websocket-client

Release 1.8.0

liris

Apr 23, 2024

INTRODUCTION:

CHAPTER

ONE

INSTALLATION

You can use either `pip install websocket-client` or `pip install -e .` to install this library.

CHAPTER

TWO

GETTING STARTED

The quickest way to get started with this library is to use the _wsdump.py script. For an easy example, run the following:

```
python _wsdump.py ws://echo.websocket.events/ -t "hello world"
```

The above command will provide you with an interactive terminal to communicate with the echo.websocket.events server. This server will echo back any message you send it.

The wsdump.py script has many additional options too, so it's a great way to try using this library without writing any custom code. The output of `python wsdump.py -h` is seen below, showing the additional options available.

```
python wsdump.py -h
usage: wsdump.py [-h] [-p PROXY] [-v [VERBOSE]] [-n] [-r]
                 [-s [SUBPROTOCOLS [SUBPROTOCOLS ...]]] [-o ORIGIN]
                 [--eof-wait EOF_WAIT] [-t TEXT] [--timings]
                 [--headers HEADERS]
                 ws_url

WebSocket Simple Dump Tool

positional arguments:
  ws_url                websocket url. ex. ws://echo.websocket.events/

optional arguments:
  -h, --help            show this help message and exit
  -p PROXY, --proxy PROXY
                        proxy url. ex. http://127.0.0.1:8080
  -v [VERBOSE], --verbose [VERBOSE]
                        set verbose mode. If set to 1, show opcode. If set to
                        2, enable to trace websocket module
  -n, --nocert          Ignore invalid SSL cert
  -r, --raw             raw output
  -s [SUBPROTOCOLS [SUBPROTOCOLS ...]], --subprotocols [SUBPROTOCOLS [SUBPROTOCOLS ...]]
                        Set subprotocols
  -o ORIGIN, --origin ORIGIN
                        Set origin
  --eof-wait EOF_WAIT   wait time(second) after 'EOF' received.
  -t TEXT, --text TEXT  Send initial text
  --timings             Print timings in seconds
  --headers HEADERS     Set custom headers. Use ',' as separator
```

CHAPTER

THREE

EXAMPLES

3.1 Creating Your First WebSocket Connection

If you want to connect to a websocket without writing any code yourself, you can try out the *Getting Started* wsdump.py script and the examples/ directory files.

You can create your first custom connection with this library using one of the simple examples below. Note that the first WebSocket example is best for a short-lived connection, while the WebSocketApp example is best for a long-lived connection.

WebSocket example

```
>>> import websocket
>>> ws = websocket.WebSocket()
>>> ws.connect("ws://echo.websocket.events")
>>> ws.send("Hello, Server")
19
>>> print(ws.recv())
echo.websocket.events sponsored by Lob.com
>>> ws.close()
```

WebSocketApp example

```
>>> import websocket
>>> def on_message(wsapp, message):
...     print(message)
>>> wsapp = websocket.WebSocketApp("wss://testnet-explorer.binance.org/ws/block", on_
↪message=on_message)
>>> wsapp.run_forever()
```

3.2 Debug and Logging Options

When you're first writing your code, you will want to make sure everything is working as you planned. The easiest way to view the verbose connection information is the use `websocket.enableTrace(True)`. For example, the following example shows how you can verify that the proper Origin header is set.

```
import websocket

websocket.enableTrace(True)
ws = websocket.WebSocket()
```

(continues on next page)

(continued from previous page)

```
ws.connect("ws://echo.websocket.events/", origin="testing_websockets.com")
ws.send("Hello, Server")
print(ws.recv())
ws.close()
```

The output you will see will look something like this:

```
--- request header ---
GET / HTTP/1.1
Upgrade: websocket
Host: echo.websocket.events
Origin: testing_websockets.com
Sec-WebSocket-Key: GnuCGEiF3OuyRESXiVnsAQ==
Sec-WebSocket-Version: 13
Connection: Upgrade


-----------------------
--- response header ---
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: websocket
Sec-Websocket-Accept: wvhwrjThsVAyr/V4Hzn5tWMSomI=
Via: 1.1 vegur
-----------------------
++Sent raw: b'\x81\x8d\xd4\xda9\xee\x9c\xbfU\x82\xbb\xf6\x19\xbd\xb1\xa80\x8b\xa6'
++Sent decoded: fin=1 opcode=1 data=b'Hello, Server'
19
++Rcv raw: b'\x81*echo.websocket.events sponsored by Lob.com'
++Rcv decoded: fin=1 opcode=1 data=b'echo.websocket.events sponsored by Lob.com'
echo.websocket.events sponsored by Lob.com
++Sent raw: b'\x88\x82\xc9\x8c\x14\x99\xcad'
++Sent decoded: fin=1 opcode=8 data=b'\x03\xe8'
```

3.3 Using websocket-client with “with” statements

It is possible to use “with” statements, as outlined in PEP 343, to help manage the closing of WebSocket connections after a message is received. Below is one example of this being done with a short-lived connection:

Short-lived WebSocket using “with” statement

```
>>> from contextlib import closing
>>> from websocket import create_connection
>>> with closing(create_connection("wss://testnet-explorer.binance.org/ws/block")) as␣
↪conn:
...     print(conn.recv())

# Connection is now closed
```

3.4 Connection Options

After you can establish a basic WebSocket connection, customizing your connection using specific options is the next step. Fortunately, this library provides many options you can configure, such as:

- "Host" header value
- "Cookie" header value
- "Origin" header value
- WebSocket subprotocols
- Custom headers
- SSL or hostname verification
- Timeout value

For a more detailed list of the options available for the different connection methods, check out the source code comments for each:

- **WebSocket().connect() option docs**
  - Related: WebSocket.create_connection() option docs
- **WebSocketApp() option docs**
  - Related: WebSocketApp.run_forever docs

3.4.1 Setting Common Header Values

To modify the `Host`, `Origin`, `Cookie`, or `Sec-WebSocket-Protocol` header values of the WebSocket handshake request, pass the `host`, `origin`, `cookie`, or `subprotocols` options to your WebSocket connection. The first two examples show the Host, Origin, and Cookies headers being set, while the Sec-WebSocket-Protocol header is set separately in the following example. For debugging, remember that it is helpful to enable *Debug and Logging Options*.

WebSocket common headers example

```
>>> import websocket

>>> ws = websocket.WebSocket()
>>> ws.connect("ws://echo.websocket.events", cookie="chocolate",
... origin="testing_websockets-client.com", host="echo.websocket.events")
```

WebSocketApp common headers example

```
>>> import websocket

>>> def on_message(wsapp, message):
...     print(message)
>>> wsapp = websocket.WebSocketApp("wss://testnet-explorer.binance.org/ws/block",
... cookie="chocolate", on_message=on_message)
>>> wsapp.run_forever(origin="testing_websockets.com", host="127.0.0.1")
```

WebSocket subprotocols example

Use this to specify STOMP, WAMP, MQTT, or other values of the "Sec-WebSocket-Protocol" header. Be aware that websocket-client does not include support for these protocols, so your code must handle the data sent over the WebSocket connection.

```
>>> import websocket

>>> ws = websocket.WebSocket()
>>> ws.connect("wss://ws.kraken.com", subprotocols=["mqtt"])
```

WebSocketApp subprotocols example

```
>>> import websocket

>>> def on_message(wsapp, message):
...     print(message)
>>> wsapp = websocket.WebSocketApp("wss://ws.kraken.com",
... subprotocols=["STOMP"], on_message=on_message)
>>> wsapp.run_forever()
```

3.4.2 Suppress Origin Header

There is a special `suppress_origin` option that can be used to remove the `Origin` header from connection handshake requests. The below examples illustrate how this can be used. For debugging, remember that it is helpful to enable *Debug and Logging Options*.

