None*)
Get a header.

Parameters

- **name** (*str*) – Name of the header to retrieve.
- **default** – Default value if header is not present.

Return type str

get_list(*name*)
Extract a list from a header.

Parameters **name** (*bytes*) – Name of the header to retrieve.

Return type str

Returns A list of strings in the header.

2.18 WebSocket

Abstract websocket functionality.

class lomond.websocket.**WebSocket**(*url*, *proxies=None*, *protocols=None*, *agent=None*, *compress=False*)
IO independent websocket functionality.

Parameters

- **url** (*str*) – A websocket URL, must have a `ws://` or `wss://` protocol.
- **proxies** (*dict*) – A dict containing `'http'` or `'https'` urls to a proxy server, or `None` to attempt to auto-detect proxies from environment. Pass an empty dict to disable proxy.
- **protocols** (*list*) – A list of supported protocols (defaults to no protocols).
- **agent** (*str*) – A user agent string to be sent in the header. The default uses the value `USER_AGENT` defined in `lomond.constants`.

add_header(*header*, *value*)
Add a custom header to the websocket request.

Parameters

- **header** (*bytes*) – Name of the header.
- **value** (*bytes*) – Value of the header.

build_request()
Get the websocket request (in bytes).

This method is called from the session, and should not be invoked explicitly.

close(*code=1000*, *reason='goodbye'*)
Close the websocket.

Parameters

- **code** (*int*) – A closing code, which should probably be one of the enumerations in *lomond.status.Status* or a valid value as specified in https://tools.ietf.org/html/rfc6455#section-7.4
- **reason** (*str*) – A short descriptive reason why the websocket is closing. This value is intended for the remote end to help in debugging.

Note: Closing the websocket won't exit the main loop immediately; it will put the websocket in to a *closing* state while it waits for the server to echo back a close packet. No data may be sent by the application when the websocket is closing.

---

connect(*session_class=<class 'lomond.session.WebsocketSession'>*, *poll=5.0*, *ping_rate=30.0*, *ping_timeout=None*, *auto_pong=True*, *close_timeout=30.0*)

Connect the websocket to a session.

Parameters

- **session_class** – An object to manage the *session*. This object is an extension mechanism that will allow the WebSocket to be *driven* by different back-ends. For now, treat it as an implementation detail and leave it as the default.
- **poll** (*float*) – Rate (in seconds) that poll events should be generated.
- **ping_rate** (*float*) – Rate that ping packets should be sent. Set to *0* to disable auto pings.
- **ping_timeout** (*float*) – Maximum time in seconds to wait for a pong response before disconnecting. Set to *None* (default) to disable. If set, double *ping_rate* would be a good starting point.
- **auto_pong** (*bool*) – Enable (default) automatic response to ping events.
- **close_timeout** (*float*) – Seconds to wait for server to respond to a close packet, before closing the socket. Set to *None* or *0* to disable the timeout.

Returns An iterable of `Event` instances.

feed(*data*)

Feed with data from the socket, and yield any events.

This method is called by the Session object, and is not needed for normal use.

Parameters **data** (*bytes*) – data received over a socket.

force_disconnect()

Force the socket to disconnect.

is_active

Boolean that indicates the socket is 'active' i.e. not in a closing state.

is_closed

Flag that indicates if the websocket is closed.

is_closing

Boolean that indicates if the websocket is in a closing state. No further messages may be sent when a websocket is closing.

is_secure

Boolean that indicates if the websocket is over ssl (i.e. the *wss* protocol).

on_disconnect()

Called on disconnect.

on_response(*response*)
Called when the HTTP response has been received.

process_extensions(*extensions*)
Process extension headers.

reset()
Reset the state.

send_binary(*data*, *compress=True*)
Send a binary message.

Parameters

- **data** (*bytes*) – Binary data to send.
- **compress** (*bool*) – Send data in compressed form, if compression is enabled on the server.

Raises **TypeError** – If data is not bytes.

send_json(*_obj=Ellipsis*, ***kwargs*)
Encode an object as JSON and send a text message.

The object to encode may be specified as a single positional argument OR if as keyword arguments which will be encoded as a JSON object. The following two lines will send the same JSON:

```
websocket.send_json({'foo': 'bar'})
websocket.send_json(foo='bar')
```

Parameters **obj** – An object to be encoded as JSON.

Raises **TypeError** – If *obj* could not be encoded as JSON.

send_ping(*data=''*)
Send a ping packet.

Parameters **data** (*bytes*) – Data to send in the ping message (must be <= 125 bytes).

Raises

- **TypeError** – If *data* is not bytes.
- **ValueError** – If *data* is > 125 bytes.

send_pong(*data*)
Send a pong packet.

Parameters **data** (*bytes*) – Data to send in the ping message (must be <= 125 bytes).

A *pong* may be sent in response to a ping, or unsolicited to keep the connection alive.

Raises

- **TypeError** – If *data* is not bytes.
- **ValueError** – If *data* is > 125 bytes.

send_text(*text*, *compress=True*)
Send a text message.

Parameters

- **text** (*str*) – Text to send.

- **compress** (*bool*) – Send the text in compressed form, if compression is enabled on the server.

Raises **TypeError** – If data is not str (or unicode on Py2).

sent_close_time
The time (seconds since session start) when a close packet was sent (or None if no close packet has been sent).

supports_compression
Boolean that indicates if the server has compression enabled.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

l

Index

S

T

U

W