# PyZMQ Documentation

***Release 15.3.0***

**Brian E. Granger** **Min Ragan-Kelley**

July 05, 2016

Contents

**Release** 15.3.0

**Date** July 05, 2016

PyZMQ is the Python bindings for ØMQ. This documentation currently contains notes on some important aspects of developing PyZMQ and an overview of what the ØMQ API looks like in Python. For information on how to use ØMQ in general, see the many examples in the excellent ØMQ Guide, all of which have a version in Python.

PyZMQ works with Python 3 ( 3.2), and Python 2 ( 2.6), with no transformations or 2to3, as well as PyPy (at least 2.0 beta), thanks to a new CFFI backend.

Please don't hesitate to report pyzmq-specific issues to our tracker on GitHub. General questions about ØMQ are better sent to the ØMQ mailing list or IRC Channel.

CHAPTER 1

# Supported LibZMQ

PyZMQ aims to support all stable ( 2.1.4, 3.2.2, 4.0.1 ) and active development ( 4.2.0 ) versions of libzmq. Building the same pyzmq against various versions of libzmq is supported, but only the functionality of the linked libzmq will be available.

**Note:** libzmq 3.0-3.1 are not, and will never be supported. There never was a stable release of either.

Binary distributions (wheels on PyPI or GitHub) of PyZMQ ship with the stable version of libzmq at the time of release, built with default configuration, and include CURVE support provided by tweetnacl. For pyzmq-15.3.0, this is 4.1.5.

CHAPTER 2

# Using PyZMQ

## 2.1 The PyZMQ API

**Release** 15.3.0

**Date** July 05, 2016

### 2.1.1 zmq

Python bindings for 0MQ.

#### Basic Classes

##### Context

**class** zmq.**Context**(*io_threads=1*, ***kwargs*)
Create a zmq Context

A zmq Context creates sockets via its `ctx.socket` method.

**closed**
boolean - whether the context has been terminated. If True, you can no longer use this Context.

**destroy**(*linger=None*)
ctx.destroy(linger=None)

Close all sockets associated with this context, and then terminate the context. If linger is specified, the LINGER sockopt of the sockets will be set prior to closing.

> **Warning:** destroy involves calling `zmq_close()`, which is **NOT** threadsafe. If there are active sockets in other threads, this must not be called.

**get**(*option*)
ctx.get(option)

Get the value of a context option.

See the 0MQ API documentation for zmq_ctx_get for details on specific options.

New in version libzmq-3.2.

New in version 13.0.

**Parameters** **option** (*int*) – The option to get. Available values will depend on your version of libzmq. Examples include:

```
zmq.IO_THREADS, zmq.MAX_SOCKETS
```

**Returns** **optval** – The value of the option as an integer.

**Return type** int

**getsockopt**(*opt*)
get default socket options for new sockets created by this Context

New in version 13.0.

classmethod **instance**(*io_threads=1*)
Returns a global Context instance.

Most single-threaded applications have a single, global Context. Use this method instead of passing around Context instances throughout your code.

A common pattern for classes that depend on Contexts is to use a default argument to enable programs with multiple Contexts but not require the argument for simpler applications:

**class MyClass(object):**

**def __init__(self, context=None):** self.context = context or Context.instance()

**set**(*option*, *optval*)
ctx.set(option, optval)

Set a context option.

See the 0MQ API documentation for zmq_ctx_set for details on specific options.

New in version libzmq-3.2.

New in version 13.0.

**Parameters**

- **option** (*int*) – The option to set. Available values will depend on your version of libzmq. Examples include:

```
zmq.IO_THREADS, zmq.MAX_SOCKETS
```

- **optval** (*int*) – The value of the option to set.

**setsockopt**(*opt*, *value*)
set default socket options for new sockets created by this Context

New in version 13.0.

classmethod **shadow**(*address*)
Shadow an existing libzmq context

address is the integer address of the libzmq context or an FFI pointer to it.

New in version 14.1.

classmethod **shadow_pyczmq**(*ctx*)
Shadow an existing pyczmq context

ctx is the FFI *zctx_t* * pointer

New in version 14.1.

**socket**(*socket_type*, ***kwargs*)
Create a Socket associated with this Context.

**Parameters**

- **socket_type** (*int*) – The socket type, which can be any of the 0MQ socket types: REQ, REP, PUB, SUB, PAIR, DEALER, ROUTER, PULL, PUSH, etc.
- **kwargs** – will be passed to the __init__ method of the socket class.

**term**()
ctx.term()

Close or terminate the context.

This can be called to close the context by hand. If this is not called, the context will automatically be closed when it is garbage collected.

**underlying**
The address of the underlying libzmq context

### Socket

**class** zmq.**Socket**(**a*, ***kw*)
The ZMQ socket object

To create a Socket, first create a Context:

```
ctx = zmq.Context.instance()
```

then call `ctx.socket(socket_type)`:

```
s = ctx.socket(zmq.ROUTER)
```

**closed**
boolean - whether the socket has been closed. If True, you can no longer use this Socket.

**bind**(*addr*)
s.bind(addr)

Bind the socket to an address.

This causes the socket to listen on a network port. Sockets on the other side of this connection will use `Socket.connect(addr)` to connect to this socket.

**Parameters** **addr** (*str*) – The address string. This has the form 'protocol://interface:port', for example 'tcp://127.0.0.1:5555'. Protocols supported include tcp, udp, pgm, epgm, inproc and ipc. If the address is unicode, it is encoded to utf-8 first.

**bind_to_random_port**(*addr*, *min_port=49152*, *max_port=65536*, *max_tries=100*)
bind this socket to a random port in a range

If the port range is unspecified, the system will choose the port.

**Parameters**

- **addr** (*str*) – The address string without the port to pass to `Socket.bind()`.
- **min_port** (*int, optional*) – The minimum port in the range of ports to try (inclusive).
- **max_port** (*int, optional*) – The maximum port in the range of ports to try (exclusive).

- **max_tries** (*int, optional*) – The maximum number of bind attempts to make.

**Returns** **port** – The port the socket was bound to.

**Return type** int

**Raises** *ZMQBindError* – if *max_tries* reached before successful bind

**close**(*linger=None*)
s.close(linger=None)

Close the socket.

If linger is specified, LINGER sockopt will be set prior to closing.

This can be called to close the socket by hand. If this is not called, the socket will automatically be closed when it is garbage collected.

**connect**(*addr*)
s.connect(addr)

Connect to a remote 0MQ socket.

**Parameters** **addr** (*str*) – The address string. This has the form 'protocol://interface:port', for example 'tcp://127.0.0.1:5555'. Protocols supported are tcp, upd, pgm, inproc and ipc. If the address is unicode, it is encoded to utf-8 first.

**disable_monitor**()
Shutdown the PAIR socket (created using get_monitor_socket) that is serving socket events.

New in version 14.4.

**disconnect**(*addr*)
s.disconnect(addr)

Disconnect from a remote 0MQ socket (undoes a call to connect).

New in version libzmq-3.2.

New in version 13.0.

**Parameters** **addr** (*str*) – The address string. This has the form 'protocol://interface:port', for example 'tcp://127.0.0.1:5555'. Protocols supported are tcp, upd, pgm, inproc and ipc. If the address is unicode, it is encoded to utf-8 first.

**get**(*option*)
s.get(option)

Get the value of a socket option.

See the 0MQ API documentation for details on specific options.

**Parameters** **option** (*int*) – The option to get. Available values will depend on your version of libzmq. Examples include:

```
zmq.IDENTITY, HWM, LINGER, FD, EVENTS
```

**Returns** **optval** – The value of the option as a bytestring or int.

**Return type** int or bytes

**get_hwm**()
get the High Water Mark

On libzmq 3, this gets SNDHWM if available, otherwise RCVHWM

**get_monitor_socket**(*events=None*, *addr=None*)
Return a connected PAIR socket ready to receive the event notifications.

New in version libzmq-4.0.

New in version 14.0.

**Parameters**

- **events** (*bitfield (int) [default: ZMQ_EVENTS_ALL]*) – The bitmask defining which events are wanted.
- **addr** (*string [default: None]*) – The optional endpoint for the monitoring sockets.

**Returns** **socket** – The socket is already connected and ready to receive messages.

**Return type** (PAIR)

**get_string**(*option*, *encoding='utf-8'*)
get the value of a socket option

See the 0MQ documentation for details on specific options.

**Parameters** **option** (*int*) – The option to retrieve.

**Returns** **optval** – The value of the option as a unicode string.

**Return type** unicode string (unicode on py2, str on py3)

**getsockopt**()
s.get(option)

Get the value of a socket option.

See the 0MQ API documentation for details on specific options.

**Parameters** **option** (*int*) – The option to get. Available values will depend on your version of libzmq. Examples include:

```
zmq.IDENTITY, HWM, LINGER, FD, EVENTS
```

**Returns** **optval** – The value of the option as a bytestring or int.

**Return type** int or bytes

**getsockopt_string**(*option*, *encoding='utf-8'*)
get the value of a socket option

See the 0MQ documentation for details on specific options.

**Parameters** **option** (*int*) – The option to retrieve.

**Returns** **optval** – The value of the option as a unicode string.

**Return type** unicode string (unicode on py2, str on py3)

**hwm**
get the High Water Mark

On libzmq 3, this gets SNDHWM if available, otherwise RCVHWM

**monitor**(*addr*, *flags*)
s.monitor(addr, flags)

Start publishing socket events on inproc. See libzmq docs for zmq_monitor for details.

While this function is available from libzmq 3.2, pyzmq cannot parse monitor messages from libzmq prior to 4.0.

**Parameters**

- **addr** (*str*) – The inproc url used for monitoring. Passing None as the addr will cause an existing socket monitor to be deregistered.
- **events** (*int [default: zmq.EVENT_ALL]*) – The zmq event bitmask for which events will be sent to the monitor.

**poll**(*timeout=None*, *flags=1*)

poll the socket for events

The default is to poll forever for incoming events. Timeout is in milliseconds, if specified.

**Parameters**

- **timeout** (*int [default: None]*) – The timeout (in milliseconds) to wait for an event. If unspecified (or specified None), will wait forever for an event.
- **flags** (*bitfield (int) [default: POLLIN]*) – The event flags to poll for (any combination of POLLIN|POLLOUT). The default is to check for incoming events (POLLIN).

**Returns** **events** – The events that are ready and waiting. Will be 0 if no events were ready by the time timeout was reached.

**Return type** bitfield (int)

**recv**(*flags=0*, *copy=True*, *track=False*)

s.recv(flags=0, copy=True, track=False)

Receive a message.

**Parameters**

- **flags** (*int*) – Any supported flag: NOBLOCK. If NOBLOCK is set, this method will raise a ZMQError with EAGAIN if a message is not ready. If NOBLOCK is not set, then this method will block until a message arrives.
- **copy** (*bool*) – Should the message be received in a copying or non-copying manner? If False a Frame object is returned, if True a string copy of message is returned.
- **track** (*bool*) – Should the message be tracked for notification that ZMQ has finished with it? (ignored if copy=True)

**Returns** **msg** – The received message frame. If *copy* is False, then it will be a Frame, otherwise it will be bytes.

**Return type** bytes, Frame

**Raises** *ZMQError* – for any of the reasons zmq_msg_recv might fail.

**recv_json**(*flags=0*, ***kwargs*)

receive a Python object as a message using json to serialize

Keyword arguments are passed on to json.loads

**Parameters** **flags** (*int*) – Any valid recv flag.

**Returns** **obj** – The Python object that arrives as a message.

**Return type** Python object

**recv_multipart** (*flags=0*, *copy=True*, *track=False*)
receive a multipart message as a list of bytes or Frame objects

**Parameters**

- **flags** (*int, optional*) – Any supported flag: NOBLOCK. If NOBLOCK is set, this method will raise a ZMQError with EAGAIN if a message is not ready. If NOBLOCK is not set, then this method will block until a message arrives.
- **copy** (*bool, optional*) – Should the message frame(s) be received in a copying or non-copying manner? If False a Frame object is returned for each part, if True a copy of the bytes is made for each frame.
- **track** (*bool, optional*) – Should the message frame(s) be tracked for notification that ZMQ has finished with it? (ignored if copy=True)

**Returns** **msg_parts** – A list of frames in the multipart message; either Frames or bytes, depending on *copy*.

**Return type** list

**recv_pyobj** (*flags=0*)
receive a Python object as a message using pickle to serialize

**Parameters** **flags** (*int*) – Any valid recv flag.

**Returns** **obj** – The Python object that arrives as a message.

**Return type** Python object

**recv_string** (*flags=0*, *encoding='utf-8'*)
receive a unicode string, as sent by send_string

**Parameters**

- **flags** (*int*) – Any valid recv flag.
- **encoding** (*str [default: 'utf-8']*) – The encoding to be used

**Returns** **s** – The Python unicode string that arrives as encoded bytes.

**Return type** unicode string (unicode on py2, str on py3)

**send** (*data*, *flags=0*, *copy=True*, *track=False*)
s.send(data, flags=0, copy=True, track=False)

Send a message on this socket.

This queues the message to be sent by the IO thread at a later time.

**Parameters**

- **data** (*object, str, Frame*) – The content of the message.
- **flags** (*int*) – Any supported flag: NOBLOCK, SNDMORE.
- **copy** (*bool*) – Should the message be sent in a copying or non-copying manner.
- **track** (*bool*) – Should the message be tracked for notification that ZMQ has finished with it? (ignored if copy=True)

**Returns**

- **None** (if *copy* or not track) – None if message was sent, raises an exception otherwise.
- **MessageTracker** (*if track and not copy*) – a MessageTracker object, whose *pending* property will be True until the send is completed.

**Raises**

- `TypeError` – If a unicode object is passed
- `ValueError` – If *track=True*, but an untracked Frame is passed.
- *`ZMQError`* – If the send does not succeed for any reason.

**send_json**(*obj*, *flags=0*, ***kwargs*)
send a Python object as a message using json to serialize

Keyword arguments are passed on to json.dumps

**Parameters**

- **obj** (*Python object*) – The Python object to send
- **flags** (*int*) – Any valid send flag

**send_multipart**(*msg_parts*, *flags=0*, *copy=True*, *track=False*)
send a sequence of buffers as a multipart message

The zmq.SNDMORE flag is added to all msg parts before the last.

**Parameters**

- **msg_parts** (*iterable*) – A sequence of objects to send as a multipart message. Each element can be any sendable object (Frame, bytes, buffer-providers)
- **flags** (*int, optional*) – SNDMORE is handled automatically for frames before the last.
- **copy** (*bool, optional*) – Should the frame(s) be sent in a copying or non-copying manner.
- **track** (*bool, optional*) – Should the frame(s) be tracked for notification that ZMQ has finished with it (ignored if copy=True).