WebSocket suppress origin example

```
>>> import websocket

>>> ws = websocket.WebSocket()
>>> ws.connect("ws://echo.websocket.events", suppress_origin=True)
```

WebSocketApp suppress origin example

```
>>> import websocket

>>> def on_message(wsapp, message):
...     print(message)
>>> wsapp = websocket.WebSocketApp("wss://testnet-explorer.binance.org/ws/block",
... on_message=on_message)
>>> wsapp.run_forever(suppress_origin=True)
```

3.4.3 Setting Custom Header Values

Setting custom header values, other than `Host`, `Origin`, `Cookie`, or `Sec-WebSocket-Protocol` (which are addressed above), in the WebSocket handshake request is similar to setting common header values. Use the `header` option to provide custom header values in a list or dict. For debugging, remember that it is helpful to enable *Debug and Logging Options*. There is no built-in support for "Sec-WebSocket-Extensions" header values as defined in RFC 7692.

WebSocket custom headers example

```
>>> import websocket

>>> ws = websocket.WebSocket()
>>> ws.connect("ws://echo.websocket.events",
... header={"CustomHeader1":"123", "NewHeader2":"Test"})
```

WebSocketApp custom headers example

```
>>> import websocket

>>> def on_message(wsapp, message):
...     print(message)
>>> wsapp = websocket.WebSocketApp("wss://testnet-explorer.binance.org/ws/block",
... header={"CustomHeader1":"123", "NewHeader2":"Test"}, on_message=on_message)
>>> wsapp.run_forever()
```

3.4.4 Disabling SSL or Hostname Verification

See the relevant *FAQ* page for instructions.

3.4.5 Using a Custom Class

You can also write your own class for the connection, if you want to handle the nitty-gritty connection details yourself.

```
>>> import socket
>>> from websocket import create_connection, WebSocket

>>> class MyWebSocket(WebSocket):
...     def recv_frame(self):
...         frame = super().recv_frame()
...         print('yay! I got this frame: ', frame)
...         return frame
>>> ws = create_connection("ws://echo.websocket.events/",
... sockopt=((socket.IPPROTO_TCP, socket.TCP_NODELAY, 1),), class_=MyWebSocket)
```

3.4.6 Setting Timeout Value

The _socket.py file contains the functions `setdefaulttimeout()` and `getdefaulttimeout()`. These two functions set the global `_default_timeout` value, which sets the socket timeout value (in seconds). These two functions should not be confused with the similarly named `settimeout()` and `gettimeout()` functions found in the _core.py file. With WebSocketApp, the `run_forever()` function gets assigned the timeout from getdefaulttimeout(). When the timeout value is reached, the exception WebSocketTimeoutException is triggered by the _socket.py `send()` and `recv()` functions. Additional timeout values can be found in other locations in this library, including the `close()` function of the WebSocket class and the `create_connection()` function of the WebSocket class.

The WebSocket timeout example below shows how an exception is triggered after no response is received from the server after 5 seconds.

WebSocket timeout example

```
>>> import websocket

>>> ws = websocket.WebSocket()
>>> ws.connect("ws://echo.websocket.events", timeout=5)
>>> # ws.send("Hello, Server") # Commented out to trigger WebSocketTimeoutException
>>> print(ws.recv())
# Program should end with a WebSocketTimeoutException
```

The WebSocketApp timeout example works a bit differently than the WebSocket example. Because WebSocketApp handles long-lived connections, it does not timeout after a certain amount of time without receiving a message. Instead, a timeout is triggered if no connection response is received from the server after the timeout interval (5 seconds in the example below).

WebSocketApp timeout example

```
>>> import websocket

>>> def on_error(wsapp, err):
...     print("EXAMPLE error encountered: ", err)
>>> websocket.setdefaulttimeout(5)
>>> wsapp = websocket.WebSocketApp("ws://nexus-websocket-a.intercom.io",
... on_error=on_error)
>>> wsapp.run_forever()
# Program should print a "timed out" error message
```

3.4.7 Connecting through a proxy

websocket-client supports proxied connections. The supported proxy protocols are HTTP, SOCKS4, SOCKS4a, SOCKS5, and SOCKS5h. If you want to route DNS requests through the proxy, use SOCKS4a or SOCKS5h. The proxy protocol should be specified in lowercase to the `proxy_type` parameter. The example below shows how to connect through a HTTP or SOCKS proxy. Proxy authentication is supported with the `http_proxy_auth` parameter, which should be a tuple of the username and password. Be aware that the current implementation of websocket-client uses the "CONNECT" method for HTTP proxies (though soon the HTTP proxy handling will use the same `python_socks` library currently enabled only for SOCKS proxies), and the HTTP proxy server must allow the "CONNECT" method. For example, the squid HTTP proxy only allows the "CONNECT" method on HTTPS ports by default. You may encounter problems if using SSL/TLS with your proxy.

WebSocket HTTP proxy with authentication example

```
import websocket

ws = websocket.WebSocket()
ws.connect("ws://echo.websocket.events",
  http_proxy_host="127.0.0.1", http_proxy_port="8080",
  proxy_type="http", http_proxy_auth=("username", "password123"))
ws.send("Hello, Server")
print(ws.recv())
ws.close()
```

WebSocket SOCKS4 (or SOCKS5) proxy example

```
import websocket

ws = websocket.WebSocket()
ws.connect("ws://echo.websocket.events",
  http_proxy_host="192.168.1.18", http_proxy_port="4444", proxy_type="socks4")
ws.send("Hello, Server")
print(ws.recv())
ws.close()
```

WebSocketApp proxy example

```
import websocket

ws = websocket.WebSocketApp("ws://echo.websocket.events")
wsapp.run_forever(proxy_type="socks5", http_proxy_host=proxy_ip, http_proxy_auth=(proxy_
↪username, proxy_password))
```

3.4.8 Connecting with Custom Sockets

You can also connect to a WebSocket server hosted on a specific socket using the `socket` option when creating your connection. Below is an example of using a unix domain socket.

```
import socket
from websocket import create_connection
my_socket = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
my_socket.connect("/path/to/my/unix.socket")

ws = create_connection("ws://localhost/", # Dummy URL
                       socket = my_socket,
                       sockopt=((socket.SOL_SOCKET, socket.SO_KEEPALIVE, 1),))
```

Other socket types can also be used. The following example is for a AF_INET (IP address) socket.

```
import socket
from websocket import create_connection
my_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
my_socket.bind(("172.18.0.1", 3002))
my_socket.connect()

ws = create_connection("ws://127.0.0.1/", # Dummy URL
                       socket = my_socket)
```

3.4.9 Post-connection Feature Summary

Autobahn|TestSuite is an independent automated test suite to verify the compliance of WebSocket implementations.

Running the test suite against this library will produce a summary report of the conformant features that have been implemented.

A recently-run autobahn report (available as an .html file) is available in the /compliance directory.

3.4.10 Ping/Pong Usage

The WebSocket specification defines ping and pong message opcodes as part of the protocol. These can serve as a way to keep a connection active even if data is not being transmitted.

Pings may be sent in either direction. If the client receives a ping, a pong reply will be automatically sent.

However, if a blocking event is happening, there may be some issues with ping/pong. Below are examples of how ping and pong can be sent by this library.

