hyperframe

Release 6.0.1

Apr 17, 2021

Contents

hyperframe is a pure-Python tool for working with HTTP/2 frames. This library allows you to create, serialize, and parse HTTP/2 frames.

Working with it is easy:

```
import hyperframe.frame

f = hyperframe.frame.DataFrame(stream_id=5)
f.data = b'some binary data'
f.flags.add('END_STREAM')
f.flags.add('PADDED')
f.padding_length = 30

data = f.serialize()

new_frame, length = hyperframe.frame.Frame.parse_frame_header(data[:9])
new_frame.parse_body(memoryview(data[9:9 + length]))
```

hyperframe is pure-Python, contains no external dependencies, and runs on a wide variety of Python interpreters and platforms. Made available under the MIT license, why write your own frame parser?

Contents:

CHAPTER 1

Installing hyperframe

hyperframe is trivial to install from the Python Package Index. Simply run:

```
$ pip install hyperframe
```

Alternatively, feel free to download one of the release tarballs from our GitHub page, extract it to your favourite directory, and then run

```
$ python setup.py install
```

hyperframe has no external dependencies.

CHAPTER 2

hyperframe API

This document provides the hyperframe API.

All frame classes are subclasses of *Frame*, and provide the methods and attributes defined there. Additionally, some frames use the *Priority* and *Padding* mixins, and make the methods and attributes defined on *those* mixins available as well.

Rather than clutter up the documentation repeatedly documenting those methods and objects, we explicitly show the inheritance hierarchy of the frames: don't forget to consult the parent classes before deciding if a method or attribute you need is not present!

class hyperframe.frame.**Frame**(*stream_id: int*, *flags: Iterable[str] = ()*)
The base class for all HTTP/2 frames.

body_len = None
The frame length, excluding the nine-byte header.

defined_flags = []
The flags defined on this type of frame.

static explain(*data: memoryview*) → Tuple[hyperframe.frame.Frame, int]
Takes a bytestring and tries to parse a single frame and print it.

This function is only provided for debugging purposes.

Parameters **data** – A memoryview object containing the raw data of at least one complete frame (header and body).

New in version 6.0.0.

flags = None
The flags set for this frame.

parse_body(*data: memoryview*) → None
Given the body of a frame, parses it into frame data. This populates the non-header parts of the frame: that is, it does not populate the stream ID or flags.

Parameters **data** – A memoryview object containing the body data of the frame. Must not contain *more* data than the length returned by *parse_frame_header*.

static parse_frame_header(*header: memoryview*, *strict: bool = False*) → Tuple[hyperframe.frame.Frame, int]

Takes a 9-byte frame header and returns a tuple of the appropriate Frame object and the length that needs to be read from the socket.

This populates the flags field, and determines how long the body is.

Parameters

- **header** – A memoryview object containing the 9-byte frame header data of a frame. Must not contain more or less.
- **strict** – Whether to raise an exception when encountering a frame not defined by spec and implemented by hyperframe.

Raises ***hyperframe.exceptions.UnknownFrameError*** – If a frame of unknown type is received.

Changed in version 5.0.0: Added **:param:'strict'** to accommodate *ExtensionFrame*

serialize() → bytes

Convert a frame into a bytestring, representing the serialized form of the frame.

stream_id = None

The stream identifier for the stream this frame was received on. Set to 0 for frames sent on the connection (stream-id 0).

type = None

The byte used to define the type of the frame.

class hyperframe.frame.**Padding**(*stream_id: int*, *pad_length: int = 0*, ***kwargs*)

Mixin for frames that contain padding. Defines extra fields that can be used and set by frames that can be padded.

pad_length = None

The length of the padding to use.

class hyperframe.frame.**Priority**(*stream_id: int*, *depends_on: int = 0*, *stream_weight: int = 0*, *exclusive: bool = False*, ***kwargs*)

Mixin for frames that contain priority data. Defines extra fields that can be used and set by frames that contain priority data.

depends_on = None

The stream ID of the stream on which this stream depends.

exclusive = None

Whether the exclusive bit was set.

stream_weight = None

The weight of the stream. This is an integer between 0 and 256.

class hyperframe.frame.**DataFrame**(*stream_id: int*, *data: bytes = b''*, ***kwargs*)

Bases: *hyperframe.frame.Padding*, *hyperframe.frame.Frame*

DATA frames convey arbitrary, variable-length sequences of octets associated with a stream. One or more DATA frames are used, for instance, to carry HTTP request or response payloads.

data = None

The data contained on this frame.

defined_flags = [Flag(name='END_STREAM', bit=1), Flag(name='PADDED', bit=8)]

The flags defined for DATA frames.

flow_controlled_length
The length of the frame that needs to be accounted for when considering flow control.

parse_body(*data: memoryview*) → None
Given the body of a frame, parses it into frame data. This populates the non-header parts of the frame: that is, it does not populate the stream ID or flags.