**Returns**

- **None** (*if copy or not track*)
- **MessageTracker** (*if track and not copy*) – a MessageTracker object, whose *pending* property will be True until the last send is completed.

**send_pyobj**(*obj*, *flags=0*, *protocol=3*)
send a Python object as a message using pickle to serialize

**Parameters**

- **obj** (*Python object*) – The Python object to send.
- **flags** (*int*) – Any valid send flag.
- **protocol** (*int*) – The pickle protocol number to use. The default is pickle.DEFAULT_PROTOCOL where defined, and pickle.HIGHEST_PROTOCOL elsewhere.

**send_string**(*u*, *flags=0*, *copy=True*, *encoding='utf-8'*)
send a Python unicode string as a message with an encoding

0MQ communicates with raw bytes, so you must encode/decode text (unicode on py2, str on py3) around 0MQ.

**Parameters**

- **u** (*Python unicode string (unicode on py2, str on py3)*) – The unicode string to send.
- **flags** (*int, optional*) – Any valid send flag.
- **encoding** (*str [default: 'utf-8']*) – The encoding to be used

**set**(*option*, *optval*)
s.set(option, optval)

Set socket options.

See the 0MQ API documentation for details on specific options.

**Parameters**

- **option** (*int*) – The option to set. Available values will depend on your version of libzmq. Examples include:

```
zmq.SUBSCRIBE, UNSUBSCRIBE, IDENTITY, HWM, LINGER, FD
```

- **optval** (*int or bytes*) – The value of the option to set.

**Notes**

**Warning:** All options other than zmq.SUBSCRIBE, zmq.UNSUBSCRIBE and zmq.LINGER only take effect for subsequent socket bind/connects.

**set_hwm**(*value*)
set the High Water Mark

On libzmq 3, this sets both SNDHWM and RCVHWM

**Warning:** New values only take effect for subsequent socket bind/connects.

**set_string**(*option*, *optval*, *encoding='utf-8'*)
set socket options with a unicode object

This is simply a wrapper for setsockopt to protect from encoding ambiguity.

See the 0MQ documentation for details on specific options.

**Parameters**

- **option** (*int*) – The name of the option to set. Can be any of: SUBSCRIBE, UNSUBSCRIBE, IDENTITY
- **optval** (*unicode string (unicode on py2, str on py3)*) – The value of the option to set.
- **encoding** (*str*) – The encoding to be used, default is utf8

**setsockopt**()
s.set(option, optval)

Set socket options.

See the 0MQ API documentation for details on specific options.

**Parameters**

- **option** (*int*) – The option to set. Available values will depend on your version of libzmq. Examples include:

```
zmq.SUBSCRIBE, UNSUBSCRIBE, IDENTITY, HWM, LINGER, FD
```

- **optval** (*int or bytes*) – The value of the option to set.

#### Notes

> **Warning:** All options other than zmq.SUBSCRIBE, zmq.UNSUBSCRIBE and zmq.LINGER only take effect for subsequent socket bind/connects.

**setsockopt_string**(*option*, *optval*, *encoding='utf-8'*)
set socket options with a unicode object

This is simply a wrapper for setsockopt to protect from encoding ambiguity.

See the 0MQ documentation for details on specific options.

**Parameters**

- **option** (*int*) – The name of the option to set. Can be any of: SUBSCRIBE, UNSUBSCRIBE, IDENTITY
- **optval** (*unicode string (unicode on py2, str on py3)*) – The value of the option to set.
- **encoding** (*str*) – The encoding to be used, default is utf8

**classmethod shadow**(*address*)
Shadow an existing libzmq socket

address is the integer address of the libzmq socket or an FFI pointer to it.

New in version 14.1.

**subscribe**(*topic*)
Subscribe to a topic

Only for SUB sockets.

New in version 15.3.

**unbind**(*addr*)
s.unbind(addr)

Unbind from an address (undoes a call to bind).

New in version libzmq-3.2.

New in version 13.0.

**Parameters** **addr** (*str*) – The address string. This has the form 'protocol://interface:port', for example 'tcp://127.0.0.1:5555'. Protocols supported are tcp, upd, pgm, inproc and ipc. If the address is unicode, it is encoded to utf-8 first.

**underlying**
The address of the underlying libzmq socket

**unsubscribe**(*topic*)
Unsubscribe from a topic

Only for SUB sockets.

New in version 15.3.

### Frame

**class** zmq.**Frame**

**__getattr__**(*key*)
get zmq options by attribute

**__setattr__**(*key*, *value*)
set zmq options by attribute

**buffer**
A read-only buffer view of the message contents.

**bytes**
The message content as a Python bytes object.

The first time this property is accessed, a copy of the message contents is made. From then on that same copy of the message is returned.

**get**(*option*)
Get a Frame option or property.

See the 0MQ API documentation for zmq_msg_get and zmq_msg_gets for details on specific options.

New in version libzmq-3.2.

New in version 13.0.

Changed in version 14.3: add support for zmq_msg_gets (requires libzmq-4.1)

**set**(*option*, *value*)
Set a Frame option.

See the 0MQ API documentation for zmq_msg_set for details on specific options.

New in version libzmq-3.2.

New in version 13.0.

### MessageTracker

**class** zmq.**MessageTracker**(**towatch*)
A class for tracking if 0MQ is done using one or more messages.

When you send a 0MQ message, it is not sent immediately. The 0MQ IO thread sends the message at some later time. Often you want to know when 0MQ has actually sent the message though. This is complicated by the fact that a single 0MQ message can be sent multiple times using different sockets. This class allows you to track all of the 0MQ usages of a message.

**Parameters** **towatch** (*Event, MessageTracker, Message instances.*) – This objects to track. This class can track the low-level Events used by the Message class, other MessageTrackers or actual Messages.

**done**
Is 0MQ completely done with the message(s) being tracked?

**wait**(*timeout=-1*)
mt.wait(timeout=-1)

Wait for 0MQ to be done with the message or until *timeout*.

**Parameters** **timeout** (*float [default: -1, wait forever]*) – Maximum time in (s) to wait before raising NotDone.

**Returns** if done before *timeout*

**Return type** None

**Raises** *NotDone* – if *timeout* reached before I am done.

## Polling

### Poller

**class** zmq.**Poller**
A stateful poll interface that mirrors Python's built-in poll.

**modify**(*socket*, *flags=3*)
Modify the flags for an already registered 0MQ socket or native fd.

**poll**(*timeout=None*)
Poll the registered 0MQ or native fds for I/O.

**Parameters** **timeout** (*float, int*) – The timeout in milliseconds. If None, no *timeout* (infinite). This is in milliseconds to be compatible with `select.poll()`.

**Returns** **events** – The list of events that are ready to be processed. This is a list of tuples of the form `(socket, event)`, where the 0MQ Socket or integer fd is the first element, and the poll event mask (POLLIN, POLLOUT) is the second. It is common to call `events = dict(poller.poll())`, which turns the list of tuples into a mapping of `socket : event`.

**Return type** list of tuples

**register**(*socket*, *flags=POLLIN|POLLOUT*)
p.register(socket, flags=POLLIN|POLLOUT)

Register a 0MQ socket or native fd for I/O monitoring.

register(s,0) is equivalent to unregister(s).

**Parameters**

- **socket** (*zmq.Socket or native socket*) – A zmq.Socket or any Python object having a `fileno()` method that returns a valid file descriptor.
- **flags** (*int*) – The events to watch for. Can be POLLIN, POLLOUT or POLLIN|POLLOUT. If *flags=0*, socket will be unregistered.

**unregister**(*socket*)
Remove a 0MQ socket or native fd for I/O monitoring.

**Parameters** **socket** (Socket) – The socket instance to stop polling.

zmq.**select**(*rlist*, *wlist*, *xlist*, *timeout=None) -> (rlist*, *wlist*, *xlist*)
Return the result of poll as a lists of sockets ready for r/w/exception.

This has the same interface as Python's built-in `select.select()` function.

**Parameters**

- **timeout** (*float, int, optional*) – The timeout in seconds. If None, no timeout (infinite). This is in seconds to be compatible with `select.select()`.
- **rlist** (*list of sockets/FDs*) – sockets/FDs to be polled for read events
- **wlist** (*list of sockets/FDs*) – sockets/FDs to be polled for write events
- **xlist** (*list of sockets/FDs*) – sockets/FDs to be polled for error events

**Returns (rlist, wlist, xlist)** – Lists correspond to sockets available for read/write/error events respectively.

**Return type** tuple of lists of sockets (length 3)

## Exceptions

### ZMQError

**class** zmq.**ZMQError**(*errno=None*, *msg=None*)
Wrap an errno style error.

**Parameters**

- **errno** (*int*) – The ZMQ errno or None. If None, then `zmq_errno()` is called and used.
- **msg** (*string*) – Description of the error or None.

**with_traceback**()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

### ZMQVersionError

**class** zmq.**ZMQVersionError**(*min_version*, *msg='Feature'*)
Raised when a feature is not provided by the linked version of libzmq.

New in version 14.2.

**with_traceback**()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

### Again

**class** zmq.**Again**(*errno=None*, *msg=None*)
Wrapper for zmq.EAGAIN

New in version 13.0.

#### ContextTerminated

**class** zmq.**ContextTerminated**(*errno=None*, *msg=None*)
Wrapper for zmq.ETERM

New in version 13.0.

#### NotDone

**class** zmq.**NotDone**
Raised when timeout is reached while waiting for 0MQ to finish with a Message

**See also:**

**MessageTracker.wait** object for tracking when ZeroMQ is done

#### ZMQBindError

**class** zmq.**ZMQBindError**
An error for Socket.bind_to_random_port().

**See also:**

Socket.bind_to_random_port

### Functions

zmq.**zmq_version**()
return the version of libzmq as a string

zmq.**pyzmq_version**()
return the version of pyzmq as a string

zmq.**zmq_version_info**()
Return the version of ZeroMQ itself as a 3-tuple of ints.

zmq.**pyzmq_version_info**()
return the pyzmq version as a tuple of at least three numbers

If pyzmq is a development version, *inf* will be appended after the third integer.

zmq.**has**()
Check for zmq capability by name (e.g. 'ipc', 'curve')

New in version libzmq-4.1.

New in version 14.1.

zmq.**device**(*device_type*, *frontend*, *backend*)
Start a zeromq device.

Deprecated since version libzmq-3.2: Use zmq.proxy

**Parameters**

- **device_type** (*(QUEUE, FORWARDER, STREAMER)*) – The type of device to start.
- **frontend** (Socket) – The Socket instance for the incoming traffic.
- **backend** (Socket) – The Socket instance for the outbound traffic.

zmq.**proxy**(*frontend*, *backend*, *capture*)
Start a zeromq proxy (replacement for device).

New in version libzmq-3.2.

New in version 13.0.

**Parameters**

- **frontend** (Socket) – The Socket instance for the incoming traffic.
- **backend** (Socket) – The Socket instance for the outbound traffic.
- **capture** (*Socket (optional)*) – The Socket instance for capturing traffic.

zmq.**curve_keypair**()
generate a Z85 keypair for use with zmq.CURVE security

Requires libzmq ( 4.0) to have been built with CURVE support.

New in version libzmq-4.0.

New in version 14.0.

**Returns (public, secret)** – The public and private keypair as 40 byte z85-encoded bytestrings.

**Return type** two bytestrings

zmq.**get_includes**()
Return a list of directories to include for linking against pyzmq with cython.

### 2.1.2 devices

#### Functions

zmq.**device**(*device_type*, *frontend*, *backend*)
Start a zeromq device.

Deprecated since version libzmq-3.2: Use zmq.proxy

**Parameters**

- **device_type** (*(QUEUE, FORWARDER, STREAMER)*) – The type of device to start.
- **frontend** (Socket) – The Socket instance for the incoming traffic.
- **backend** (Socket) – The Socket instance for the outbound traffic.

zmq.**proxy**(*frontend*, *backend*, *capture*)
Start a zeromq proxy (replacement for device).

New in version libzmq-3.2.

New in version 13.0.

**Parameters**

- **frontend** (Socket) – The Socket instance for the incoming traffic.
- **backend** (Socket) – The Socket instance for the outbound traffic.
- **capture** (*Socket (optional)*) – The Socket instance for capturing traffic.

### Module: zmq.devices

0MQ Device classes for running in background threads or processes.

### Base Devices

#### Device

**class** zmq.devices.**Device**(*device_type=3*, *in_type=None*, *out_type=None*)
A 0MQ Device to be run in the background.

You do not pass Socket instances to this, but rather Socket types:

```
Device(device_type, in_socket_type, out_socket_type)
```

For instance:

```
dev = Device(zmq.QUEUE, zmq.DEALER, zmq.ROUTER)
```

Similar to zmq.device, but socket types instead of sockets themselves are passed, and the sockets are created in the work thread, to avoid issues with thread safety. As a result, additional bind_{in|out} and connect_{in|out} methods and setsockopt_{in|out} allow users to specify connections for the sockets.

**Parameters**

- **device_type** (*int*) – The 0MQ Device type
- **{in|out}_type** (*int*) – zmq socket types, to be passed later to context.socket(). e.g. zmq.PUB, zmq.SUB, zmq.REQ. If out_type is < 0, then in_socket is used for both in_socket and out_socket.

**bind_{in_out}(iface)**
passthrough for {in|out}_socket.bind(iface), to be called in the thread

**connect_{in_out}(iface)**
passthrough for {in|out}_socket.connect(iface), to be called in the thread

**setsockopt_{in_out}(opt,value)**
passthrough for {in|out}_socket.setsockopt(opt, value), to be called in the thread

**daemon**
*int*

sets whether the thread should be run as a daemon Default is true, because if it is false, the thread will not exit unless it is killed

**context_factory**
*callable (class attribute)*

Function for creating the Context. This will be Context.instance in ThreadDevices, and Context in ProcessDevices. The only reason it is not instance() in ProcessDevices is that there may be a stale Context instance already initialized, and the forked environment should *never* try to use it.

**bind_in**(*addr*)
Enqueue ZMQ address for binding on in_socket.

See zmq.Socket.bind for details.

**bind_out**(*addr*)
Enqueue ZMQ address for binding on out_socket.

See zmq.Socket.bind for details.

**connect_in**(*addr*)
Enqueue ZMQ address for connecting on in_socket.

See zmq.Socket.connect for details.

**connect_out**(*addr*)
Enqueue ZMQ address for connecting on out_socket.

See zmq.Socket.connect for details.

**join**(*timeout=None*)
wait for me to finish, like Thread.join.

Reimplemented appropriately by subclasses.

**setsockopt_in**(*opt*, *value*)
Enqueue setsockopt(opt, value) for in_socket

See zmq.Socket.setsockopt for details.