You can get additional debugging information by using Wireshark to view the ping and pong messages being sent. In order for Wireshark to identify the WebSocket protocol properly, it should observe the initial HTTP handshake and the

HTTP 101 response in cleartext (without encryption) - otherwise the WebSocket messages may be categorized as TCP or TLS messages. For debugging, remember that it is helpful to enable *Debug and Logging Options*.

WebSocket ping/pong example

This example is best for a quick test where you want to check the effect of a ping, or where situations where you want to customize when the ping is sent.

```
>>> import websocket
>>> websocket.enableTrace(True)

>>> ws = websocket.WebSocket()
>>> ws.connect("ws://echo.websocket.events")
>>> ws.ping()
>>> ws.ping("This is an optional ping payload")
>>> ws.close()
```

WebSocketApp ping/pong example

This example, and `run_forever()` in general, is better for long-lived connections.

In this example, if a ping is received and a pong is sent in response, then the client is notified via `on_ping()`.

Further, a ping is transmitted every 60 seconds. If a pong is received, then the client is notified via `on_pong()`. If no pong is received within 10 seconds, then `run_forever()` will exit with a `WebSocketTimeoutException`.

```
>>> import websocket

>>> def on_message(wsapp, message):
...     print(message)
>>> def on_ping(wsapp, message):
...     print("Got a ping! A pong reply has already been automatically sent.")
>>> def on_pong(wsapp, message):
...     print("Got a pong! No need to respond")
>>> wsapp = websocket.WebSocketApp("wss://testnet-explorer.binance.org/ws/block",
... on_message=on_message, on_ping=on_ping, on_pong=on_pong)
>>> wsapp.run_forever(ping_interval=60, ping_timeout=10, ping_payload="This is an␣
↪optional ping payload")
```

3.4.11 Sending Connection Close Status Codes

RFC6455 defines various status codes that can be used to identify the reason for a close frame ending a connection. These codes are defined in the websocket/_abnf.py file. To view the code used to close a connection, you can *enable logging* to view the status code information. You can also specify your own status code in the .close() function, as seen in the examples below. Specifying a custom status code is necessary when using the custom status code values between 3000-4999.

WebSocket sending close() status code example

```
>>> import websocket
>>> websocket.enableTrace(True)

>>> ws = websocket.WebSocket()
>>> ws.connect("ws://echo.websocket.events")
>>> ws.send("Hello, Server")
```

(continues on next page)

(continued from previous page)

```
19
>>> print(ws.recv())
echo.websocket.events sponsored by Lob.com
>>> ws.close(websocket.STATUS_PROTOCOL_ERROR)
# Alternatively, use ws.close(status=1002)
```

WebSocketApp sending close() status code example

```
>>> import websocket
>>> websocket.enableTrace(True)

>>> def on_message(wsapp, message):
...     print(message)
...     wsapp.close(status=websocket.STATUS_PROTOCOL_ERROR) # Alternatively, use wsapp.
↪close(status=1002)
>>> wsapp = websocket.WebSocketApp("wss://testnet-explorer.binance.org/ws/block", on_
↪message=on_message)
>>> wsapp.run_forever(skip_utf8_validation=True)
```

3.4.12 Receiving Connection Close Status Codes

The RFC6455 spec states that it is optional for a server to send a close status code when closing a connection. The RFC refers to these codes as WebSocket Close Code Numbers, and their meanings are described in the RFC. It is possible to view this close code, if it is being sent, to understand why the connection is being close. One option to view the code is to *enable logging* to view the status code information. If you want to use the close status code in your program, examples are shown below for how to do this.

WebSocket receiving close status code example

```
>>> import websocket
>>> import struct

>>> websocket.enableTrace(True)
>>> ws = websocket.WebSocket()
>>> ws.connect("wss://tsock.us1.twilio.com/v3/wsconnect")
>>> ws.send("Hello")
11
>>> resp_opcode, msg = ws.recv_data()
>>> print("Response opcode: " + str(resp_opcode))
>>> if resp_opcode == 8 and len(msg) >= 2:
...     print("Response close code: " + str(struct.unpack("!H", msg[0:2])[0]))
...     print("Response message: " + str(msg[2:]))
... else:
...     print("Response message: " + str(msg))
```

WebSocketApp receiving close status code example

```
>>> import websocket
>>> websocket.enableTrace(True)

>>> def on_close(wsapp, close_status_code, close_msg):
...     # Because on_close was triggered, we know the opcode = 8
```

(continues on next page)

(continued from previous page)

```
...     print("on_close args:")
...     if close_status_code or close_msg:
...         print("close status code: " + str(close_status_code))
...         print("close message: " + str(close_msg))
>>> def on_open(wsapp):
...     wsapp.send("Hello")
>>> wsapp = websocket.WebSocketApp("wss://tsock.us1.twilio.com/v3/wsconnect", on_
↪close=on_close, on_open=on_open)
>>> wsapp.run_forever()
```

3.4.13 Customizing frame mask

WebSocket frames use masking with a random value to add entropy. The masking value in websocket-client is normally set using os.urandom in the websocket/_abnf.py file. However, this value can be customized as you wish. One use case, outlined in issue #473, is to set the masking key to a null value to make it easier to decode the messages being sent and received. This is effectively the same as "removing" the mask, though the mask cannot be fully "removed" because it is a part of the WebSocket frame. Tools such as Wireshark can automatically remove masking from payloads to decode the payload message, but it may be easier to skip the demasking step in your custom project.

WebSocket custom masking key code example

```
>>> import websocket
>>> websocket.enableTrace(True)

>>> def zero_mask_key(_):
...     return "\x00\x00\x00\x00"
>>> ws = websocket.WebSocket()
>>> ws.set_mask_key(zero_mask_key)
>>> ws.connect("ws://echo.websocket.events")
>>> ws.send("Hello, Server")
>>> print(ws.recv())
>>> ws.close()
```

WebSocketApp custom masking key code example

```
>>> import websocket
>>> websocket.enableTrace(True)

>>> def zero_mask_key(_):
...     return "\x00\x00\x00\x00"
>>> def on_message(wsapp, message):
...     print(message)
>>> wsapp = websocket.WebSocketApp("wss://testnet-explorer.binance.org/ws/block", on_
↪message=on_message, get_mask_key=zero_mask_key)
>>> wsapp.run_forever()
```

3.4.14 Customizing opcode

WebSocket frames contain an opcode, which defines whether the frame contains text data, binary data, or is a special frame. The different opcode values are defined in RFC6455 section 11.8. Although the text opcode, 0x01, is the most commonly used value, the websocket-client library makes it possible to customize which opcode is used.

WebSocket custom opcode code example

```
>>> import websocket
>>> websocket.enableTrace(True)

>>> ws = websocket.WebSocket()
>>> ws.connect("ws://echo.websocket.events")
>>> ws.send("Hello, Server", websocket.ABNF.OPCODE_TEXT)
>>> print(ws.recv())
>>> ws.send("This is a ping", websocket.ABNF.OPCODE_PING)
>>> ws.close()
```

WebSocketApp custom opcode code example

The WebSocketApp class contains different functions to handle different message opcodes. For instance, on_close, on_ping, on_pong, on_cont_message. One drawback of the current implementation (as of May 2021) is the lack of binary support for WebSocketApp, as noted by issue #351.

```
>>> import websocket
>>> websocket.enableTrace(True)

>>> def on_open(wsapp):
...     wsapp.send("Hello")

>>> def on_message(ws, message):
...     print(message)
...     ws.send("Send a ping", websocket.ABNF.OPCODE_PING)

>>> def on_pong(wsapp, message):
...     print("Got a pong! No need to respond")

>>> wsapp = websocket.WebSocketApp("ws://echo.websocket.events", on_open=on_open, on_
↪message=on_message, on_pong=on_pong)
>>> wsapp.run_forever()
```

3.5 Dispatching Multiple WebSocketApps

You can use an asynchronous dispatcher such as rel to run multiple WebSocketApps in the same application without resorting to threads.