Parameters **data** – A memoryview object containing the body data of the frame. Must not contain *more* data than the length returned by *parse_frame_header*.

type = 0
The type byte for data frames.

class hyperframe.frame.**PriorityFrame**(*stream_id: int*, *depends_on: int = 0*, *stream_weight: int = 0*, *exclusive: bool = False*, ***kwargs*)
Bases: *hyperframe.frame.Priority*, *hyperframe.frame.Frame*

The PRIORITY frame specifies the sender-advised priority of a stream. It can be sent at any time for an existing stream. This enables reprioritisation of existing streams.

defined_flags = []
The flags defined for PRIORITY frames.

parse_body(*data: memoryview*) → None
Given the body of a frame, parses it into frame data. This populates the non-header parts of the frame: that is, it does not populate the stream ID or flags.

Parameters **data** – A memoryview object containing the body data of the frame. Must not contain *more* data than the length returned by *parse_frame_header*.

type = 2
The type byte defined for PRIORITY frames.

class hyperframe.frame.**RstStreamFrame**(*stream_id: int*, *error_code: int = 0*, ***kwargs*)
Bases: *hyperframe.frame.Frame*

The RST_STREAM frame allows for abnormal termination of a stream. When sent by the initiator of a stream, it indicates that they wish to cancel the stream or that an error condition has occurred. When sent by the receiver of a stream, it indicates that either the receiver is rejecting the stream, requesting that the stream be cancelled or that an error condition has occurred.

defined_flags = []
The flags defined for RST_STREAM frames.

error_code = None
The error code used when resetting the stream.

parse_body(*data: memoryview*) → None
Given the body of a frame, parses it into frame data. This populates the non-header parts of the frame: that is, it does not populate the stream ID or flags.

Parameters **data** – A memoryview object containing the body data of the frame. Must not contain *more* data than the length returned by *parse_frame_header*.

type = 3
The type byte defined for RST_STREAM frames.

class hyperframe.frame.**SettingsFrame**(*stream_id: int = 0*, *settings: Optional[Dict[int, int]] = None*, ***kwargs*)
Bases: *hyperframe.frame.Frame*

The SETTINGS frame conveys configuration parameters that affect how endpoints communicate. The parameters are either constraints on peer behavior or preferences.

Settings are not negotiated. Settings describe characteristics of the sending peer, which are used by the receiving peer. Different values for the same setting can be advertised by each peer. For example, a client might set a high initial flow control window, whereas a server might set a lower value to conserve resources.

ENABLE_CONNECT_PROTOCOL = 8
The byte that signals SETTINGS_ENABLE_CONNECT_PROTOCOL setting.

ENABLE_PUSH = 2
The byte that signals the SETTINGS_ENABLE_PUSH setting.

HEADER_TABLE_SIZE = 1
The byte that signals the SETTINGS_HEADER_TABLE_SIZE setting.

INITIAL_WINDOW_SIZE = 4
The byte that signals the SETTINGS_INITIAL_WINDOW_SIZE setting.

MAX_CONCURRENT_STREAMS = 3
The byte that signals the SETTINGS_MAX_CONCURRENT_STREAMS setting.

MAX_FRAME_SIZE = 5
The byte that signals the SETTINGS_MAX_FRAME_SIZE setting.

MAX_HEADER_LIST_SIZE = 6
The byte that signals the SETTINGS_MAX_HEADER_LIST_SIZE setting.

defined_flags = [Flag(name='ACK', bit=1)]
The flags defined for SETTINGS frames.

parse_body(*data: memoryview*) → None
Given the body of a frame, parses it into frame data. This populates the non-header parts of the frame: that is, it does not populate the stream ID or flags.