**setsockopt_out**(*opt*, *value*)
Enqueue setsockopt(opt, value) for out_socket

See zmq.Socket.setsockopt for details.

**start**()
Start the device. Override me in subclass for other launchers.

#### ThreadDevice

**class** zmq.devices.**ThreadDevice**(*device_type=3*, *in_type=None*, *out_type=None*)
A Device that will be run in a background Thread.

See Device for details.

#### ProcessDevice

**class** zmq.devices.**ProcessDevice**(*device_type=3*, *in_type=None*, *out_type=None*)
A Device that will be run in a background Process.

See Device for details.

**context_factory**(*io_threads=1*, ***kwargs*)
Callable that returns a context. Typically either Context.instance or Context, depending on whether the device should share the global instance or not.

alias of Context

### Proxy Devices

#### Proxy

**class** zmq.devices.**Proxy**(*in_type*, *out_type*, *mon_type=1*)
Threadsafe Proxy object.

See zmq.devices.Device for most of the spec. This subclass adds a <method>_mon version of each <method>_{in|out} method, for configuring the monitor socket.

A Proxy is a 3-socket ZMQ Device that functions just like a QUEUE, except each message is also sent out on the monitor socket.

A PUB socket is the most logical choice for the mon_socket, but it is not required.

**bind_mon**(*addr*)
Enqueue ZMQ address for binding on mon_socket.

See zmq.Socket.bind for details.

**connect_mon**(*addr*)
Enqueue ZMQ address for connecting on mon_socket.

See zmq.Socket.bind for details.

**setsockopt_mon**(*opt*, *value*)
Enqueue setsockopt(opt, value) for mon_socket

See zmq.Socket.setsockopt for details.

#### ThreadProxy

**class** zmq.devices.**ThreadProxy**(*in_type*, *out_type*, *mon_type=1*)
Proxy in a Thread. See Proxy for more.

#### ProcessProxy

**class** zmq.devices.**ProcessProxy**(*in_type*, *out_type*, *mon_type=1*)
Proxy in a Process. See Proxy for more.

### MonitoredQueue Devices

zmq.devices.**monitored_queue**()

**monitored_queue(in_socket, out_socket, mon_socket,** in_prefix=b'in', out_prefix=b'out')

Start a monitored queue device.

A monitored queue is very similar to the zmq.proxy device (monitored queue came first).

Differences from zmq.proxy:

- monitored_queue supports both in and out being ROUTER sockets (via swapping IDENTITY prefixes).
- monitor messages are prefixed, making in and out messages distinguishable.

**Parameters**

- **in_socket** (Socket) – One of the sockets to the Queue. Its messages will be prefixed with 'in'.
- **out_socket** (Socket) – One of the sockets to the Queue. Its messages will be prefixed with 'out'. The only difference between in/out socket is this prefix.
- **mon_socket** (Socket) – This socket sends out every message received by each of the others with an in/out prefix specifying which one it was.
- **in_prefix** (*str*) – Prefix added to broadcast messages from in_socket.
- **out_prefix** (*str*) – Prefix added to broadcast messages from out_socket.

#### MonitoredQueue

**class** zmq.devices.**MonitoredQueue**(*in_type*, *out_type*, *mon_type=1*, *in_prefix=b'in'*, *out_prefix=b'out'*)

Class for running monitored_queue in the background.

See zmq.devices.Device for most of the spec. MonitoredQueue differs from Proxy, only in that it adds a `prefix` to messages sent on the monitor socket, with a different prefix for each direction.

MQ also supports ROUTER on both sides, which zmq.proxy does not.

If a message arrives on *in_sock*, it will be prefixed with *in_prefix* on the monitor socket. If it arrives on out_sock, it will be prefixed with *out_prefix*.

A PUB socket is the most logical choice for the mon_socket, but it is not required.

#### ThreadMonitoredQueue

**class** zmq.devices.**ThreadMonitoredQueue**(*in_type*, *out_type*, *mon_type=1*, *in_prefix=b'in'*, *out_prefix=b'out'*)

Run zmq.monitored_queue in a background thread.

See MonitoredQueue and Proxy for details.

#### ProcessMonitoredQueue

**class** zmq.devices.**ProcessMonitoredQueue**(*in_type*, *out_type*, *mon_type=1*, *in_prefix=b'in'*, *out_prefix=b'out'*)

Run zmq.monitored_queue in a background thread.

See MonitoredQueue and Proxy for details.

### 2.1.3 decorators

#### Module: zmq.decorators

Decorators for running functions with context/sockets.

New in version 15.3.

Like using Contexts and Sockets as context managers, but with decorator syntax. Context and sockets are closed at the end of the function.

For example:

```
from zmq.decorators import context, socket

@context()
@socket(zmq.PUSH)
def work(ctx, push):
    ...
```

#### Decorators

zmq.decorators.**context**(**args*, ***kwargs*)
Decorator for adding a Context to a function.

Usage:

```
@context()
def foo(ctx):
    ...
```

New in version 15.3.

**Parameters** **name** (*str*) – the keyword argument passed to decorated function

zmq.decorators.**socket**(**args*, ***kwargs*)
Decorator for adding a socket to a function.

Usage:

```
@socket(zmq.PUSH)
def foo(push):
    ...
```

New in version 15.3.

**Parameters**

- **name** (*str*) – the keyword argument passed to decorated function
- **context_name** (*str*) – the keyword only argument to identify context object

### 2.1.4 green

#### Module: green

zmq.green - gevent compatibility with zeromq.

#### Usage

Instead of importing zmq directly, do so in the following manner:

import zmq.green as zmq

Any calls that would have blocked the current thread will now only block the current green thread.

This compatibility is accomplished by ensuring the nonblocking flag is set before any blocking operation and the ØMQ file descriptor is polled internally to trigger needed events.

### 2.1.5 eventloop.ioloop

#### Module: eventloop.ioloop

tornado IOLoop API with zmq compatibility

If you have tornado 3.0, this is a subclass of tornado's IOLoop, otherwise we ship a minimal subset of tornado in zmq.eventloop.minitornado.

The minimal shipped version of tornado's IOLoop does not include support for concurrent futures - this will only be available if you have tornado 3.0.

## Classes

### DelayedCallback

**class** zmq.eventloop.ioloop.**DelayedCallback**(*callback*, *callback_time*, *io_loop=None*)
Schedules the given callback to be called once.

The callback is called once, after callback_time milliseconds.

*start* must be called after the DelayedCallback is created.

The timeout is calculated from when *start* is called.

**is_running**()
Return True if this .*PeriodicCallback* has been started.

New in version 4.1.

**start**()
Starts the timer.

**stop**()
Stops the timer.

### ZMQIOLoop

**class** zmq.eventloop.ioloop.**ZMQIOLoop**
ZMQ subclass of tornado's IOLoop

**ERROR = 24**

**NONE = 0**

**READ = 1**

**WRITE = 4**

**add_callback**(*callback*, **args*, ***kwargs*)

**add_callback_from_signal**(*callback*, **args*, ***kwargs*)

**add_future**(*future*, *callback*)
Schedules a callback on the IOLoop when the given .*Future* is finished.

The callback is invoked with one argument, the .*Future*.

**add_handler**(*fd*, *handler*, *events*)

**add_timeout**(*deadline*, *callback*, **args*, ***kwargs*)
Runs the callback at the time deadline from the I/O loop.

Returns an opaque handle that may be passed to *remove_timeout* to cancel.

deadline may be a number denoting a time (on the same scale as *IOLoop.time*, normally *time.time*), or a *datetime.timedelta* object for a deadline relative to the current time. Since Tornado 4.0, *call_later* is a more convenient alternative for the relative case since it does not require a timedelta object.

Note that it is not safe to call *add_timeout* from other threads. Instead, you must use *add_callback* to transfer control to the *IOLoop*'s thread, and then call *add_timeout* from there.

Subclasses of IOLoop must implement either *add_timeout* or *call_at*; the default implementations of each will call the other. *call_at* is usually easier to implement, but subclasses that wish to maintain compatibility with Tornado versions prior to 4.0 must use *add_timeout* instead.

Changed in version 4.0: Now passes through `*args` and `**kwargs` to the callback.

**call_at**(*deadline*, *callback*, **args*, ***kwargs*)

**call_later**(*delay*, *callback*, **args*, ***kwargs*)
Runs the `callback` after `delay` seconds have passed.

Returns an opaque handle that may be passed to *remove_timeout* to cancel. Note that unlike the *asyncio* method of the same name, the returned object does not have a `cancel()` method.

See *add_timeout* for comments on thread-safety and subclassing.

New in version 4.0.

**clear_current**()

**clear_instance**()
Clear the global *IOLoop* instance.

New in version 4.0.

**close**(*all_fds=False*)

**close_fd**(*fd*)
Utility method to close an `fd`.

If `fd` is a file-like object, we close it directly; otherwise we use *os.close*.

This method is provided for use by *IOLoop* subclasses (in implementations of `IOLoop.close(all_fds=True)` and should not generally be used by application code.

New in version 4.0.

**configurable_base**()

**configurable_default**()

**configure**(*impl*, ***kwargs*)
Sets the class to use when the base class is instantiated.

Keyword arguments will be saved and added to the arguments passed to the constructor. This can be used to set global defaults for some parameters.

**configured_class**()
Returns the currently configured class.

**static current**(**args*, ***kwargs*)
Returns the current thread's IOLoop.

**handle_callback_exception**(*callback*)
This method is called whenever a callback run by the *IOLoop* throws an exception.

By default simply logs the exception as an error. Subclasses may override this method to customize reporting of exceptions.

The exception itself is not passed explicitly, but is available in *sys.exc_info*.

**initialize**(*impl=None*, ***kwargs*)

**initialized**()
Returns true if the singleton instance has been created.

**install**()
Installs this *IOLoop* object as the singleton instance.

This is normally not necessary as *instance()* will create an *IOLoop* on demand, but you may want to call *install* to use a custom subclass of *IOLoop*.

**static instance**(**args*, ***kwargs*)
Returns a global *IOLoop* instance.

Most applications have a single, global *IOLoop* running on the main thread. Use this method to get this instance from another thread. To get the current thread's *IOLoop*, use *current()*.

**log_stack**(*signal*, *frame*)
Signal handler to log the stack trace of the current thread.

For use with *set_blocking_signal_threshold*.

**make_current**()
Makes this the *IOLoop* for the current thread.

An *IOLoop* automatically becomes current for its thread when it is started, but it is sometimes useful to call *make_current* explicitly before starting the *IOLoop*, so that code run at startup time can find the right instance.

Changed in version 4.1: An *IOLoop* created while there is no current *IOLoop* will automatically become current.

**remove_handler**(*fd*)

**remove_timeout**(*timeout*)

**run_sync**(*func*, *timeout=None*)
Starts the *IOLoop*, runs the given function, and stops the loop.

The function must return either a yieldable object or `None`. If the function returns a yieldable object, the *IOLoop* will run until the yieldable is resolved (and *run_sync()* will return the yieldable's result). If it raises an exception, the *IOLoop* will stop and the exception will be re-raised to the caller.

The keyword-only argument `timeout` may be used to set a maximum duration for the function. If the timeout expires, a *TimeoutError* is raised.

This method is useful in conjunction with *tornado.gen.coroutine* to allow asynchronous calls in a `main()` function:

```
@gen.coroutine
def main():
    # do stuff...

if __name__ == '__main__':
    IOLoop.current().run_sync(main)
```

Changed in version 4.3: Returning a non-`None`, non-yieldable value is now an error.

**set_blocking_log_threshold**(*seconds*)
Logs a stack trace if the *IOLoop* is blocked for more than `s` seconds.

Equivalent to `set_blocking_signal_threshold(seconds, self.log_stack)`

**set_blocking_signal_threshold**(*seconds*, *action*)

**spawn_callback**(*callback*, **args*, ***kwargs*)
Calls the given callback on the next IOLoop iteration.

Unlike all other callback-related methods on IOLoop, `spawn_callback` does not associate the callback with its caller's `stack_context`, so it is suitable for fire-and-forget callbacks that should not interfere with the caller.

New in version 4.0.

**split_fd**(*fd*)
Returns an (fd, obj) pair from an `fd` parameter.

We accept both raw file descriptors and file-like objects as input to *add_handler* and related methods. When a file-like object is passed, we must retain the object itself so we can close it correctly when the *IOLoop* shuts down, but the poller interfaces favor file descriptors (they will accept file-like objects and call `fileno()` for you, but they always return the descriptor itself).

This method is provided for use by *IOLoop* subclasses and should not generally be used by application code.

New in version 4.0.

**start**()

**stop**()

**time**()

**update_handler**(*fd*, *events*)

### ZMQPoller

**class** zmq.eventloop.ioloop.**ZMQPoller**
A poller that can be used in the tornado IOLoop.

This simply wraps a regular zmq.Poller, scaling the timeout by 1000, so that it is in seconds rather than milliseconds.

**close**()

**modify**(*fd*, *events*)

**poll**(*timeout*)
poll in seconds rather than milliseconds.

Event masks will be IOLoop.READ/WRITE/ERROR

**register**(*fd*, *events*)

**unregister**(*fd*)

## Function

zmq.eventloop.ioloop.**install**()
set the tornado IOLoop instance with the pyzmq IOLoop.

After calling this function, tornado's IOLoop.instance() and pyzmq's IOLoop.instance() will return the same object.

An assertion error will be raised if tornado's IOLoop has been initialized prior to calling this function.

### 2.1.6 eventloop.future

#### Module: `eventloop.future`

Future-returning APIs for coroutines.

New in version 15.0.

As of pyzmq 15, there is a new Socket subclass that returns Futures for recv methods, which can be found at `zmq.eventloop.future.Socket`. You can create these sockets by instantiating a `Context` from the same module. These sockets let you easily use zmq with tornado's coroutines.

**See also:**

`tornado:tornado.gen`

```
from tornado import gen
from zmq.eventloop.future import Context

ctx = Context()

@gen.coroutine
def recv_and_process():
    sock = ctx.socket(zmq.PULL)
    sock.bind(url)
    msg = yield sock.recv_multipart() # waits for msg to be ready
    reply = yield async_process(msg)
    yield sock.send_multipart(reply)
```

#### Classes

##### Context

Context class that creates Future-returning sockets. See *zmq.Context* for more info.

**class** `zmq.eventloop.future.`**Context**(**args*, ***kwargs*)

##### Socket

Socket subclass that returns `Future` s from blocking methods, for use in coroutines and async applications.

**See also:**

*zmq.Socket* for the inherited API.

**class** `zmq.eventloop.future.`**Socket**(*context*, *socket_type*, *io_loop=None*)

**recv**(*flags=0*, *copy=True*, *track=False*)
Receive a single zmq frame.

Returns a Future, whose result will be the received frame.