WebSocketApp asynchronous dispatcher code example

```
>>> import websocket, rel

>>> addr = "wss://api.gemini.com/v1/marketdata/%s"
>>> for symbol in ["BTCUSD", "ETHUSD", "ETHBTC"]:
...     ws = websocket.WebSocketApp(addr % (symbol,), on_message=lambda w, m : print(m))
```

(continues on next page)

(continued from previous page)

```
...     ws.run_forever(dispatcher=rel, reconnect=3)
>>> rel.signal(2, rel.abort)  # Keyboard Interrupt
>>> rel.dispatch()
```

3.6 README Examples

The examples are found in the README and are copied here for sphinx doctests to verify they run without errors.

Long-lived Connection

```
>>> import websocket
>>> import _thread
>>> import time
>>> import rel

>>> def on_message(ws, message):
...     print(message)

>>> def on_error(ws, error):
...     print(error)

>>> def on_close(ws, close_status_code, close_msg):
...     print("### closed ###")

>>> def on_open(ws):
...     print("Opened connection")

>>> if __name__ == "__main__":
...     websocket.enableTrace(True)
...     ws = websocket.WebSocketApp("wss://api.gemini.com/v1/marketdata/BTCUSD",
...                                 on_open=on_open,
...                                 on_message=on_message,
...                                 on_error=on_error,
...                                 on_close=on_close)

>>> ws.run_forever(dispatcher=rel, reconnect=5)  # Set dispatcher to automatic␣
↪reconnection, 5 second reconnect delay if connection closed unexpectedly
>>> rel.signal(2, rel.abort)  # Keyboard Interrupt
<Signal Object | Callback:"abort">
>>> rel.dispatch()
```

Short-lived Connection

```
>>> from websocket import create_connection

>>> ws = create_connection("ws://echo.websocket.events/")
>>> print(ws.recv())
echo.websocket.events sponsored by Lob.com
>>> print("Sending 'Hello, World'...")
Sending 'Hello, World'...
>>> ws.send("Hello, World")
```

(continues on next page)

(continued from previous page)

```
18
>>> print("Sent")
Sent
>>> print("Receiving...")
Receiving...
>>> result =  ws.recv()
>>> print("Received '%s'" % result)
Received ...
>>> ws.close()
```

3.7 Real-world Examples

Other projects that depend on websocket-client may be able to illustrate more complex use cases for this library. A list of 600+ dependent projects is found on libraries.io, and a few of the projects using websocket-client are listed below:

- https://github.com/apache/airflow
- https://github.com/docker/docker-py
- https://github.com/scrapinghub/slackbot
- https://github.com/slackapi/python-slack-sdk
- https://github.com/wee-slack/wee-slack
- https://github.com/aluzzardi/wssh/
- https://github.com/llimllib/limbo
- https://github.com/miguelgrinberg/python-socketio
- https://github.com/invisibleroads/socketIO-client
- https://github.com/s4w3d0ff/python-poloniex
- https://github.com/Ape/samsungctl
- https://github.com/xchwarze/samsung-tv-ws-api
- https://github.com/andresriancho/websocket-fuzzer

CHAPTER

FOUR

THREADING

4.1 Importance of enable_multithread

The `enable_multithread` variable should be set to `True` when working with multiple threads. If `enable_multithread` is not set to `True`, websocket-client will act asynchronously and not be thread safe. This variable should be enabled by default starting with the 1.1.0 release, but had a default value of `False` in older versions. See issues #591 and #507 for related issues.

4.2 asyncio library usage

Issue #496 indicates that websocket-client is not compatible with asyncio. The engine-io project, which is used in a popular socket-io client, specifically uses websocket-client as a dependency only in places where asyncio is not used. If asyncio is an important part of your project, you might consider using another websockets library. However, some simple use cases, such as asynchronously receiving data, may be a place to use asyncio. Here is one snippet showing how asynchronous listening might be implemented.

```
async def mylisten(ws):
    result = await asyncio.get_event_loop().run_in_executor(None, ws.recv)
    return result
```

4.3 threading library usage

The websocket-client library has some built-in threading support provided by the `threading` library. You will see `import threading` in some of this project's code. The echoapp_client.py example is a good illustration of how `threading` can be used in the websocket-client library. Another example is found in an external site's documentation, which demonstrates using the _thread library, which is lower level than the threading library.

4.4 Possible issues with threading

Further investigation into using the `threading` module is seen in issue #612 which illustrates one situation where using the threading module can impact the observed behavior of this library. The first code example below does not trigger the `on_close()` function, but the second code example does trigger the `on_close()` function. The highlighted rows show the lines added exclusively in the second example. This threading approach is identical to the echoapp_client.py example. However, further testing found that some WebSocket servers, such as ws://echo.websocket.events, do not trigger the `on_close()` function.

NOT working on_close() example, without threading

```
import websocket

websocket.enableTrace(True)

def on_open(ws):
    ws.send("hi")

def on_message(ws, message):
    print(message)
    ws.close()
    print("Message received...")

def on_close(ws, close_status_code, close_msg):
    print(">>>>>>CLOSED")

wsapp = websocket.WebSocketApp("wss://api.bitfinex.com/ws/1", on_open=on_open, on_
↪message=on_message, on_close=on_close)
wsapp.run_forever()
```

Working on_close() example, with threading

```
import websocket
import threading

websocket.enableTrace(True)

def on_open(ws):
    ws.send("hi")

def on_message(ws, message):
    def run(*args):
        print(message)
        ws.close()
        print("Message received...")
```

(continues on next page)

(continued from previous page)

```
    threading.Thread(target=run).start()

def on_close(ws, close_status_code, close_msg):
    print(">>>>>>CLOSED")

wsapp = websocket.WebSocketApp("wss://api.bitfinex.com/ws/1", on_open=on_open, on_
↪message=on_message, on_close=on_close)
wsapp.run_forever()
```

Another example of code that does not trigger *on_close* is below. The fix is to use a timer.

NOT working on_close() example, with sleep delay

```
import websocket
from threading import Thread
import sys
import time

def on_close(ws, close_status_code, close_msg):
    print("### closed ###")

def on_message(ws, message):
    print(message)
    time.sleep(2)

if __name__ == "__main__":
    websocket.enableTrace(True)
    if len(sys.argv) < 2:
        host = "ws://echo.websocket.events/"
    else:
        host = sys.argv[1]
    ws = websocket.WebSocketApp(host,
                                on_message=on_message,
                                on_close=on_close)
    Thread(target=ws.run_forever).start()
    time.sleep(1)
    ws.close()
```

Working on_close() example, with threading delay

```
import websocket
from threading import Thread
import sys
import time

def on_close(ws, close_status_code, close_msg):
    print("### closed ###")

def on_message(ws, message):
    print(message)
    timer = threading.Timer(2, ws.close)
    timer.start()
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    websocket.enableTrace(True)
    if len(sys.argv) < 2:
        host = "ws://echo.websocket.events/"
    else:
        host = sys.argv[1]
    ws = websocket.WebSocketApp(host,
                                on_message=on_message,
                                on_close=on_close)
    Thread(target=ws.run_forever).start()
    time.sleep(1)
    ws.close()
```

In part because threading is hard, but also because this project has (until recently) lacked any threading documentation, there are many issues on this topic, including:

- #562
- #580
- #591

CHAPTER

FIVE

FAQ

5.1 What about Python 2 support?

Release 0.59.0 was the last main release supporting Python 2. All future releases 1.X.X and beyond will only support Python 3.