Parameters **data** – A memoryview object containing the body data of the frame. Must not contain *more* data than the length returned by *parse_frame_header*.

settings = None
A dictionary of the setting type byte to the value of the setting.

type = 4
The type byte defined for SETTINGS frames.

class hyperframe.frame.**PushPromiseFrame**(*stream_id: int*, *promised_stream_id: int = 0*, *data: bytes = b''*, ***kwargs*)
Bases: *hyperframe.frame.Padding*, *hyperframe.frame.Frame*

The PUSH_PROMISE frame is used to notify the peer endpoint in advance of streams the sender intends to initiate.

data = None
The HPACK-encoded header block for the simulated request on the new stream.

defined_flags = [Flag(name='END_HEADERS', bit=4), Flag(name='PADDED', bit=8)]
The flags defined for PUSH_PROMISE frames.

parse_body(*data: memoryview*) → None
Given the body of a frame, parses it into frame data. This populates the non-header parts of the frame: that is, it does not populate the stream ID or flags.

Parameters **data** – A memoryview object containing the body data of the frame. Must not contain *more* data than the length returned by *parse_frame_header*.

promised_stream_id = None
The stream ID that is promised by this frame.

type = 5
The type byte defined for PUSH_PROMISE frames.

class hyperframe.frame.**PingFrame**(*stream_id: int = 0*, *opaque_data: bytes = b''*, ***kwargs*)
Bases: *hyperframe.frame.Frame*

The PING frame is a mechanism for measuring a minimal round-trip time from the sender, as well as determining whether an idle connection is still functional. PING frames can be sent from any endpoint.

defined_flags = [Flag(name='ACK', bit=1)]
The flags defined for PING frames.

opaque_data = None
The opaque data sent in this PING frame, as a bytestring.

parse_body(*data: memoryview*) → None
Given the body of a frame, parses it into frame data. This populates the non-header parts of the frame: that is, it does not populate the stream ID or flags.

Parameters **data** – A memoryview object containing the body data of the frame. Must not contain *more* data than the length returned by *parse_frame_header*.

type = 6
The type byte defined for PING frames.

class hyperframe.frame.**GoAwayFrame**(*stream_id: int = 0*, *last_stream_id: int = 0*, *error_code: int = 0*, *additional_data: bytes = b''*, ***kwargs*)
Bases: *hyperframe.frame.Frame*

The GOAWAY frame informs the remote peer to stop creating streams on this connection. It can be sent from the client or the server. Once sent, the sender will ignore frames sent on new streams for the remainder of the connection.

additional_data = None
Any additional data sent in the GOAWAY.

defined_flags = []
The flags defined for GOAWAY frames.

error_code = None
The error code for connection teardown.

last_stream_id = None
The last stream ID definitely seen by the remote peer.

parse_body(*data: memoryview*) → None
Given the body of a frame, parses it into frame data. This populates the non-header parts of the frame: that is, it does not populate the stream ID or flags.

Parameters **data** – A memoryview object containing the body data of the frame. Must not contain *more* data than the length returned by *parse_frame_header*.

type = 7
The type byte defined for GOAWAY frames.

class hyperframe.frame.**WindowUpdateFrame**(*stream_id: int*, *window_increment: int = 0*, ***kwargs*)
Bases: *hyperframe.frame.Frame*

The WINDOW_UPDATE frame is used to implement flow control.

Flow control operates at two levels: on each individual stream and on the entire connection.

Both types of flow control are hop by hop; that is, only between the two endpoints. Intermediaries do not forward WINDOW_UPDATE frames between dependent connections. However, throttling of data transfer by any receiver can indirectly cause the propagation of flow control information toward the original sender.

defined_flags = []
The flags defined for WINDOW_UPDATE frames.

parse_body(*data: memoryview*) → None
Given the body of a frame, parses it into frame data. This populates the non-header parts of the frame: that is, it does not populate the stream ID or flags.

Parameters **data** – A memoryview object containing the body data of the frame. Must not contain *more* data than the length returned by *parse_frame_header*.

type = 8
The type byte defined for WINDOW_UPDATE frames.

window_increment = None
The amount the flow control window is to be incremented.

class hyperframe.frame.**HeadersFrame**(*stream_id: int*, *data: bytes = b''*, ***kwargs*)
Bases: *hyperframe.frame.Padding*, *hyperframe.frame.Priority*, *hyperframe.frame.Frame*

The HEADERS frame carries name-value pairs. It is used to open a stream. HEADERS frames can be sent on a stream in the "open" or "half closed (remote)" states.