Recommend using recv_multipart instead.

**recv_multipart**(*flags=0*, *copy=True*, *track=False*)
Receive a complete multipart zmq message.

Returns a Future whose result will be a multipart message.

**send**(*msg*, *flags=0*, *copy=True*, *track=False*)
Send a single zmq frame.

Returns a Future that resolves when sending is complete.

Recommend using send_multipart instead.

**send_multipart**(*msg*, *flags=0*, *copy=True*, *track=False*)
Send a complete multipart zmq message.

Returns a Future that resolves when sending is complete.

**poll**(*timeout=None*, *flags=1*)
poll the socket for events

returns a Future for the poll results.

#### Poller

Poller subclass that returns `Future` s from poll, for use in coroutines and async applications.

**See also:**

*`zmq.Poller`* for the inherited API.

**class** `zmq.eventloop.future.`**Poller**

**poll**(*timeout=-1*)
Return a Future for a poll event

### 2.1.7 asyncio

#### Module: zmq.asyncio

AsyncIO support for zmq

Requires asyncio and Python 3.

New in version 15.0.

As of 15.0, pyzmq now supports `asyncio`, via *`zmq.asyncio`*. When imported from this module, blocking methods such as `zmq.asyncio.Socket.recv_multipart()`, `zmq.asyncio.Socket.poll()`, and `zmq.asyncio.Poller.poll()` return `Future` s.

It also provides a *`zmq.asyncio.ZMQEventLoop`*.

```
import asyncio
import zmq
import zmq.asyncio

ctx = zmq.asyncio.Context()
loop = zmq.asyncio.ZMQEventLoop()
asyncio.set_event_loop(loop)

@asyncio.coroutine
def recv_and_process():
    sock = ctx.socket(zmq.PULL)
    sock.bind(url)
    msg = yield from sock.recv_multipart() # waits for msg to be ready
```

```
    reply = yield from async_process(msg)
    yield from sock.send_multipart(reply)

loop.run_until_complete(recv_and_process())
```

## Classes

### ZMQEventLoop

An asyncio event loop using zmq_poll for zmq socket support.

**class** zmq.asyncio.**ZMQEventLoop**(*selector=None*)
AsyncIO eventloop using zmq_poll

### Context

Context class that creates Future-returning sockets. See *zmq.Context* for more info.

**class** zmq.asyncio.**Context**(*io_threads=1*, ***kwargs*)
Context for creating asyncio-compatible Sockets

### Socket

Socket subclass that returns asyncio.Future s from blocking methods, for use in coroutines and async applications.

**See also:**

*zmq.Socket* for the inherited API.

**class** zmq.asyncio.**Socket**(*context*, *socket_type*, *io_loop=None*)
Socket returning asyncio Futures for send/recv/poll methods.

**recv**(*flags=0*, *copy=True*, *track=False*)
Receive a single zmq frame.

Returns a Future, whose result will be the received frame.

Recommend using recv_multipart instead.

**recv_multipart**(*flags=0*, *copy=True*, *track=False*)
Receive a complete multipart zmq message.

Returns a Future whose result will be a multipart message.

**send**(*msg*, *flags=0*, *copy=True*, *track=False*)
Send a single zmq frame.

Returns a Future that resolves when sending is complete.

Recommend using send_multipart instead.

**send_multipart**(*msg*, *flags=0*, *copy=True*, *track=False*)
Send a complete multipart zmq message.

Returns a Future that resolves when sending is complete.

**poll**(*timeout=None*, *flags=1*)
poll the socket for events

returns a Future for the poll results.

#### Poller

Poller subclass that returns `asyncio.Future` s from poll, for use in coroutines and async applications.

**See also:**

*zmq.Poller* for the inherited API.

**class** zmq.asyncio.**Poller**
Poller returning asyncio.Future for poll results.

**poll**(*timeout=-1*)
Return a Future for a poll event

### 2.1.8 eventloop.zmqstream

#### Module: eventloop.zmqstream

A utility class to send to and recv from a non-blocking socket.

#### ZMQStream

**class** zmq.eventloop.zmqstream.**ZMQStream**(*socket*, *io_loop=None*)
A utility class to register callbacks when a zmq socket sends and receives

For use with zmq.eventloop.ioloop

There are three main methods

Methods:

- **on_recv(callback, copy=True):** register a callback to be run every time the socket has something to receive
- **on_send(callback):** register a callback to be run every time you call send
- **send(self, msg, flags=0, copy=False, callback=None):** perform a send that will trigger the callback if callback is passed, on_send is also called.

  There are also send_multipart(), send_json(), send_pyobj()

Three other methods for deactivating the callbacks:

- **stop_on_recv():** turn off the recv callback
- **stop_on_send():** turn off the send callback

which simply call `on_<evt>(None)`.

The entire socket interface, excluding direct recv methods, is also provided, primarily through direct-linking the methods. e.g.

```
>>> stream.bind is stream.socket.bind
True
```

**close**(*linger=None*)
Close this stream.

**closed**()

**flush**(*flag=3*, *limit=None*)
Flush pending messages.

This method safely handles all pending incoming and/or outgoing messages, bypassing the inner loop, passing them to the registered callbacks.

A limit can be specified, to prevent blocking under high load.

**flush will return the first time ANY of these conditions are met:**

- No more events matching the flag are pending.
- the total number of events handled reaches the limit.

Note that if `flag|POLLIN != 0`, recv events will be flushed even if no callback is registered, unlike normal IOLoop operation. This allows flush to be used to remove *and ignore* incoming messages.

**Parameters**

- **flag** (*int, default=POLLIN|POLLOUT*) – 0MQ poll flags. If flag|POLLIN, recv events will be flushed. If flag|POLLOUT, send events will be flushed. Both flags can be set at once, which is the default.
- **limit** (*None or int, optional*) – The maximum number of messages to send or receive. Both send and recv count against this limit.

**Returns** **int**

**Return type** count of events handled (both send and recv)

**io_loop = None**

**on_err**(*callback*)
DEPRECATED, does nothing

**on_recv**(*callback*, *copy=True*)
Register a callback for when a message is ready to recv.

There can be only one callback registered at a time, so each call to *on_recv* replaces previously registered callbacks.

on_recv(None) disables recv event polling.

Use on_recv_stream(callback) instead, to register a callback that will receive both this ZMQStream and the message, instead of just the message.

**Parameters**

- **callback** (*callable*) – callback must take exactly one argument, which will be a list, as returned by socket.recv_multipart() if callback is None, recv callbacks are disabled.
- **copy** (*bool*) – copy is passed directly to recv, so if copy is False, callback will receive Message objects. If copy is True, then callback will receive bytes/str objects.
- **Returns** (*None*) –

**on_recv_stream**(*callback*, *copy=True*)
Same as on_recv, but callback will get this stream as first argument

callback must take exactly two arguments, as it will be called as:

```
callback(stream, msg)
```

Useful when a single callback should be used with multiple streams.

**on_send**(*callback*)
Register a callback to be called on each send

There will be two arguments:

```
callback(msg, status)
```

- *msg* will be the list of sendable objects that was just sent
- *status* will be the return result of socket.send_multipart(msg) - MessageTracker or None.

Non-copying sends return a MessageTracker object whose *done* attribute will be True when the send is complete. This allows users to track when an object is safe to write to again.

The second argument will always be None if copy=True on the send.

Use on_send_stream(callback) to register a callback that will be passed this ZMQStream as the first argument, in addition to the other two.

on_send(None) disables recv event polling.

**Parameters** **callback** (*callable*) – callback must take exactly two arguments, which will be the message being sent (always a list), and the return result of socket.send_multipart(msg) - MessageTracker or None.

if callback is None, send callbacks are disabled.

**on_send_stream**(*callback*)
Same as on_send, but callback will get this stream as first argument

Callback will be passed three arguments:

```
callback(stream, msg, status)
```

Useful when a single callback should be used with multiple streams.

**poller = None**

**receiving**()
Returns True if we are currently receiving from the stream.

**send**(*msg*, *flags=0*, *copy=True*, *track=False*, *callback=None*)
Send a message, optionally also register a new callback for sends. See zmq.socket.send for details.

**send_json**(*obj*, *flags=0*, *callback=None*)
Send json-serialized version of an object. See zmq.socket.send_json for details.

**send_multipart**(*msg*, *flags=0*, *copy=True*, *track=False*, *callback=None*)
Send a multipart message, optionally also register a new callback for sends. See zmq.socket.send_multipart for details.

**send_pyobj**(*obj*, *flags=0*, *protocol=-1*, *callback=None*)
Send a Python object as a message using pickle to serialize.

See zmq.socket.send_json for details.

**send_string**(*u*, *flags=0*, *encoding='utf-8'*, *callback=None*)
Send a unicode message with an encoding. See zmq.socket.send_unicode for details.

**send_unicode**(*u*, *flags=0*, *encoding='utf-8'*, *callback=None*)
Send a unicode message with an encoding. See zmq.socket.send_unicode for details.

**sending**()
Returns True if we are currently sending to the stream.

**set_close_callback**(*callback*)
Call the given callback when the stream is closed.

**socket = None**

**stop_on_err**()
DEPRECATED, does nothing

**stop_on_recv**()
Disable callback and automatic receiving.

**stop_on_send**()
Disable callback on sending.

### 2.1.9 auth

#### Module: auth

Utilities for ZAP authentication.

To run authentication in a background thread, see *zmq.auth.thread*. For integration with the tornado eventloop, see zmq.auth.ioloop. For integration with the asyncio event loop, see zmq.auth.asyncio.

New in version 14.1.

#### Authenticator

**class** zmq.auth.**Authenticator**(*context=None*, *encoding='utf-8'*, *log=None*)
Implementation of ZAP authentication for zmq connections.

Note:

- libzmq provides four levels of security: default NULL (which the Authenticator does not see), and authenticated NULL, PLAIN, CURVE, and GSSAPI, which the Authenticator can see.
- until you add policies, all incoming NULL connections are allowed. (classic ZeroMQ behavior), and all PLAIN and CURVE connections are denied.
- GSSAPI requires no configuration.

**allow**(**addresses*)
Allow (whitelist) IP address(es).

Connections from addresses not in the whitelist will be rejected.

- For NULL, all clients from this address will be accepted.
- For real auth setups, they will be allowed to continue with authentication.

whitelist is mutually exclusive with blacklist.

**configure_curve**(*domain='**', *location=None*)
Configure CURVE authentication for a given domain.

CURVE authentication uses a directory that holds all public client certificates, i.e. their public keys.

To cover all domains, use "*".

You can add and remove certificates in that directory at any time.

To allow all client keys without checking, specify CURVE_ALLOW_ANY for the location.

**configure_gssapi**(*domain='*'*, *location=None*)
Configure GSSAPI authentication

Currently this is a no-op because there is nothing to configure with GSSAPI.

**configure_plain**(*domain='*'*, *passwords=None*)
Configure PLAIN authentication for a given domain.

PLAIN authentication uses a plain-text password file. To cover all domains, use "*". You can modify the password file at any time; it is reloaded automatically.

**deny**(**addresses*)
Deny (blacklist) IP address(es).

Addresses not in the blacklist will be allowed to continue with authentication.

Blacklist is mutually exclusive with whitelist.

**handle_zap_message**(*msg*)
Perform ZAP authentication

**start**()
Create and bind the ZAP socket

**stop**()
Close the ZAP socket

#### Functions

zmq.auth.**create_certificates**(*key_dir*, *name*, *metadata=None*)
Create zmq certificates.

Returns the file paths to the public and secret certificate files.

zmq.auth.**load_certificate**(*filename*)
Load public and secret key from a zmq certificate.

Returns (public_key, secret_key)

If the certificate file only contains the public key, secret_key will be None.

If there is no public key found in the file, ValueError will be raised.

zmq.auth.**load_certificates**(*directory='.'*)
Load public keys from all certificates in a directory

### 2.1.10 auth.thread

#### Module: auth.thread

ZAP Authenticator in a Python Thread.

New in version 14.1.

### Classes

**ThreadAuthenticator**

**class** zmq.auth.thread.**ThreadAuthenticator**(*context=None*, *encoding='utf-8'*, *log=None*)
Run ZAP authentication in a background thread

**allow**(**addresses*)
Allow (whitelist) IP address(es).

Connections from addresses not in the whitelist will be rejected.

•For NULL, all clients from this address will be accepted.

•For real auth setups, they will be allowed to continue with authentication.

whitelist is mutually exclusive with blacklist.

**configure_curve**(*domain='*'*, *location=''*)
Configure CURVE authentication for a given domain.

CURVE authentication uses a directory that holds all public client certificates, i.e. their public keys.

To cover all domains, use "*".

You can add and remove certificates in that directory at any time.

To allow all client keys without checking, specify CURVE_ALLOW_ANY for the location.

**configure_plain**(*domain='*'*, *passwords=None*)
Configure PLAIN authentication for a given domain.

PLAIN authentication uses a plain-text password file. To cover all domains, use "*". You can modify the password file at any time; it is reloaded automatically.

**deny**(**addresses*)
Deny (blacklist) IP address(es).

Addresses not in the blacklist will be allowed to continue with authentication.

Blacklist is mutually exclusive with whitelist.

**is_alive**()
Is the ZAP thread currently running?

**start**()
Start the authentication thread

**stop**()
Stop the authentication thread

## 2.1.11 auth.ioloop

### Module: auth.ioloop

ZAP Authenticator integrated with the tornado IOLoop.

New in version 14.1.

#### IOLoopAuthenticator

**class** zmq.auth.ioloop.**IOLoopAuthenticator**(*context=None*, *encoding='utf-8'*, *log=None*, *io_loop=None*)

ZAP authentication for use in the tornado IOLoop

**allow**(**addresses*)
Allow (whitelist) IP address(es).

Connections from addresses not in the whitelist will be rejected.

- For NULL, all clients from this address will be accepted.
- For real auth setups, they will be allowed to continue with authentication.

whitelist is mutually exclusive with blacklist.

**configure_curve**(*domain='**', *location=None*)
Configure CURVE authentication for a given domain.

CURVE authentication uses a directory that holds all public client certificates, i.e. their public keys.

To cover all domains, use "*".

You can add and remove certificates in that directory at any time.

To allow all client keys without checking, specify CURVE_ALLOW_ANY for the location.

**configure_gssapi**(*domain='**', *location=None*)
Configure GSSAPI authentication

Currently this is a no-op because there is nothing to configure with GSSAPI.

**configure_plain**(*domain='**', *passwords=None*)
Configure PLAIN authentication for a given domain.

PLAIN authentication uses a plain-text password file. To cover all domains, use "*". You can modify the password file at any time; it is reloaded automatically.