5.2 Why is this library slow?

The `send` and `validate_utf8` methods are very slow in pure Python. You can disable UTF8 validation in this library (and receive a performance enhancement) with the `skip_utf8_validation` parameter. If you want to get better performance, install wsaccel. While websocket-client does not depend on wsaccel, it will be used if available. wsaccel doubles the speed of UTF8 validation and offers a very minor 10% performance boost when masking the payload data as part of the `send` process. Numpy used to be a suggested alternative, but issue #687 found it didn't help.

5.3 How to troubleshoot an unclear callback error?

To get more information about a callback error, you can specify a custom `on_error()` function that raises errors to provide more information. Sample code of such a solution is shown below, although the example URL provided will probably not trigger an error under normal circumstances. Issue #377 discussed this topic previously.

```
>>> import websocket
>>>
>>> def on_message(ws, message):
...     print(message)
>>> def on_error(wsapp, err):
...     print("Got a an error: ", err)
>>> wsapp = websocket.WebSocketApp("ws://echo.websocket.events/",
... on_message = on_message,
... on_error=on_error)
>>> wsapp.run_forever()
```

5.4 How to solve the "connection is already closed" error?

The WebSocketConnectionClosedException, which returns the message "Connection is already closed.", occurs when a WebSocket function such as `send()` or `recv()` is called but the WebSocket connection is already closed. One way to handle exceptions in Python is by using a try/except statement, which allows you to control what your program does if the WebSocket connection is closed when you try to use it. In order to properly carry out further functions with your WebSocket connection after the connection has closed, you will need to reconnect the WebSocket, using `connect()` or `create_connection()` (from the _core.py file). The WebSocketApp `run_forever()` function automatically tries to reconnect when the connection is lost if a dispatcher parameter is provided to the `run_forever()` function.

5.5 What's going on with the naming of this library?

To install this library, you use `pip install websocket-client`, while `import websocket` imports this library, and PyPi lists the package as `websocket_client`. Why is it so confusing? To see the original issue about the choice of `import websocket`, see issue #60 and to read about websocket-client vs. websocket_client, see issue #147

5.6 Is WebSocket Compression using the permessage-deflate extension supported?

No, RFC 7692 for WebSocket Compression is unfortunately not supported by the websocket-client library at this time. You can view the currently supported WebSocket features in the latest autobahn compliance HTML report, found under the compliance folder. If you use the `Sec-WebSocket-Extensions: permessage-deflate` header with websocket-client, you will probably encounter errors, such as the ones described in issue #314.

5.7 I get the error 'utf8' codec can't decode byte 0x81 in position 0

This error is caused when you receive a character that is not a UTF-8 character, so the UTF-8 decoding fails. You can set *skip_utf8_validation* to false, but if this does not work, you can change the encoding to ISO-8859-1 which was a workaround suggested in [issue 481](https://github.com/websocket-client/websocket-client/issues/481#issuecomment-1112506666).

5.8 If a connection is re-established after getting disconnected, does the new connection continue where the previous one dropped off?

The answer to this question depends on how the WebSocket server handles new connections. If the server keeps a list of recently dropped WebSocket connection sessions, then it may allow you to recontinue your WebSocket connection where you left off before disconnecting. However, this requires extra effort from the server and may create security issues. For these reasons it is rare to encounter such a WebSocket server. The server would need to identify each connecting client with authentication and keep track of which data was received using a method like TCP's SYN/ACK. That's a lot of overhead for a lightweight protocol! Both HTTP and WebSocket connections use TCP sockets, and when a new WebSocket connection is created, it uses a new TCP socket. Therefore, at the TCP layer, the default behavior is to give each WebSocket connection a separate TCP socket. This means the re-established connection after a disconnect is the same as a completely new connection. Another way to think about this is: what should the server do if you create two WebSocket connections from the same client to the same server? The easiest solution for the server is to treat each

connection separately, unless the WebSocket uses an authentication method to identify individual clients connecting to the server.

5.9 What is the difference between recv_frame(), recv_data_frame(), and recv_data()?

This is explained in issue #688. This information is useful if you do NOT want to use `run_forever()` but want to have similar functionality. In short, `recv_data()` is the recommended choice and you will need to manage ping/pong on your own, while `run_forever()` handles ping/pong by default.

5.10 How to disable ssl cert verification?

Set the sslopt to `{"cert_reqs": ssl.CERT_NONE}`. The same sslopt argument is provided for all examples seen below.

WebSocketApp example

```
>>> import websocket, ssl
>>> ws = websocket.WebSocketApp("wss://echo.websocket.events")
>>> ws.run_forever(sslopt={"cert_reqs": ssl.CERT_NONE})
```

create_connection example

```
>>> import websocket, ssl
>>> ws = websocket.create_connection("wss://echo.websocket.events",
... sslopt={"cert_reqs": ssl.CERT_NONE})
```

WebSocket example

```
>>> import websocket, ssl
>>> ws = websocket.WebSocket(sslopt={"cert_reqs": ssl.CERT_NONE})
>>> ws.connect("wss://echo.websocket.events")
```

5.11 How to disable hostname verification?

Please set sslopt to `{"check_hostname": False}`. (since v0.18.0)

WebSocketApp example

```
>>> import websocket
>>> ws = websocket.WebSocketApp("wss://echo.websocket.events")
>>> ws.run_forever(sslopt={"check_hostname": False})
```

create_connection example

```
>>> import websocket
>>> ws = websocket.create_connection("wss://echo.websocket.events",
... sslopt={"check_hostname": False})
```

WebSocket example

```
>>> import websocket
>>> ws = websocket.WebSocket(sslopt={"check_hostname": False})
>>> ws.connect("wss://echo.websocket.events")
```

5.12 What else can I do with sslopts?

The `sslopt` parameter is a dictionary to which the following keys can be assigned:

- `certfile`, `keyfile`, `password` (see SSLContext.load_cert_chain)
- `ecdh_curve` (see SSLContext.set_ecdh_curve)
- `ciphers` (see SSLContext.set_ciphers)
- `cert_reqs` (see SSLContext.verify_mode)
- `ssl_version` (see SSLContext.protocol)
- `ca_certs`, `ca_cert_path` (see SSLContext.load_verify_locations)
- `check_hostname` (see SSLContext.check_hostname)
- `server_hostname`, `do_handshake_on_connect`, `suppress_ragged_eofs` (see SSLContext.wrap_socket)

If any other SSL options are required, they can be used by creating a custom SSLContext from the python SSL library and then passing that in as the value of the `context` key. (since v1.2.2)

For example, if you wanted to load all of the default CA verification certificates, but also add your own additional custom CAs (of which the certs are located in the file "my_extra_CAs.cer"), you could do this:

```
>>> import ssl
>>> my_context = ssl.create_default_context()
>>> my_context.load_verify_locations('my_extra_CAs.cer')
>>> ws.run_forever(sslopt={'context': my_context})
```

Note that when passing in a custom `context`, all of the other context-related options are ignored. In other words, only the `server_hostname`, `do_handshake_on_connect`, and `suppress_ragged_eofs` options can be used in conjunction with `context`.