The HeadersFrame class is actually basically a data frame in this implementation, because of the requirement to control the sizes of frames. A header block fragment that doesn't fit in an entire HEADERS frame needs to be followed with CONTINUATION frames. From the perspective of the frame building code the header block is an opaque data segment.

data = None
The HPACK-encoded header block.

defined_flags = [Flag(name='END_STREAM', bit=1), Flag(name='END_HEADERS', bit=4), Flag
The flags defined for HEADERS frames.

parse_body(*data: memoryview*) → None
Given the body of a frame, parses it into frame data. This populates the non-header parts of the frame: that is, it does not populate the stream ID or flags.

Parameters **data** – A memoryview object containing the body data of the frame. Must not contain *more* data than the length returned by *parse_frame_header*.

type = 1
The type byte defined for HEADERS frames.

class hyperframe.frame.**ContinuationFrame**(*stream_id: int*, *data: bytes = b''*, ***kwargs*)
Bases: *hyperframe.frame.Frame*

The CONTINUATION frame is used to continue a sequence of header block fragments. Any number of CONTINUATION frames can be sent on an existing stream, as long as the preceding frame on the same stream is one of HEADERS, PUSH_PROMISE or CONTINUATION without the END_HEADERS flag set.

Much like the HEADERS frame, hyper treats this as an opaque data frame with different flags and a different type.

data = None
The HPACK-encoded header block.

defined_flags = [Flag(name='END_HEADERS', bit=4)]
The flags defined for CONTINUATION frames.

parse_body(*data: memoryview*) → None
Given the body of a frame, parses it into frame data. This populates the non-header parts of the frame: that is, it does not populate the stream ID or flags.

Parameters **data** – A memoryview object containing the body data of the frame. Must not contain *more* data than the length returned by *parse_frame_header*.

type = 9
The type byte defined for CONTINUATION frames.

class hyperframe.frame.**ExtensionFrame**(*type: int*, *stream_id: int*, *flag_byte: int = 0*, *body: bytes = b''*, ***kwargs*)
Bases: *hyperframe.frame.Frame*

ExtensionFrame is used to wrap frames which are not natively interpretable by hyperframe.

Although certain byte prefixes are ordained by specification to have certain contextual meanings, frames with other prefixes are not prohibited, and may be used to communicate arbitrary meaning between HTTP/2 peers.

Thus, hyperframe, rather than raising an exception when such a frame is encountered, wraps it in a generic frame to be properly acted upon by upstream consumers which might have additional context on how to use it.

New in version 5.0.0.

parse_body(*data: memoryview*) → None
Given the body of a frame, parses it into frame data. This populates the non-header parts of the frame: that is, it does not populate the stream ID or flags.

Parameters **data** – A memoryview object containing the body data of the frame. Must not contain *more* data than the length returned by *parse_frame_header*.

parse_flags(*flag_byte: int*) → None
For extension frames, we parse the flags by just storing a flag byte.

serialize() → bytes
A broad override of the serialize method that ensures that the data comes back out exactly as it came in. This should not be used in most user code: it exists only as a helper method if frames need to be reconstituted.

hyperframe.frame.**FRAMES = {0: <class 'hyperframe.frame.DataFrame'>, 1: <class 'hyperframe**
FRAMES maps the type byte for each frame to the class used to represent that frame.

2.1 Exceptions

class hyperframe.exceptions.**HyperframeError**
The base class for all exceptions for the hyperframe module.

New in version 6.0.0.

class hyperframe.exceptions.**UnknownFrameError**(*frame_type: int*, *length: int*)
A frame of unknown type was received.

Changed in version 6.0.0: Changed base class from *ValueError* to *HyperframeError*

frame_type = None
The type byte of the unknown frame that was received.

length = None
The length of the data portion of the unknown frame.

class hyperframe.exceptions.**InvalidPaddingError**
A frame with invalid padding was received.

Changed in version 6.0.0: Changed base class from *ValueError* to *HyperframeError*

class hyperframe.exceptions.**InvalidFrameError**
Parsing a frame failed because the data was not laid out appropriately.

New in version 3.0.2.

Changed in version 6.0.0: Changed base class from *ValueError* to *HyperframeError*

class hyperframe.exceptions.**InvalidDataError**
Content or data of a frame was is invalid or violates the specification.

New in version 6.0.0.

Index

I

L

M

O

P

R

S

T

U

W