**deny**(**addresses*)
Deny (blacklist) IP address(es).

Addresses not in the blacklist will be allowed to continue with authentication.

Blacklist is mutually exclusive with whitelist.

**handle_zap_message**(*msg*)
Perform ZAP authentication

**start**()
Start ZAP authentication

**stop**()
Stop ZAP authentication

### 2.1.12 log.handlers

#### Module: log.handlers

pyzmq logging handlers.

This mainly defines the PUBHandler object for publishing logging messages over a zmq.PUB socket.

The PUBHandler can be used with the regular logging module, as in:

```
>>> import logging
>>> handler = PUBHandler('tcp://127.0.0.1:12345')
>>> handler.root_topic = 'foo'
>>> logger = logging.getLogger('foobar')
>>> logger.setLevel(logging.DEBUG)
>>> logger.addHandler(handler)
```

After this point, all messages logged by `logger` will be published on the PUB socket.

Code adapted from StarCluster:

http://github.com/jtriley/StarCluster/blob/master/starcluster/logger.py

## Classes

### PUBHandler

**class** zmq.log.handlers.**PUBHandler**(*interface_or_socket*, *context=None*)

A basic logging handler that emits log messages through a PUB socket.

Takes a PUB socket already bound to interfaces or an interface to bind to.

Example:

```
sock = context.socket(zmq.PUB)
sock.bind('inproc://log')
handler = PUBHandler(sock)
```

Or:

```
handler = PUBHandler('inproc://loc')
```

These are equivalent.

Log messages handled by this handler are broadcast with ZMQ topics `this.root_topic` comes first, followed by the log level (DEBUG,INFO,etc.), followed by any additional subtopics specified in the message by: log.debug("subtopic.subsub::the real message")

**acquire**()

Acquire the I/O thread lock.

**addFilter**(*filter*)

Add the specified filter to this handler.

**close**()

Tidy up any resources used by the handler.

This version removes the handler from an internal map of handlers, _handlers, which is used for handler lookup by name. Subclasses should ensure that this gets called from overridden close() methods.

**createLock**()

Acquire a thread lock for serializing access to the underlying I/O.

**emit**(*record*)

Emit a log message on my socket.

**filter**(*record*)

Determine if a record is loggable by consulting all the filters.

The default is to allow the record to be logged; any filter can veto this and the record is then dropped. Returns a zero value if a record is to be dropped, else non-zero.

**flush**()
Ensure all logging output has been flushed.

This version does nothing and is intended to be implemented by subclasses.

**format**(*record*)
Format a record.

**formatters = {40: <logging.Formatter object at 0x7f414cf08240>, 10: <logging.Formatter object at 0x7f414d072c50>, 2**

**get_name**()

**handle**(*record*)
Conditionally emit the specified logging record.

Emission depends on filters which may have been added to the handler. Wrap the actual emission of the record with acquisition/release of the I/O thread lock. Returns whether the filter passed the record for emission.

**handleError**(*record*)
Handle errors which occur during an emit() call.

This method should be called from handlers when an exception is encountered during an emit() call. If raiseExceptions is false, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors. You could, however, replace this with a custom handler if you wish. The record which was being processed is passed in to this method.

**name**

**release**()
Release the I/O thread lock.

**removeFilter**(*filter*)
Remove the specified filter from this handler.

**root_topic = ''**

**setFormatter**(*fmt*)
Set the formatter for this handler.

**setLevel**(*level*)
Set the logging level of this handler. level must be an int or a str.

**set_name**(*name*)

**socket = None**

### TopicLogger

**class** zmq.log.handlers.**TopicLogger**(*name*, *level=0*)
A simple wrapper that takes an additional argument to log methods.

All the regular methods exist, but instead of one msg argument, two arguments: topic, msg are passed.

That is:

```
logger.debug('msg')
```

Would become:

```
logger.debug('topic.sub', 'msg')
```

**addFilter**(*filter*)
Add the specified filter to this handler.

**addHandler**(*hdlr*)
Add the specified handler to this logger.

**callHandlers**(*record*)
Pass a record to all relevant handlers.

Loop through all handlers for this logger and its parents in the logger hierarchy. If no handler was found, output a one-off error message to sys.stderr. Stop searching up the hierarchy whenever a logger with the "propagate" attribute set to zero is found - that will be the last logger whose handlers are called.

**critical**(*level*, *topic*, *msg*, **args*, ***kwargs*)

**debug**(*level*, *topic*, *msg*, **args*, ***kwargs*)

**error**(*level*, *topic*, *msg*, **args*, ***kwargs*)

**exception**(*msg*, **args*, ***kwargs*)
Convenience method for logging an ERROR with exception information.

**fatal**(*level*, *topic*, *msg*, **args*, ***kwargs*)

**filter**(*record*)
Determine if a record is loggable by consulting all the filters.

The default is to allow the record to be logged; any filter can veto this and the record is then dropped. Returns a zero value if a record is to be dropped, else non-zero.

**findCaller**(*stack_info=False*)
Find the stack frame of the caller so that we can note the source file name, line number and function name.

**getChild**(*suffix*)
Get a logger which is a descendant to this one.

This is a convenience method, such that

logging.getLogger('abc').getChild('def.ghi')

is the same as

logging.getLogger('abc.def.ghi')

It's useful, for example, when the parent logger is named using __name__ rather than a literal string.

**getEffectiveLevel**()
Get the effective level for this logger.

Loop through this logger and its parents in the logger hierarchy, looking for a non-zero logging level. Return the first one found.

**handle**(*record*)
Call the handlers for the specified record.

This method is used for unpickled records received from a socket, as well as those created locally. Logger-level filtering is applied.

**hasHandlers**()
See if this logger has any handlers configured.

Loop through all handlers for this logger and its parents in the logger hierarchy. Return True if a handler was found, else False. Stop searching up the hierarchy whenever a logger with the "propagate" attribute set to zero is found - that will be the last logger which is checked for the existence of handlers.

**info** (*msg*, **args*, ***kwargs*)
Log 'msg % args' with severity 'INFO'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.info("Houston, we have a %s", "interesting problem", exc_info=1)

**isEnabledFor** (*level*)
Is this logger enabled for level 'level'?

**log** (*level*, *topic*, *msg*, **args*, ***kwargs*)
Log 'msg % args' with level and topic.

To pass exception information, use the keyword argument exc_info with a True value:

```
logger.log(level, "zmq.fun", "We have a %s",
        "mysterious problem", exc_info=1)
```

**makeRecord** (*name*, *level*, *fn*, *lno*, *msg*, *args*, *exc_info*, *func=None*, *extra=None*, *sinfo=None*)
A factory method which can be overridden in subclasses to create specialized LogRecords.

**manager = <logging.Manager object>**

**removeFilter** (*filter*)
Remove the specified filter from this handler.

**removeHandler** (*hdlr*)
Remove the specified handler from this logger.

**root = <logging.RootLogger object>**

**setLevel** (*level*)
Set the logging level of this logger. level must be an int or a str.

**warn** (*level*, *topic*, *msg*, **args*, ***kwargs*)

**warning** (*level*, *topic*, *msg*, **args*, ***kwargs*)

### 2.1.13 ssh.tunnel

#### Module: `ssh.tunnel`

Basic ssh tunnel utilities, and convenience functions for tunneling zeromq connections.

#### Functions

zmq.ssh.tunnel.**open_tunnel** (*addr*, *server*, *keyfile=None*, *password=None*, *paramiko=None*, *timeout=60*)
Open a tunneled connection from a 0MQ url.

For use inside tunnel_connection.

**Returns (url, tunnel)** – The 0MQ url that has been forwarded, and the tunnel object

**Return type** (str, object)

zmq.ssh.tunnel.**openssh_tunnel** (*lport*, *rport*, *server*, *remoteip='127.0.0.1'*, *keyfile=None*, *password=None*, *timeout=60*)
Create an ssh tunnel using command-line ssh that connects port lport on this machine to localhost:rport on server. The tunnel will automatically close when not in use, remaining open for a minimum of timeout seconds for an initial connection.

This creates a tunnel redirecting *localhost:lport* to *remoteip:rport*, as seen from *server*.

keyfile and password may be specified, but ssh config is checked for defaults.

**Parameters**

- **lport** (*int*) – local port for connecting to the tunnel from this machine.
- **rport** (*int*) – port on the remote machine to connect to.
- **server** (*str*) – The ssh server to connect to. The full ssh server string will be parsed. user@server:port
- **remoteip** (*str [Default: 127.0.0.1]*) – The remote ip, specifying the destination of the tunnel. Default is localhost, which means that the tunnel would redirect localhost:lport on this machine to localhost:rport on the *server*.
- **keyfile** (*str; path to public key file*) – This specifies a key to be used in ssh login, default None. Regular default ssh keys will be used without specifying this argument.
- **password** (*str;*) – Your ssh password to the ssh server. Note that if this is left None, you will be prompted for it if passwordless key based login is unavailable.
- **timeout** (*int [default: 60]*) – The time (in seconds) after which no activity will result in the tunnel closing. This prevents orphaned tunnels from running forever.

zmq.ssh.tunnel.**paramiko_tunnel**(*lport*, *rport*, *server*, *remoteip='127.0.0.1'*, *keyfile=None*, *password=None*, *timeout=60*)

launch a tunner with paramiko in a subprocess. This should only be used when shell ssh is unavailable (e.g. Windows).

This creates a tunnel redirecting *localhost:lport* to *remoteip:rport*, as seen from *server*.

If you are familiar with ssh tunnels, this creates the tunnel:

ssh server -L localhost:lport:remoteip:rport

keyfile and password may be specified, but ssh config is checked for defaults.

**Parameters**

- **lport** (*int*) – local port for connecting to the tunnel from this machine.
- **rport** (*int*) – port on the remote machine to connect to.
- **server** (*str*) – The ssh server to connect to. The full ssh server string will be parsed. user@server:port
- **remoteip** (*str [Default: 127.0.0.1]*) – The remote ip, specifying the destination of the tunnel. Default is localhost, which means that the tunnel would redirect localhost:lport on this machine to localhost:rport on the *server*.
- **keyfile** (*str; path to public key file*) – This specifies a key to be used in ssh login, default None. Regular default ssh keys will be used without specifying this argument.
- **password** (*str;*) – Your ssh password to the ssh server. Note that if this is left None, you will be prompted for it if passwordless key based login is unavailable.
- **timeout** (*int [default: 60]*) – The time (in seconds) after which no activity will result in the tunnel closing. This prevents orphaned tunnels from running forever.

zmq.ssh.tunnel.**select_random_ports**(*n*)

Selects and return n random ports that are available.

zmq.ssh.tunnel.**try_passwordless_ssh**(*server*, *keyfile*, *paramiko=None*)
Attempt to make an ssh connection without a password. This is mainly used for requiring password input only once when many tunnels may be connected to the same server.

If paramiko is None, the default for the platform is chosen.

zmq.ssh.tunnel.**tunnel_connection**(*socket*, *addr*, *server*, *keyfile=None*, *password=None*, *paramiko=None*, *timeout=60*)
Connect a socket to an address via an ssh tunnel.

This is a wrapper for socket.connect(addr), when addr is not accessible from the local machine. It simply creates an ssh tunnel using the remaining args, and calls socket.connect('tcp://localhost:lport') where lport is the randomly selected local port of the tunnel.

### 2.1.14 utils.jsonapi

#### Module: utils.jsonapi

Priority based json library imports.

Always serializes to bytes instead of unicode for zeromq compatibility on Python 2 and 3.

Use `jsonapi.loads()` and `jsonapi.dumps()` for guaranteed symmetry.

Priority: `simplejson` > `jsonlib2` > stdlib `json`

`jsonapi.loads/dumps` provide kwarg-compatibility with stdlib json.

`jsonapi.jsonmod` will be the module of the actual underlying implementation.

#### Functions

zmq.utils.jsonapi.**dumps**(*o*, ***kwargs*)
Serialize object to JSON bytes (utf-8).

See jsonapi.jsonmod.dumps for details on kwargs.

zmq.utils.jsonapi.**loads**(*s*, ***kwargs*)
Load object from JSON bytes (utf-8).

See jsonapi.jsonmod.loads for details on kwargs.

### 2.1.15 utils.monitor

#### Module: utils.monitor

Module holding utility and convenience functions for zmq event monitoring.

#### Functions

zmq.utils.monitor.**parse_monitor_message**(*msg*)
decode zmq_monitor event messages.

**Parameters** **msg** (*list(bytes)*) – zmq multipart message that has arrived on a monitor PAIR socket.

First frame is:

```
16 bit event id
32 bit event value
no padding
```

Second frame is the endpoint as a bytestring

**Returns** **event** – event description as dict with the keys *event*, *value*, and *endpoint*.

**Return type** dict

zmq.utils.monitor.**recv_monitor_message**(*socket*, *flags=0*)

Receive and decode the given raw message from the monitoring socket and return a dict.

Requires libzmq 4.0

**The returned dict will have the following entries:** event : int, the event id as described in libzmq.zmq_socket_monitor value : int, the event value associated with the event, see libzmq.zmq_socket_monitor endpoint : string, the affected endpoint

**Parameters**

- **socket** (*zmq PAIR socket*) – The PAIR socket (created by other.get_monitor_socket()) on which to recv the message
- **flags** (*bitfield (int)*) – standard zmq recv flags

**Returns** **event** – event description as dict with the keys *event*, *value*, and *endpoint*.

**Return type** dict

### 2.1.16 utils.z85

#### Module: utils.z85

Python implementation of Z85 85-bit encoding

Z85 encoding is a plaintext encoding for a bytestring interpreted as 32bit integers. Since the chunks are 32bit, a bytestring must be a multiple of 4 bytes. See ZMQ RFC 32 for details.

#### Functions

zmq.utils.z85.**decode**(*z85bytes*)

decode Z85 bytes to raw bytes, accepts ASCII string

zmq.utils.z85.**encode**(*rawbytes*)

encode raw bytes into Z85

### 2.1.17 utils.win32

#### Module: zmq.utils.win32

Win32 compatibility utilities.

#### allow_interrupt

**class** zmq.utils.win32.**allow_interrupt**(*action=None*)

Utility for fixing CTRL-C events on Windows.

On Windows, the Python interpreter intercepts CTRL-C events in order to translate them into `KeyboardInterrupt` exceptions. It (presumably) does this by setting a flag in its "console control handler" and checking it later at a convenient location in the interpreter.

However, when the Python interpreter is blocked waiting for the ZMQ poll operation to complete, it must wait for ZMQ's `select()` operation to complete before translating the CTRL-C event into the `KeyboardInterrupt` exception.