5.13 How to enable SNI?

SNI support is available for Python 2.7.9+ and 3.2+. It will be enabled automatically whenever possible.

5.14 Why don't I receive all the server's message(s)?

Depending on how long your connection exists, it can help to ping the server to keep the connection alive. See issue #200 for possible solutions.

5.15 Using Subprotocols

The WebSocket RFC outlines the usage of subprotocols. The subprotocol can be specified as in the example below:

```
>>> import websocket
>>> ws = websocket.create_connection("ws://echo.websocket.events", subprotocols=["binary
↪", "base64"])
```

CHAPTER

SIX

CONTRIBUTING

Contributions are welcome! See this project's contributing guidelines

CHAPTER

SEVEN

ABOUT

The websocket-client project was started in 2011, but experienced a slowdown in development in 2019-2020. The original creator of this project was liris and the current maintainer (since 2021) is engn33r. The project is in the process of being rejuvenated, so any edits or suggestions are appreciated. No typo is too small for a pull request! See the *Contributing* page for more info.

CHAPTER

EIGHT

WEBSOCKET/_ABNF.PY

The _abnf.py file

class websocket._abnf.**ABNF**(*fin: int = 0*, *rsv1: int = 0*, *rsv2: int = 0*, *rsv3: int = 0*, *opcode: int = 1*, *mask_value: int = 1*, *data: str | bytes | None = ''*)

ABNF frame class. See http://tools.ietf.org/html/rfc5234 and http://tools.ietf.org/html/rfc6455#section-5.2

__init__(*fin: int = 0*, *rsv1: int = 0*, *rsv2: int = 0*, *rsv3: int = 0*, *opcode: int = 1*, *mask_value: int = 1*, *data: str | bytes | None = ''*) → None

Constructor for ABNF. Please check RFC for arguments.

static create_frame(*data: bytes | str*, *opcode: int*, *fin: int = 1*) → *ABNF*

Create frame to send text, binary and other data.

Parameters

- **data** (*str*) – data to send. This is string value(byte array). If opcode is OPCODE_TEXT and this value is unicode, data value is converted into unicode string, automatically.
- **opcode** (*int*) – operation code. please see OPCODE_MAP.
- **fin** (*int*) – fin flag. if set to 0, create continue fragmentation.

format() → bytes

Format this object to string(byte array) to send data to server.

static mask(*mask_key: str | bytes*, *data: str | bytes*) → bytes

Mask or unmask data. Just do xor for each byte

Parameters

- **mask_key** (*bytes or str*) – 4 byte mask.
- **data** (*bytes or str*) – data to mask/unmask.

validate(*skip_utf8_validation: bool = False*) → None

Validate the ABNF frame.

Parameters
skip_utf8_validation (*skip utf8 validation.*)

CHAPTER

NINE

WEBSOCKET/_APP.PY

The _app.py file

class websocket._app.**WebSocketApp**(*url: str*, *header: list | dict | Callable | None = None*, *on_open: Callable[[WebSocket], None] | None = None*, *on_reconnect: Callable[[WebSocket], None] | None = None*, *on_message: Callable[[WebSocket, Any], None] | None = None*, *on_error: Callable[[WebSocket, Any], None] | None = None*, *on_close: Callable[[WebSocket, Any, Any], None] | None = None*, *on_ping: Callable | None = None*, *on_pong: Callable | None = None*, *on_cont_message: Callable | None = None*, *keep_running: bool = True*, *get_mask_key: Callable | None = None*, *cookie: str | None = None*, *subprotocols: list | None = None*, *on_data: Callable | None = None*, *socket: socket | None = None*)

Higher level of APIs are provided. The interface is like JavaScript WebSocket object.

__init__(*url: str*, *header: list | dict | Callable | None = None*, *on_open: Callable[[WebSocket], None] | None = None*, *on_reconnect: Callable[[WebSocket], None] | None = None*, *on_message: Callable[[WebSocket, Any], None] | None = None*, *on_error: Callable[[WebSocket, Any], None] | None = None*, *on_close: Callable[[WebSocket, Any, Any], None] | None = None*, *on_ping: Callable | None = None*, *on_pong: Callable | None = None*, *on_cont_message: Callable | None = None*, *keep_running: bool = True*, *get_mask_key: Callable | None = None*, *cookie: str | None = None*, *subprotocols: list | None = None*, *on_data: Callable | None = None*, *socket: socket | None = None*) → None

WebSocketApp initialization

Parameters

- **url** (*str*) – Websocket url.
- **header** (*list or dict or Callable*) – Custom header for websocket handshake. If the parameter is a callable object, it is called just before the connection attempt. The returned dict or list is used as custom header value. This could be useful in order to properly setup timestamp dependent headers.
- **on_open** (*function*) – Callback object which is called at opening websocket. on_open has one argument. The 1st argument is this class object.
- **on_reconnect** (*function*) – Callback object which is called at reconnecting websocket. on_reconnect has one argument. The 1st argument is this class object.
- **on_message** (*function*) – Callback object which is called when received data. on_message has 2 arguments. The 1st argument is this class object. The 2nd argument is utf-8 data received from the server.

- **on_error** (*function*) – Callback object which is called when we get error. on_error has 2 arguments. The 1st argument is this class object. The 2nd argument is exception object.
- **on_close** (*function*) – Callback object which is called when connection is closed. on_close has 3 arguments. The 1st argument is this class object. The 2nd argument is close_status_code. The 3rd argument is close_msg.
- **on_cont_message** (*function*) – Callback object which is called when a continuation frame is received. on_cont_message has 3 arguments. The 1st argument is this class object. The 2nd argument is utf-8 string which we get from the server. The 3rd argument is continue flag. if 0, the data continue to next frame data
- **on_data** (*function*) – Callback object which is called when a message received. This is called before on_message or on_cont_message, and then on_message or on_cont_message is called. on_data has 4 argument. The 1st argument is this class object. The 2nd argument is utf-8 string which we get from the server. The 3rd argument is data type. ABNF.OPCODE_TEXT or ABNF.OPCODE_BINARY will be came. The 4th argument is continue flag. If 0, the data continue
- **keep_running** (*bool*) – This parameter is obsolete and ignored.
- **get_mask_key** (*function*) – A callable function to get new mask keys, see the WebSocket.set_mask_key's docstring for more information.
- **cookie** (*str*) – Cookie value.
- **subprotocols** (*list*) – List of available sub protocols. Default is None.
- **socket** (*socket*) – Pre-initialized stream socket.

close(***kwargs*) → None

Close websocket connection.

run_forever(*sockopt: tuple = None*, *sslopt: dict = None*, *ping_interval: float | int = 0*, *ping_timeout: float | int | None = None*, *ping_payload: str = ''*, *http_proxy_host: str = None*, *http_proxy_port: int | str = None*, *http_no_proxy: list = None*, *http_proxy_auth: tuple = None*, *http_proxy_timeout: float | None = None*, *skip_utf8_validation: bool = False*, *host: str = None*, *origin: str = None*, *dispatcher=None*, *suppress_origin: bool = False*, *proxy_type: str = None*, *reconnect: int = None*) → bool

Run event loop for WebSocket framework.

This loop is an infinite loop and is alive while websocket is available.