The only way to fix this seems to be to add our own "console control handler" and perform some application-defined operation that will unblock the ZMQ polling operation in order to force ZMQ to pass control back to the Python interpreter.

This context manager performs all that Windows-y stuff, providing you with a hook that is called when a CTRL-C event is intercepted. This hook allows you to unblock your ZMQ poll operation immediately, which will then result in the expected `KeyboardInterrupt` exception.

Without this context manager, your ZMQ-based application will not respond normally to CTRL-C events on Windows. If a CTRL-C event occurs while blocked on ZMQ socket polling, the translation to a `KeyboardInterrupt` exception will be delayed until the I/O completes and control returns to the Python interpreter (this may never happen if you use an infinite timeout).

A no-op implementation is provided on non-Win32 systems to avoid the application from having to conditionally use it.

Example usage:

```
def stop_my_application():
    # ...

with allow_interrupt(stop_my_application):
    # main polling loop.
```

In a typical ZMQ application, you would use the "self pipe trick" to send message to a `PAIR` socket in order to interrupt your blocking socket polling operation.

In a Tornado event loop, you can use the `IOLoop.stop` method to unblock your I/O loop.

## 2.2 Changes in PyZMQ

This is a coarse summary of changes in pyzmq versions. For a real changelog, consult the git log

### 2.2.1 15.3

- Bump bundled libzmq to 4.1.5, using tweetnacl for bundled curve support instead of libsodium
- FIX: include .pxi includes in installation for consumers of Cython API
- FIX: various fixes in new async sockets
- Introduce *zmq.decorators* API for decorating functions to create sockets or contexts

- Add *zmq.Socket.subscribe()* and *zmq.Socket.unsubscribe()* methods to sockets, so that assignment is no longer needed for subscribing. Verbs should be methods! Assignment is still supported for backward-compatibility.
- Accept text (unicode) input to z85 encoding, not just bytes
- *zmq.Context.socket()* forwards keyword arguments to the `Socket` constructor

### 2.2.2 15.2

- FIX: handle multiple events in a single register call in *zmq.asyncio*
- FIX: unicode/bytes bug in password prompt in `zmq.ssh` on Python 3
- FIX: workaround gevent monkeypatches in garbage collection thread
- update bundled minitornado from tornado-4.3.
- improved inspection by setting `binding=True` in cython compile options
- add asyncio Authenticator implementation in `zmq.auth.asyncio`
- workaround overflow bug in libzmq preventing receiving messages larger than `MAX_INT`

### 2.2.3 15.1

- FIX: Remove inadvertant tornado dependency when using *zmq.asyncio*
- FIX: 15.0 Python 3.5 wheels didn't work on Windows
- Add GSSAPI support to Authenticators
- Support new constants defined in upcoming libzmq-4.2.dev

### 2.2.4 15.0

PyZMQ 15 adds Future-returning sockets and pollers for both `asyncio` and `tornado`.

- add `asyncio` support via *zmq.asyncio*
- add `tornado` future support via *zmq.eventloop.future*
- trigger bundled libzmq if system libzmq is found to be < 3. System libzmq 2 can be forced by explicitly requesting `--zmq=/prefix/`.

### 2.2.5 14.7.0

Changes:

- Update bundled libzmq to 4.1.2.
- Following the lead of Python 3.5, interrupted system calls will be retried.

Fixes:

- Fixes for CFFI backend on Python 3 + support for PyPy 3.
- Verify types of all frames in *send_multipart()* before sending, to avoid partial messages.
- Fix build on Windows when both debug and release versions of libzmq are found.

- Windows build fixes for Python 3.5.

### 2.2.6 14.6.0

Changes:

- **improvements in *zmq.Socket.bind_to_random_port()*:**
  - use system to allocate ports by default
  - catch EACCES on Windows
- include libsodium when building bundled libzmq on Windows (includes wheels on PyPI)
- pyzmq no longer bundles external libzmq when making a bdist. You can use delocate to do this.

Bugfixes:

- add missing `ndim` on memoryviews of Frames
- allow `copy.copy()` and `copy.deepcopy()` on Sockets, Contexts

### 2.2.7 14.5.0

Changes:

- use pickle.DEFAULT_PROTOCOL by default in send_pickle
- with the release of pip-6, OS X wheels are only marked as 10.6-intel, indicating that they should be installable on any newer or single-arch Python.
- raise SSHException on failed check of host key

Bugfixes:

- fix method name in utils.wi32.allow_interrupt
- fork-related fixes in garbage collection thread
- add missing import in `zmq.__init__`, causing failure to import in some circumstances

### 2.2.8 14.4.1

Bugfixes for 14.4

- SyntaxError on Python 2.6 in zmq.ssh
- Handle possible bug in garbage collection after fork

### 2.2.9 14.4.0

New features:

- Experimental support for libzmq-4.1.0 rc (new constants, plus *zmq.has()*).
- Update bundled libzmq to 4.0.5
- Update bundled libsodium to 1.0.0
- Fixes for SSH dialogs when using `zmq.ssh` to create tunnels

- More build/link/load fixes on OS X and Solaris
- Get Frame metadata via dict access (libzmq 4)
- Contexts and Sockets are context managers (term/close on `__exit__`)
- Add *zmq.utils.win32.allow_interrupt* context manager for catching SIGINT on Windows

Bugs fixed:

- Bundled libzmq should not trigger recompilation after install on PyPy

### 2.2.10 14.3.1

**Note:** pyzmq-14.3.1 is the last version to include bdists for Python 3.3

Minor bugfixes to pyzmq 14.3:

- Fixes to building bundled libzmq on OS X < 10.9
- Fixes to import-failure warnings on Python 3.4
- Fixes to tests
- Pull upstream fixes to zmq.ssh for ssh multiplexing

### 2.2.11 14.3.0

- PyZMQ no longer calls *Socket.close()* or *Context.term()* during process cleanup. Changes to garbage collection in Python 3.4 make this impossible to do sensibly.
- `ZMQStream.close()` closes its socket immediately, rather than scheduling a timeout.
- Raise the original ImportError when importing zmq fails. Should be more informative than *no module cffi...*.

**Warning:** Users of Python 3.4 should not use pyzmq < 14.3, due to changes in garbage collection.

### 2.2.12 14.2.0

#### New Stuff

- Raise new ZMQVersionError when a requested method is not supported by the linked libzmq. For backward compatibility, this subclasses NotImplementedError.

#### Bugs Fixed

- Memory leak introduced in pyzmq-14.0 in zero copy.
- OverflowError on 32 bit systems in zero copy.

### 2.2.13 14.1.0

#### Security

The headline features for 14.1 are adding better support for libzmq's security features.

- When libzmq is bundled as a Python extension (e.g. wheels, eggs), libsodium is also bundled (excluding Windows), ensuring that libzmq security is available to users who install from wheels
- New *zmq.auth*, implementing zeromq's ZAP authentication, modeled on czmq zauth. For more information, see the examples.

#### Other New Stuff

- Add PYZMQ_BACKEND for enabling use of backends outside the pyzmq codebase.
- Add *underlying* property and *shadow()* method to Context and Socket, for handing off sockets and contexts. between pyzmq and other bindings (mainly pyczmq).
- Add TOS, ROUTER_HANDOVER, and IPC_FILTER constants from libzmq-4.1-dev.
- Add Context option support in the CFFI backend.
- Various small unicode and build fixes, as always.
- *send_json()* and *recv_json()* pass any extra kwargs to `json.dumps/loads`.

#### Deprecations

- `Socket.socket_type` is deprecated, in favor of `Socket.type`, which has been available since 2.1.

### 2.2.14 14.0.1

Bugfix release

- Update bundled libzmq to current (4.0.3).
- Fix bug in *Context.destroy()* with no open sockets.
- Threadsafety fixes in the garbage collector.
- Python 3 fixes in `zmq.ssh`.

### 2.2.15 14.0.0

- Update bundled libzmq to current (4.0.1).
- Backends are now implemented in `zmq.backend` instead of `zmq.core`. This has no effect on public APIs.
- Various build improvements for Cython and CFFI backends (PyPy compiles at build time).
- Various GIL-related performance improvements - the GIL is no longer touched from a zmq IO thread.
- Adding a constant should now be a bit easier - only zmq/sugar/constant_names should need updating, all other constant-related files should be automatically updated by `setup.py constants`.
- add support for latest libzmq-4.0.1 (includes ZMQ_CURVE security and socket event monitoring).

#### New stuff

- *Socket.monitor()*
- *Socket.get_monitor_socket()*
- *zmq.curve_keypair()*
- *zmq.utils.monitor*
- *zmq.utils.z85*

### 2.2.16 13.1.0

The main new feature is improved tornado 3 compatibility. PyZMQ ships a 'minitornado' submodule, which contains a small subset of tornado 3.0.1, in order to get the IOLoop base class. zmq.eventloop.ioloop.IOLoop is now a simple subclass, and if the system tornado is 3.0, then the zmq IOLoop is a proper registered subclass of the tornado one itself, and minitornado is entirely unused.

### 2.2.17 13.0.2

Bugfix release!

A few things were broken in 13.0.0, so this is a quick bugfix release.

- **FIXED** EAGAIN was unconditionally turned into KeyboardInterrupt
- **FIXED** we used totally deprecated ctypes_configure to generate constants in CFFI backend
- **FIXED** memory leak in CFFI backend for PyPy
- **FIXED** typo prevented IPC_PATH_MAX_LEN from ever being defined
- **FIXED** various build fixes - linking with librt, Cython compatibility, etc.

### 2.2.18 13.0.1

defunct bugfix. We do not speak of this...

### 2.2.19 13.0.0

PyZMQ now officially targets libzmq-3 (3.2.2), 0MQ 2.1.4 is still supported for the indefinite future, but 3.x is recommended. PyZMQ has detached from libzmq versioning, and will just follow its own regular versioning scheme from now on. PyZMQ bdists will include whatever is the latest stable libzmq release (3.2.2 for pyzmq-13.0).

---

**Note:** set/get methods are exposed via get/setattr on all Context, Socket, and Frame classes. This means that subclasses of these classes that require extra attributes **must declare these attributes at the class level**.

---

#### Experiments Removed

- The Threadsafe ZMQStream experiment in 2.2.0.1 was deemed inappropriate and not useful, and has been removed.
- The `zmq.web` experiment has been removed, to be developed as a standalone project.

### New Stuff

- Support for PyPy via CFFI backend (requires py, ctypes-configure, and cffi).
- Add support for new APIs in libzmq-3
  - *Socket.disconnect()*
  - *Socket.unbind()*
  - *Context.set()*
  - *Context.get()*
  - *Frame.set()*
  - *Frame.get()*
  - *zmq.proxy()*
  - *zmq.devices.Proxy*
  - Exceptions for common zmq errnos: *zmq.Again*, *zmq.ContextTerminated* (subclass `ZMQError`, so fully backward-compatible).
- Setting and getting *Socket.hwm* sets or gets *both* SNDHWM/RCVHWM for libzmq-3.
- Implementation splits core Cython bindings from pure-Python subclasses with sugar methods (send/recv_multipart). This should facilitate non-Cython backends and PyPy support [spoiler: it did!].

### Bugs Fixed

- Unicode fixes in log and monitored queue
- MinGW, ppc, cross-compilation, and HP-UX build fixes
- *zmq.green* should be complete - devices and tornado eventloop both work in gevent contexts.

## 2.2.20 2.2.0.1

This is a tech-preview release, to try out some new features. It is expected to be short-lived, as there are likely to be issues to iron out, particularly with the new pip-install support.

### Experimental New Stuff

These features are marked 'experimental', which means that their APIs are not set in stone, and may be removed or changed in incompatible ways in later releases.

#### Threadsafe ZMQStream

With the IOLoop inherited from tornado, there is exactly one method that is threadsafe: `IOLoop.add_callback()`. With this release, we are trying an experimental option to pass all IOLoop calls via this method, so that ZMQStreams can be used from one thread while the IOLoop runs in another. To try out a threadsafe stream:

```
stream = ZMQStream(socket, threadsafe=True)
```

#### pip install pyzmq

PyZMQ should now be pip installable, even on systems without libzmq. In these cases, when pyzmq fails to find an appropriate libzmq to link against, it will try to build libzmq as a Python extension. This work is derived from pyzmq_static.

To this end, PyZMQ source distributions include the sources for libzmq (2.2.0) and libuuid (2.21), both used under the LGPL.

#### zmq.green

The excellent gevent_zeromq socket subclass which provides gevent compatibility has been merged as *zmq.green*.

**See also:**

*PyZMQ and gevent*

### Bugs Fixed

- TIMEO sockopts are properly included for libzmq-2.2.0
- avoid garbage collection of sockets after fork (would cause `assert (mailbox.cpp:79)`).

## 2.2.21 2.2.0

Some effort has gone into refining the pyzmq API in this release to make it a model for other language bindings. This is principally made in a few renames of objects and methods, all of which leave the old name for backwards compatibility.

---

**Note:** As of this release, all code outside `zmq.core` is BSD licensed (where possible), to allow more permissive use of less-critical code and utilities.

---

### Name Changes

- The `Message` class has been renamed to *Frame*, to better match other zmq bindings. The old Message name remains for backwards-compatibility. Wherever pyzmq docs say "Message", they should refer to a complete zmq atom of communication (one or more Frames, connected by ZMQ_SNDMORE). Please report any remaining instances of Message==MessagePart with an Issue (or better yet a Pull Request).
- All `foo_unicode` methods are now called `foo_string` (`_unicode` remains for backwards compatibility). This is not only for cross-language consistency, but it makes more sense in Python 3, where native strings are unicode, and the `_unicode` suffix was wedded too much to Python 2.

### Other Changes and Removals

- `prefix` removed as an unused keyword argument from *send_multipart()*.
- ZMQStream *send()* default has been changed to *copy=True*, so it matches Socket *send()*.
- ZMQStream *on_err()* is deprecated, because it never did anything.

- Python 2.5 compatibility has been dropped, and some code has been cleaned up to reflect no-longer-needed hacks.
- Some Cython files in `zmq.core` have been split, to reduce the amount of Cython-compiled code. Much of the body of these files were pure Python, and thus did not benefit from the increased compile time. This change also aims to ease maintaining feature parity in other projects, such as pyzmq-ctypes.

### New Stuff

- *`Context`* objects can now set default options when they create a socket. These are set and accessed as attributes to the context. Socket options that do not apply to a socket (e.g. SUBSCRIBE on non-SUB sockets) will simply be ignored.
- *`on_recv_stream()`* has been added, which adds the stream itself as a second argument to the callback, making it easier to use a single callback on multiple streams.
- A `more` boolean attribute has been added to the *`Frame`* (née Message) class, so that frames can be identified as terminal without extra queires of `rcvmore`.