Parameters

- **sockopt** (*tuple*) – Values for socket.setsockopt. sockopt must be tuple and each element is argument of sock.setsockopt.
- **sslopt** (*dict*) – Optional dict object for ssl socket option.
- **ping_interval** (*int or float*) – Automatically send "ping" command every specified period (in seconds). If set to 0, no ping is sent periodically.
- **ping_timeout** (*int or float*) – Timeout (in seconds) if the pong message is not received.
- **ping_payload** (*str*) – Payload message to send with each ping.
- **http_proxy_host** (*str*) – HTTP proxy host name.
- **http_proxy_port** (*int or str*) – HTTP proxy port. If not set, set to 80.
- **http_no_proxy** (*list*) – Whitelisted host names that don't use the proxy.

- **http_proxy_timeout** (*int or float*) – HTTP proxy timeout, default is 60 sec as per python-socks.
- **http_proxy_auth** (*tuple*) – HTTP proxy auth information. tuple of username and password. Default is None.
- **skip_utf8_validation** (*bool*) – skip utf8 validation.
- **host** (*str*) – update host header.
- **origin** (*str*) – update origin header.
- **dispatcher** (*Dispatcher object*) – customize reading data from socket.
- **suppress_origin** (*bool*) – suppress outputting origin header.
- **proxy_type** (*str*) – type of proxy from: http, socks4, socks4a, socks5, socks5h
- **reconnect** (*int*) – delay interval when reconnecting

Returns

teardown – False if the *WebSocketApp* is closed or caught KeyboardInterrupt, True if any other exception was raised during a loop.

Return type

bool

send(*data: bytes | str*, *opcode: int = 1*) → None

send message

Parameters

- **data** (*str*) – Message to send. If you set opcode to OPCODE_TEXT, data must be utf-8 string or unicode.
- **opcode** (*int*) – Operation code of data. Default is OPCODE_TEXT.

send_bytes(*data: bytes | bytearray*) → None

Sends a sequence of bytes.

send_text(*text_data: str*) → None

Sends UTF-8 encoded text.

CHAPTER

TEN

WEBSOCKET/_CORE.PY

The _core.py file

class websocket._core.**WebSocket**(*get_mask_key=None*, *sockopt=None*, *sslopt=None*, *fire_cont_frame: bool = False*, *enable_multithread: bool = True*, *skip_utf8_validation: bool = False*, ***_)

Low level WebSocket interface.

This class is based on the WebSocket protocol draft-hixie-thewebsocketprotocol-76

We can connect to the websocket server and send/receive data. The following example is an echo client.

```
>>> import websocket
>>> ws = websocket.WebSocket()
>>> ws.connect("ws://echo.websocket.events")
>>> ws.recv()
'echo.websocket.events sponsored by Lob.com'
>>> ws.send("Hello, Server")
19
>>> ws.recv()
'Hello, Server'
>>> ws.close()
```

Parameters

- **get_mask_key** (*func*) – A callable function to get new mask keys, see the WebSocket.set_mask_key's docstring for more information.
- **sockopt** (*tuple*) – Values for socket.setsockopt. sockopt must be tuple and each element is argument of sock.setsockopt.
- **sslopt** (*dict*) – Optional dict object for ssl socket options. See FAQ for details.
- **fire_cont_frame** (*bool*) – Fire recv event for each cont frame. Default is False.
- **enable_multithread** (*bool*) – If set to True, lock send method.
- **skip_utf8_validation** (*bool*) – Skip utf8 validation.

__init__(*get_mask_key=None*, *sockopt=None*, *sslopt=None*, *fire_cont_frame: bool = False*, *enable_multithread: bool = True*, *skip_utf8_validation: bool = False*, ***_)

Initialize WebSocket object.

Parameters

sslopt (*dict*) – Optional dict object for ssl socket options. See FAQ for details.

abort()

Low-level asynchronous abort, wakes up other threads that are waiting in recv_*

close(*status: int = 1000*, *reason: bytes = b''*, *timeout: int = 3*)

Close Websocket object

Parameters

- **status** (*int*) – Status code to send. See VALID_CLOSE_STATUS in ABNF.
- **reason** (*bytes*) – The reason to close in UTF-8.
- **timeout** (*int or float*) – Timeout until receive a close frame. If None, it will wait forever until receive a close frame.

connect(*url*, ***options*)

Connect to url. url is websocket url scheme. ie. ws://host:port/resource You can customize using 'options'. If you set "header" list object, you can set your own custom header.

```
>>> ws = WebSocket()
>>> ws.connect("ws://echo.websocket.events",
        ...     header=["User-Agent: MyProgram",
        ...             "x-custom: header"])
```

Parameters

- **header** (*list or dict*) – Custom http header list or dict.
- **cookie** (*str*) – Cookie value.
- **origin** (*str*) – Custom origin url.
- **connection** (*str*) – Custom connection header value. Default value "Upgrade" set in _handshake.py
- **suppress_origin** (*bool*) – Suppress outputting origin header.
- **host** (*str*) – Custom host header string.
- **timeout** (*int or float*) – Socket timeout time. This value is an integer or float. If you set None for this value, it means "use default_timeout value"
- **http_proxy_host** (*str*) – HTTP proxy host name.
- **http_proxy_port** (*str or int*) – HTTP proxy port. Default is 80.
- **http_no_proxy** (*list*) – Whitelisted host names that don't use the proxy.
- **http_proxy_auth** (*tuple*) – HTTP proxy auth information. Tuple of username and password. Default is None.
- **http_proxy_timeout** (*int or float*) – HTTP proxy timeout, default is 60 sec as per python-socks.
- **redirect_limit** (*int*) – Number of redirects to follow.
- **subprotocols** (*list*) – List of available subprotocols. Default is None.
- **socket** (*socket*) – Pre-initialized stream socket.

getheaders()

Get handshake response header

getstatus()

Get handshake status

getsubprotocol()

Get subprotocol

gettimeout() → float | int | None

Get the websocket timeout (in seconds) as an int or float

Returns
timeout – returns timeout value (in seconds). This value could be either float/integer.

Return type
int or float

property headers

Get handshake response header

ping(*payload: str | bytes = ''*)

Send ping data.

Parameters
payload (*str*) – data payload to send server.

pong(*payload: str | bytes = ''*)

Send pong data.

Parameters
payload (*str*) – data payload to send server.

recv() → str | bytes

Receive string data(byte array) from the server.

Returns
data

Return type
string (byte array) value.

recv_data(*control_frame: bool = False*) → tuple

Receive data with operation code.

Parameters
control_frame (*bool*) – a boolean flag indicating whether to return control frame data, defaults to False

Returns
opcode, frame.data – tuple of operation code and string(byte array) value.

Return type
tuple

recv_data_frame(*control_frame: bool = False*) → tuple

Receive data with operation code.

If a valid ping message is received, a pong response is sent.

Parameters
control_frame (*bool*) – a boolean flag indicating whether to return control frame data, defaults to False

Returns
frame.opcode, frame – tuple of operation code and string(byte array) value.

Return type
tuple

recv_frame()

Receive data as frame from server.

Returns
self.frame_buffer.recv_frame()

Return type
ABNF frame object

send(*payload: bytes | str*, *opcode: int = 1*) → int

Send the data as string.

Parameters

- **payload** (*str*) – Payload must be utf-8 string or unicode, If the opcode is OPCODE_TEXT. Otherwise, it must be string(byte array).
- **opcode** (*int*) – Operation code (opcode) to send.

send_binary(*payload: bytes*) → int

Send a binary message (OPCODE_BINARY).

Parameters
payload (*bytes*) – payload of message to send.

send_bytes(*data: bytes | bytearray*) → int

Sends a sequence of bytes.

send_close(*status: int = 1000*, *reason: bytes = b''*)

Send close data to the server.