### Experimental New Stuff

These features are marked 'experimental', which means that their APIs are not set in stone, and may be removed or changed in incompatible ways in later releases.

- `zmq.web` added for load-balancing requests in a tornado webapp with zeromq.

## 2.2.22 2.1.11

- remove support for LABEL prefixes. A major feature of libzmq-3.0, the LABEL prefix, has been removed from libzmq, prior to the first stable libzmq 3.x release.
  - The prefix argument to *`send_multipart()`* remains, but it continue to behave in exactly the same way as it always has on 2.1.x, simply prepending message parts.
  - *`recv_multipart()`* will always return a list, because prefixes are once again indistinguishable from regular message parts.
- add *`Socket.poll()`* method, for simple polling of events on a single socket.
- no longer require monkeypatching tornado IOLoop. The *`ioloop.ZMQPoller`* class is a poller implementation that matches tornado's expectations, and pyzmq sockets can be used with any tornado application just by specifying the use of this poller. The pyzmq IOLoop implementation now only trivially differs from tornado's.

  It is still recommended to use *`ioloop.install()`*, which sets *both* the zmq and tornado global IOLoop instances to the same object, but it is no longer necessary.

  **Warning:** The most important part of this change is that the `IOLoop.READ/WRITE/ERROR` constants now match tornado's, rather than being mapped directly to the zmq `POLLIN/OUT/ERR`. So applications that used the low-level `IOLoop.add_handler()` code with `POLLIN/OUT/ERR` directly (used to work, but was incorrect), rather than using the IOLoop class constants will no longer work. Fixing these to use the IOLoop constants should be insensitive to the actual value of the constants.

### 2.2.23 2.1.10

- Add support for libzmq-3.0 LABEL prefixes:

  **Warning:** This feature has been removed from libzmq, and thus removed from future pyzmq as well.

  - send a message with label-prefix with:

```
send_multipart([b'msg', b'parts'], prefix=[b'label', b'prefix'])
```

  - `recv_multipart()` returns a tuple of `(prefix,msg)` if a label prefix is detected
  - ZMQStreams and devices also respect the LABEL prefix

- add czmq-style close&term as `ctx.destroy()`, so that `ctx.term()` remains threadsafe and 1:1 with libzmq.
- `Socket.close()` takes optional linger option, for setting linger prior to closing.
- add `zmq_version_info()` and `pyzmq_version_info()` for getting libzmq and pyzmq versions as tuples of numbers. This helps with the fact that version string comparison breaks down once versions get into double-digits.
- ioloop changes merged from upstream Tornado 2.1

### 2.2.24 2.1.9

- added zmq.ssh tools for tunneling socket connections, copied from IPython
- Expanded sockopt support to cover changes in libzmq-4.0 dev.
- Fixed an issue that prevented `KeyboardInterrupts` from being catchable.
- Added attribute-access for set/getsockopt. Setting/Getting attributes of `Sockets` with the names of socket options is mapped to calls of set/getsockopt.

```
s.hwm = 10
s.identity = b'whoda'
s.linger
# -1
```

- Terminating a `Context` closes the sockets it created, matching the behavior in czmq.
- `ThreadDevices` use `Context.instance()` to create sockets, so they can use inproc connections to sockets in other threads.
- fixed units error on *`zmq.select()`*, where the poll timeout was 1000 times longer than expected.
- Add missing `DEALER/ROUTER` socket type names (currently aliases, to be replacements for `XREP/XREQ`).
- base libzmq dependency raised to 2.1.4 (first stable release) from 2.1.0.

### 2.2.25 2.1.7.1

- bdist for 64b Windows only. This fixed a type mismatch on the `ZMQ_FD` sockopt that only affected that platform.

### 2.2.26 2.1.7

- Added experimental support for libzmq-3.0 API
- Add `zmq.eventloop.ioloop.install()` for using pyzmq's IOLoop in a tornado application.

### 2.2.27 2.1.4

- First version with binary distribution support
- Added `instance()` method for using a single Context throughout an application without passing references around.

## 2.3 More Than Just Bindings

PyZMQ is ostensibly the Python bindings for ØMQ, but the project, following Python's 'batteries included' philosophy, provides more than just Python methods and objects for calling into the ØMQ C++ library.

### 2.3.1 The Core as Bindings

PyZMQ is currently broken up into four subpackages. First, is the Core. `zmq.core` contains the actual bindings for ZeroMQ, and no extended functionality beyond the very basic. The core modules are split, such that each basic ZeroMQ object (or function, if no object is associated) is a separate module, e.g. `zmq.core.context` contains the `Context` object, `zmq.core.poll` contains a `Poller` object, as well as the `select()` function, etc. ZMQ constants are, for convenience, all kept together in `zmq.core.constants`.

There are two reasons for breaking the core into submodules: *recompilation* and *derivative projects*. The monolithic PyZMQ became quite tedious to have to recompile everything for a small change to a single object. With separate files, that's no longer necessary. The second reason has to do with Cython. PyZMQ is written in Cython, a tool for efficiently writing C-extensions for Python. By separating out our objects into individual *pyx* files, each with their declarations in a *pxd* header, other projects can write extensions in Cython and call directly to ZeroMQ at the C-level without the penalty of going through our Python objects.

### 2.3.2 Thread Safety

In ØMQ, Contexts are threadsafe objects, but Sockets are **not**. It is safe to use a single Context (e.g. via `zmq.Context.instance()`) in your entire multithreaded application, but you should create sockets on a per-thread basis. If you share sockets across threads, you are likely to encounter uncatchable c-level crashes of your application unless you use judicious application of `threading.Lock`, but this approach is not recommended.

**See also:**

ZeroMQ API note on threadsafety on 2.2 or 3.2

### 2.3.3 Socket Options as Attributes

New in version 2.1.9.

In 0MQ, socket options are set/retrieved with the `set/getsockopt()` methods. With the class-based approach in pyzmq, it would be logical to perform these operations with simple attribute access, and this has been added in pyzmq 2.1.9. Simply assign to or request a Socket attribute with the (case-insensitive) name of a sockopt, and it should behave just as you would expect:

```
s = ctx.socket(zmq.DEALER)
s.identity = b'dealer'
s.hwm = 10
s.events
# 0
s.fd
# 16
```

#### Default Options on the Context

New in version 2.1.11.

Just like setting socket options as attributes on Sockets, you can do the same on Contexts. This affects the default options of any *new* sockets created after the assignment.

```
ctx = zmq.Context()
ctx.linger = 0
rep = ctx.socket(zmq.REP)
req = ctx.socket(zmq.REQ)
```

Socket options that do not apply to a socket (e.g. SUBSCRIBE on non-SUB sockets) will simply be ignored.

### 2.3.4 Core Extensions

We have extended the core functionality in two ways that appear inside the `core` bindings, and are not general ØMQ features.

#### Builtin Serialization

First, we added common serialization with the builtin `json` and `pickle` as first-class methods to the `Socket` class. A socket has the methods *`send_json()`* and *`send_pyobj()`*, which correspond to sending an object over the wire after serializing with `json` and `pickle` respectively, and any object sent via those methods can be reconstructed with the *`recv_json()`* and *`recv_pyobj()`* methods. Unicode strings are other objects that are not unambiguously sendable over the wire, so we include *`send_string()`* and *`recv_string()`* that simply send bytes after encoding the message ('utf-8' is the default).

**See also:**

- *Further information* on serialization in pyzmq.
- *Our Unicode discussion* for more information on the trials and tribulations of working with Unicode in a C extension while supporting Python 2 and 3.

#### MessageTracker

The second extension of basic ØMQ functionality is the `MessageTracker`. The MessageTracker is an object used to track when the underlying ZeroMQ is done with a message buffer. One of the main use cases for ØMQ in Python is the ability to perform non-copying sends. Thanks to Python's buffer interface, many objects (including NumPy arrays) provide the buffer interface, and are thus directly sendable. However, as with any asynchronous non-copying messaging system like ØMQ or MPI, it can be important to know when the message has actually been sent, so it is safe again to edit the buffer without worry of corrupting the message. This is what the MessageTracker is for.

The MessageTracker is a simple object, but there is a penalty to its use. Since by its very nature, the MessageTracker must involve threadsafe communication (specifically a builtin `Queue` object), instantiating a MessageTracker takes

a modest amount of time (10s of µs), so in situations instantiating many small messages, this can actually dominate performance. As a result, tracking is optional, via the `track` flag, which is optionally passed, always defaulting to `False`, in each of the three places where a Frame object (the pyzmq object for wrapping a segment of a message) is instantiated: The *`Frame`* constructor, and non-copying sends and receives.

A MessageTracker is very simple, and has just one method and one attribute. The property `MessageTracker.done` will be `True` when the Frame(s) being tracked are no longer in use by ØMQ, and *`MessageTracker.wait()`* will block, waiting for the Frame(s) to be released.

---

**Note:** A Frame cannot be tracked after it has been instantiated without tracking. If a Frame is to even have the *option* of tracking, it must be constructed with `track=True`.

---

### 2.3.5 Extensions

So far, PyZMQ includes four extensions to core ØMQ that we found basic enough to be included in PyZMQ itself:

- *zmq.log* : Logging handlers for hooking Python logging up to the network
- *zmq.devices* : Custom devices and objects for running devices in the background
- *zmq.eventloop* : The Tornado event loop, adapted for use with ØMQ sockets.
- *zmq.ssh* : Simple tools for tunneling zeromq connections via ssh.

## 2.4 Serializing messages with PyZMQ

When sending messages over a network, you often need to marshall your data into bytes.

### 2.4.1 Builtin serialization

PyZMQ is primarily bindings for libzmq, but we do provide three builtin serialization methods for convenience, to help Python developers learn libzmq. Python has two primary packages for serializing objects: `json` and `pickle`, so we provide simple convenience methods for sending and receiving objects serialized with these modules. A socket has the methods *`send_json()`* and *`send_pyobj()`*, which correspond to sending an object over the wire after serializing with json and pickle respectively, and any object sent via those methods can be reconstructed with the *`recv_json()`* and *`recv_pyobj()`* methods.

These methods designed for convenience, not for performance, so developers who do want to emphasize performance should use their own serialized send/recv methods.

### 2.4.2 Using your own serialization

In general, you will want to provide your own serialization that is optimized for your application or library availability. This may include using your own preferred serialization (*[msgpack]*, *[protobuf]*), or adding compression via *[zlib]* in the standard library, or the super fast *[blosc]* library.

There are two simple models for implementing your own serialization: write a function that takes the socket as an argument, or subclass Socket for use in your own apps.

For instance, pickles can often be reduced substantially in size by compressing the data. The following will send *compressed* pickles over the wire:

```
import zlib, cPickle as pickle

def send_zipped_pickle(socket, obj, flags=0, protocol=-1):
    """pickle an object, and zip the pickle before sending it"""
    p = pickle.dumps(obj, protocol)
    z = zlib.compress(p)
    return socket.send(z, flags=flags)

def recv_zipped_pickle(socket, flags=0, protocol=-1):
    """inverse of send_zipped_pickle"""
    z = socket.recv(flags)
    p = zlib.decompress(z)
    return pickle.loads(p)
```

A common data structure in Python is the numpy array. PyZMQ supports sending numpy arrays without copying any data, since they provide the Python buffer interface. However just the buffer is not enough information to reconstruct the array on the receiving side. Here is an example of a send/recv that allow non-copying sends/recvs of numpy arrays including the dtype/shape data necessary for reconstructing the array.

```
import numpy

def send_array(socket, A, flags=0, copy=True, track=False):
    """send a numpy array with metadata"""
    md = dict(
        dtype = str(A.dtype),
        shape = A.shape,
    )
    socket.send_json(md, flags|zmq.SNDMORE)
    return socket.send(A, flags, copy=copy, track=track)

def recv_array(socket, flags=0, copy=True, track=False):
    """recv a numpy array"""
    md = socket.recv_json(flags=flags)
    msg = socket.recv(flags=flags, copy=copy, track=track)
    buf = buffer(msg)
    A = numpy.frombuffer(buf, dtype=md['dtype'])
    return A.reshape(md['shape'])
```

## 2.5 Devices in PyZMQ

**See also:**

ØMQ Guide Device coverage.

ØMQ has a notion of Devices - simple programs that manage a send-recv pattern for connecting two or more sockets. Being full programs, devices include a `while(True)` loop and thus block execution permanently once invoked. We have provided in the `devices` subpackage some facilities for running these devices in the background, as well as a custom three-socket *MonitoredQueue* device.

### 2.5.1 BackgroundDevices

It seems fairly rare that in a Python program one would actually want to create a zmq device via *`device()`* in the main thread, since such a call would block execution forever. The most likely model for launching devices is in background threads or processes. We have provided classes for launching devices in a background thread with *`ThreadDevice`* and via multiprocessing with *`ProcessDevice`*. For threadsafety and running across processes, these methods do

not take Socket objects as arguments, but rather socket types, and then the socket creation and configuration happens via the BackgroundDevice's `foo_in()` proxy methods. For each configuration method (bind/connect/setsockopt), there are proxy methods for calling those methods on the Socket objects created in the background thread or process, prefixed with 'in_' or 'out_', corresponding to the *in_socket* and *out_socket*:

```
from zmq.devices import ProcessDevice

pd = ProcessDevice(zmq.QUEUE, zmq.ROUTER, zmq.DEALER)
pd.bind_in('tcp://*:12345')
pd.connect_out('tcp://127.0.0.1:12543')
pd.setsockopt_in(zmq.IDENTITY, 'ROUTER')
pd.setsockopt_out(zmq.IDENTITY, 'DEALER')
pd.start()
# it will now be running in a background process
```

### 2.5.2 MonitoredQueue

One of ØMQ's builtin devices is the `QUEUE`. This is a symmetric two-socket device that fully supports passing messages in either direction via any pattern. We saw a logical extension of the `QUEUE` as one that behaves in the same way with respect to the in/out sockets, but also sends every message in either direction *also* on a third *monitor* socket. For performance reasons, this *`monitored_queue()`* function is written in Cython, so the loop does not involve Python, and should have the same performance as the basic `QUEUE` device.

One shortcoming of the `QUEUE` device is that it does not support having `ROUTER` sockets as both input and output. This is because `ROUTER` sockets, when they receive a message, prepend the `IDENTITY` of the socket that sent the message (for use in routing the reply). The result is that the output socket will always try to route the incoming message back to the original sender, which is presumably not the intended pattern. In order for the queue to support a ROUTER-ROUTER connection, it must swap the first two parts of the message in order to get the right message out the other side.