Parameters

- **status** (*int*) – Status code to send. See STATUS_XXX.
- **reason** (*str or bytes*) – The reason to close. This must be string or UTF-8 bytes.

send_frame(*frame*) → int

Send the data frame.

```
>>> ws = create_connection("ws://echo.websocket.events")
>>> frame = ABNF.create_frame("Hello", ABNF.OPCODE_TEXT)
>>> ws.send_frame(frame)
>>> cont_frame = ABNF.create_frame("My name is ", ABNF.OPCODE_CONT, 0)
>>> ws.send_frame(frame)
>>> cont_frame = ABNF.create_frame("Foo Bar", ABNF.OPCODE_CONT, 1)
>>> ws.send_frame(frame)
```

Parameters
frame (*ABNF frame*) – frame data created by ABNF.create_frame

send_text(*text_data: str*) → int

Sends UTF-8 encoded text.

set_mask_key(*func*)

Set function to create mask key. You can customize mask key generator. Mainly, this is for testing purpose.

Parameters

func (*func*) – callable object. the func takes 1 argument as integer. The argument means length of mask key. This func must return string(byte array), which length is argument specified.

settimeout(*timeout: float | int | None*)

Set the timeout to the websocket.

Parameters

timeout (*int or float*) – timeout time (in seconds). This value could be either float/integer.

shutdown()

close socket, immediately.

property status

Get handshake status

property subprotocol

Get subprotocol

property timeout: float | int | None

Get the websocket timeout (in seconds) as an int or float

Returns

timeout – returns timeout value (in seconds). This value could be either float/integer.

Return type

int or float

websocket._core.**create_connection**(*url: str*, *timeout=None*, *class_=<class 'websocket._core.WebSocket'>*, ***options*)

Connect to url and return websocket object.

Connect to url and return the WebSocket object. Passing optional timeout parameter will set the timeout on the socket. If no timeout is supplied, the global default timeout setting returned by getdefaulttimeout() is used. You can customize using 'options'. If you set "header" list object, you can set your own custom header.

```
>>> conn = create_connection("ws://echo.websocket.events",
     ...     header=["User-Agent: MyProgram",
     ...             "x-custom: header"])
```

Parameters

- **class** (*class*) – class to instantiate when creating the connection. It has to implement settimeout and connect. It's __init__ should be compatible with WebSocket.__init__, i.e. accept all of it's kwargs.
- **header** (*list or dict*) – custom http header list or dict.
- **cookie** (*str*) – Cookie value.
- **origin** (*str*) – custom origin url.
- **suppress_origin** (*bool*) – suppress outputting origin header.
- **host** (*str*) – custom host header string.

- **timeout** (*int or float*) – socket timeout time. This value could be either float/integer. If set to None, it uses the default_timeout value.
- **http_proxy_host** (*str*) – HTTP proxy host name.
- **http_proxy_port** (*str or int*) – HTTP proxy port. If not set, set to 80.
- **http_no_proxy** (*list*) – Whitelisted host names that don't use the proxy.
- **http_proxy_auth** (*tuple*) – HTTP proxy auth information. tuple of username and password. Default is None.
- **http_proxy_timeout** (*int or float*) – HTTP proxy timeout, default is 60 sec as per python-socks.
- **enable_multithread** (*bool*) – Enable lock for multithread.
- **redirect_limit** (*int*) – Number of redirects to follow.
- **sockopt** (*tuple*) – Values for socket.setsockopt. sockopt must be a tuple and each element is an argument of sock.setsockopt.
- **sslopt** (*dict*) – Optional dict object for ssl socket options. See FAQ for details.
- **subprotocols** (*list*) – List of available subprotocols. Default is None.
- **skip_utf8_validation** (*bool*) – Skip utf8 validation.
- **socket** (*socket*) – Pre-initialized stream socket.

CHAPTER

ELEVEN

WEBSOCKET/_EXCEPTIONS.PY

The _exceptions.py file _exceptions.py websocket - WebSocket client library for Python

Copyright 2024 engn33r

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

exception websocket._exceptions.**WebSocketAddressException**

If the websocket address info cannot be found, this exception will be raised.

exception websocket._exceptions.**WebSocketBadStatusException**(*message: str*, *status_code: int*, *status_message=None*, *resp_headers=None*, *resp_body=None*)

WebSocketBadStatusException will be raised when we get bad handshake status code.

__init__(*message: str*, *status_code: int*, *status_message=None*, *resp_headers=None*, *resp_body=None*)

exception websocket._exceptions.**WebSocketConnectionClosedException**

If remote host closed the connection or some network error happened, this exception will be raised.

exception websocket._exceptions.**WebSocketException**

WebSocket exception class.

exception websocket._exceptions.**WebSocketPayloadException**

If the WebSocket payload is invalid, this exception will be raised.

exception websocket._exceptions.**WebSocketProtocolException**

If the WebSocket protocol is invalid, this exception will be raised.

exception websocket._exceptions.**WebSocketProxyException**

WebSocketProxyException will be raised when proxy error occurred.

exception websocket._exceptions.**WebSocketTimeoutException**

WebSocketTimeoutException will be raised at socket timeout during read/write data.

CHAPTER

TWELVE

WEBSOCKET/_LOGGING.PY

The _logging.py file

websocket._logging.**enableTrace**(*traceable: bool*, *handler: ~logging.StreamHandler = <StreamHandler <stderr> (NOTSET)>*, *level: str = 'DEBUG'*) → None

Turn on/off the traceability.

Parameters

traceable (*bool*) – If set to True, traceability is enabled.

CHAPTER

THIRTEEN

WEBSOCKET/_SOCKET.PY

The _socket.py file

websocket._socket.**getdefaulttimeout**() → int | float | None

Get default timeout

Returns
_default_timeout – Return the global timeout setting (in seconds) to connect.

Return type
int or float

websocket._socket.**setdefaulttimeout**(*timeout: int | float | None*) → None

Set the global timeout setting to connect.

Parameters
timeout (*int or float*) – default socket timeout time (in seconds)

CHAPTER

FOURTEEN

WEBSOCKET/_URL.PY

The _url.py file

websocket._url.**get_proxy_info**(*hostname: str*, *is_secure: bool*, *proxy_host: str | None = None*, *proxy_port: int = 0*, *proxy_auth: tuple | None = None*, *no_proxy: list | None = None*, *proxy_type: str = 'http'*) → tuple

Try to retrieve proxy host and port from environment if not provided in options. Result is (proxy_host, proxy_port, proxy_auth). proxy_auth is tuple of username and password of proxy authentication information.

Parameters

- **hostname** (*str*) – Websocket server name.
- **is_secure** (*bool*) – Is the connection secure? (wss) looks for "https_proxy" in env instead of "http_proxy"
- **proxy_host** (*str*) – http proxy host name.
- **proxy_port** (*str or int*) – http proxy port.
- **no_proxy** (*list*) – Whitelisted host names that don't use the proxy.
- **proxy_auth** (*tuple*) – HTTP proxy auth information. Tuple of username and password. Default is None.
- **proxy_type** (*str*) – Specify the proxy protocol (http, socks4, socks4a, socks5, socks5h). Default is "http". Use socks4a or socks5h if you want to send DNS requests through the proxy.

websocket._url.**parse_url**(*url: str*) → tuple

parse url and the result is tuple of (hostname, port, resource path and the flag of secure mode)

Parameters

url (*str*) – url string.

CHAPTER

FIFTEEN

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

W

INDEX

Symbols

A

C

E

F

G

H

M

P

R

S

T

V

W