To invoke a monitored queue is similar to invoking a regular ØMQ device:

```
from zmq.devices import monitored_queue
ins = ctx.socket(zmq.ROUTER)
outs = ctx.socket(zmq.DEALER)
mons = ctx.socket(zmq.PUB)
configure_sockets(ins,outs,mons)
monitored_queue(ins, outs, mons, in_prefix='in', out_prefix='out')
```

The *in_prefix* and *out_prefix* default to 'in' and 'out' respectively, and a PUB socket is most logical for the monitor socket, since it will never receive messages, and the in/out prefix is well suited to the PUB/SUB topic subscription model. All messages sent on *mons* will be multipart, the first part being the prefix corresponding to the socket that received the message.

Or for launching an MQ in the background, there are *`ThreadMonitoredQueue`* and *`ProcessMonitoredQueue`*, which function just like the base BackgroundDevice objects, but add `foo_mon()` methods for configuring the monitor socket.

## 2.6 Eventloops and PyZMQ

Integrating zmq with eventloops is *almost* really easy, since most eventloops happily support sockets. What gets messy is that zmq sockets aren't regular sockets, so they need special handling. libzmq provides a `zmq_poll()` function that is the same as regular polling, but **also** support zmq sockets. PyZMQ wrapps this in a *`Poller`* class. Most of pyzmq's eventloop support involves setting up existing eventloops (tornado, asyncio) to use `zmq_poll()` as the

inner poller, rather than the default select/poll/etc. Once that's done, zmq sockets can be happily treated like regular sockets, and regular sockets should continue to work as before.

---

**Note:** It *is* possible to integrate zmq sockets into existing eventloops without modifying the poller by using the `socket.FD` attribute. The incredibly unfortunate aspect of this is that it was implemented as an edge-triggered fd, which is highly error prone, and I wouldn't recommend using unless absolutely necessary. This is used in *zmq.green*, and has been the source of many problems.

---

### 2.6.1 Tornado IOLoop

Facebook's Tornado includes an eventloop for handing poll events on filedescriptors and native sockets. We have included a small part of Tornado (specifically its *ioloop*), and adapted its `IOStream` class into *ZMQStream* for handling poll events on ØMQ sockets. A ZMQStream object works much like a Socket object, but instead of calling *recv()* directly, you register a callback with *on_recv()*. Callbacks can also be registered for send events with *on_send()*.

#### install()

With PyZMQ's ioloop, you can use zmq sockets in any tornado application. You can tell tornado to use zmq's poller by calling the *ioloop.install()* function:

```
from zmq.eventloop import ioloop
ioloop.install()
```

You can also do the same thing by requesting the global instance from pyzmq:

```
from zmq.eventloop.ioloop import IOLoop
loop = IOLoop.current()
```

This configures tornado's `tornado.ioloop.IOLoop` to use zmq's poller, and registers the current instance.

Either `install()` or retrieving the zmq instance must be done before the global * instance is registered, else there will be a conflict.

It is possible to use PyZMQ sockets with tornado *without* registering as the global instance, but it is less convenient. First, you must instruct the tornado IOLoop to use the zmq poller:

```
from zmq.eventloop.ioloop import ZMQIOLoop

loop = ZMQIOLoop()
```

Then, when you instantiate tornado and ZMQStream objects, you must pass the *io_loop* argument to ensure that they use this loop, instead of the global instance.

This is especially useful for writing tests, such as this:

```
from tornado.testing import AsyncTestCase
from zmq.eventloop.ioloop import ZMQIOLoop
from zmq.eventloop.zmqstream import ZMQStream

class TestZMQBridge(AsyncTestCase):

    # Use a ZMQ-compatible I/O loop so that we can use `ZMQStream`.
    def get_new_ioloop(self):
        return ZMQIOLoop()
```

You can also manually install this IOLoop as the global tornado instance, with:

```
from zmq.eventloop.ioloop import ZMQIOLoop
loop = ZMQIOLoop()
loop.install()
```

### Futures and coroutines

PyZMQ 15 adds *zmq.eventloop.future*, containing a Socket subclass that returns `Future` objects for use in `tornado` coroutines.

#### ZMQStream

`ZMQStream` objects let you register callbacks to handle messages as they arrive, for use with the tornado eventloop.

#### send()

ZMQStream objects do have *send()* and *send_multipart()* methods, which behaves the same way as *Socket.send()*, but instead of sending right away, the `IOLoop` will wait until socket is able to send (for instance if `HWM` is met, or a `REQ/REP` pattern prohibits sending at a certain point). Messages sent via send will also be passed to the callback registered with *on_send()* after sending.

#### on_recv()

*ZMQStream.on_recv()* is the primary method for using a ZMQStream. It registers a callback to fire with messages as they are received, which will *always* be multipart, even if its length is 1. You can easily use this to build things like an echo socket:

```
s = ctx.socket(zmq.REP)
s.bind('tcp://localhost:12345')
stream = ZMQStream(s)
def echo(msg):
    stream.send_multipart(msg)
stream.on_recv(echo)
ioloop.IOLoop.instance().start()
```

on_recv can also take a *copy* flag, just like *Socket.recv()*. If *copy=False*, then callbacks registered with on_recv will receive tracked *Frame* objects instead of bytes.

**Note:** A callback must be registered using either *ZMQStream.on_recv()* or *ZMQStream.on_recv_stream()* before any data will be received on the underlying socket. This allows you to temporarily pause processing on a socket by setting both callbacks to None. Processing can later be resumed by restoring either callback.

#### on_recv_stream()

*ZMQStream.on_recv_stream()* is just like on_recv above, but the callback will be passed both the message and the stream, rather than just the message. This is meant to make it easier to use a single callback with multiple streams.

```
s1 = ctx.socket(zmq.REP)
s1.bind('tcp://localhost:12345')
stream1 = ZMQStream(s1)

s2 = ctx.socket(zmq.REP)
s2.bind('tcp://localhost:54321')
stream2 = ZMQStream(s2)

def echo(stream, msg):
    stream.send_multipart(msg)

stream1.on_recv_stream(echo)
stream2.on_recv_stream(echo)

ioloop.IOLoop.instance().start()
```

**flush()**

Sometimes with an eventloop, there can be multiple events ready on a single iteration of the loop. The *flush()* method allows developers to pull messages off of the queue to enforce some priority over the event loop ordering. flush pulls any pending events off of the queue. You can specify to flush only recv events, only send events, or any events, and you can specify a limit for how many events to flush in order to prevent starvation.

### 2.6.2 AsyncIO

PyZMQ 15 adds support for asyncio via *zmq.asyncio*.

### 2.6.3 PyZMQ and gevent

PyZMQ 2.2.0.1 ships with a gevent compatible API as *zmq.green*. To use it, simply:

```
import zmq.green as zmq
```

Then write your code as normal.

Socket.send/recv and zmq.Poller are gevent-aware.

In PyZMQ 2.2.0.2, green.device and green.eventloop should be gevent-friendly as well.

---

**Note:** The green device does *not* release the GIL, unlike the true device in zmq.core.

---

zmq.green.eventloop includes minimally patched IOLoop/ZMQStream in order to use the gevent-enabled Poller, so you should be able to use the ZMQStream interface in gevent apps as well, though using two eventloops simultaneously (tornado + gevent) is not recommended.

**Warning:** There is a known issue in gevent 1.0 or libevent, which can cause zeromq socket events to be missed. PyZMQ works around this by adding a timeout so it will not wait forever for gevent to notice events. The only known solution for this is to use gevent 1.0, which is currently at 1.0b3, and does not exhibit this behavior.

**See also:**

zmq.green examples on GitHub.

*zmq.green* is simply gevent_zeromq, merged into the pyzmq project.

## 2.7 Asynchronous Logging via PyZMQ

**See also:**

- The ØMQ guide coverage of PUB/SUB messaging
- Python logging module documentation

Python provides extensible logging facilities through its `logging` module. This module allows for easily extensible logging functionality through the use of `Handler` objects. The most obvious case for hooking up pyzmq to logging would be to broadcast log messages over a PUB socket, so we have provided a *PUBHandler* class for doing just that.

### 2.7.1 PUB/SUB and Topics

The ØMQ PUB/SUB pattern consists of a PUB socket broadcasting messages, and a collection of SUB sockets that receive those messages. Each PUB message is a multipart-message, where the first part is interpreted as a topic. SUB sockets can subscribe to topics by setting their `SUBSCRIBE` sockopt, e.g.:

```
sub = ctx.socket(zmq.SUB)
sub.setsockopt(zmq.SUBSCRIBE, 'topic1')
sub.setsockopt(zmq.SUBSCRIBE, 'topic2')
```

When subscribed, the SUB socket will only receive messages where the first part *starts with* one of the topics set via `SUBSCRIBE`. The default behavior is to exclude all messages, and subscribing to the empty string ‘’ will receive all messages.

### 2.7.2 PUBHandler

The *PUBHandler* object is created for allowing the python logging to be emitted on a PUB socket. The main difference between a PUBHandler and a regular logging Handler is the inclusion of topics. For the most basic logging, you can simply create a PUBHandler with an interface or a configured PUB socket, and just let it go:

```
pub = context.socket(zmq.PUB)
pub.bind('tcp://*:12345')
handler = PUBHandler(pub)
logger = logging.getLogger()
logger.addHandler(handler)
```

At this point, all messages logged with the default logger will be broadcast on the pub socket.

the PUBHandler does work with topics, and the handler has an attribute `root_topic`:

```
handler.root_topic = 'myprogram'
```

Python loggers also have loglevels. The base topic of messages emitted by the PUBHandler will be of the form: `<handler.root_topic>.<loglevel>`, e.g. ‘myprogram.INFO’ or ‘whatever.ERROR’. This way, subscribers can easily subscribe to subsets of the logging messages. Log messages are always two-part, where the first part is the topic tree, and the second part is the actual log message.

```
>>> logger.info('hello there')
>>> print sub.recv_multipart()
['myprogram.INFO', 'hello there']
```

### Subtopics

You can also add to the topic tree below the loglevel on an individual message basis. Assuming your logger is connected to a PUBHandler, you can add as many additional topics on the front of the message, which will be added always after the loglevel. A special delimiter defined at `zmq.log.handlers.TOPIC_DELIM` is scanned by the PUBHandler, so if you pass your own subtopics prior to that symbol, they will be stripped from the message and added to the topic tree:

```
>>> log_msg  = "hello there"
>>> subtopic = "sub.topic"
>>> msg = zmq.log.handlers.TOPIC_DELIM.join([subtopic, log_msg])
>>> logger.warn(msg)
>>> print sub.recv_multipart()
['myprogram.WARN.sub.topic', 'hello there']
```

## 2.8 Tunneling PyZMQ Connections with SSH

New in version 2.1.9.

You may want to connect ØMQ sockets across machines, or untrusted networks. One common way to do this is to tunnel the connection via SSH. IPython introduced some tools for tunneling ØMQ connections over ssh in simple cases. These functions have been brought into pyzmq as `zmq.ssh` under IPython's BSD license.

PyZMQ will use the shell ssh command via pexpect by default, but it also supports using paramiko for tunnels, so it should work on Windows.

An SSH tunnel has five basic components:

- server : the SSH server through which the tunnel will be created
- remote ip : the IP of the remote machine *as seen from the server* (remote ip may be, but is not not generally the same machine as server).
- remote port : the port on the remote machine that you want to connect to.
- local ip : the interface on your local machine you want to use (default: 127.0.0.1)
- local port : the local port you want to forward to the remote port (default: high random)

So once you have established the tunnel, connections to `localip:localport` will actually be connections to `remoteip:remoteport`.

In most cases, you have a zeromq url for a remote machine, but you need to tunnel the connection through an ssh server. This is

So if you would use this command from the same LAN as the remote machine:

```
sock.connect("tcp://10.0.1.2:5555")
```

to make the same connection from another machine that is outside the network, but you have ssh access to a machine `server` on the same LAN, you would simply do:

```
from zmq import ssh
ssh.tunnel_connection(sock, "tcp://10.0.1.2:5555", "server")
```

Note that `"server"` can actually be a fully specified `"user@server:port"` ssh url. Since this really just launches a shell command, all your ssh configuration of usernames, aliases, keys, etc. will be respected. If necessary, `tunnel_connection()` does take arguments for specific passwords, private keys (the ssh `-i` option), and non-default choice of whether to use paramiko.

If you are on the same network as the machine, but it is only listening on localhost, you can still connect by making the machine itself the server, and using loopback as the remote ip:

```
from zmq import ssh
ssh.tunnel_connection(sock, "tcp://127.0.0.1:5555", "10.0.1.2")
```

The `tunnel_connection()` function is a simple utility that forwards a random localhost port to the real destination, and connects a socket to the new local url, rather than the remote one that wouldn't actually work.

**See also:**

A short discussion of ssh tunnels: http://www.revsys.com/writings/quicktips/ssh-tunnel.html

CHAPTER 3

# Notes from developing PyZMQ

## 3.1 PyZMQ, Python2.5, and Python3

PyZMQ is a fairly light, low-level library, so supporting as many versions as is reasonable is our goal. Currently, we support at least Python 2.5-3.1. Making the changes to the codebase required a few tricks, which are documented here for future reference, either by us or by other developers looking to support several versions of Python.

**Note:** It is far simpler to support 2.6-3.x than to include 2.5. Many of the significant syntax changes have been backported to 2.6, so just writing new-style code would work in many cases. I will try to note these points as they come up.

### 3.1.1 pyversion_compat.h

Many functions we use, primarily involved in converting between C-buffers and Python objects, are not available on all supported versions of Python. In order to resolve missing symbols, we added a header `utils/pyversion_compat.h` that defines missing symbols with macros. Some of these macros alias new names to old functions (e.g. `PyBytes_AsString`), so that we can call new-style functions on older versions, and some simply define the function as an empty exception raiser. The important thing is that the symbols are defined to prevent compiler warnings and linking errors. Everywhere we use C-API functions that may not be available in a supported version, at the top of the file is the code:

```
cdef extern from "pyversion_compat.h":
    pass
```

This ensures that the symbols are defined in the Cython generated C-code. Higher level switching logic exists in the code itself, to prevent actually calling unavailable functions, but the symbols must still be defined.

### 3.1.2 Bytes and Strings

**Note:** If you are using Python >= 2.6, to prepare your PyZMQ code for Python3 you should use the `b'message'` syntax to ensure all your string literal messages will still be `bytes` after you make the upgrade.

The most cumbersome part of PyZMQ compatibility from a user's perspective is the fact that, since ØMQ uses C-strings, and would like to do so without copying, we must use the Py3k `bytes` object, which is backported to 2.6. In order to do this in a Python-version independent way, we added a small utility that unambiguously defines the string

types: bytes, unicode, basestring. This is important, because str means different things on 2.x and 3.x, and bytes is undefined on 2.5, and both unicode and basestring are undefined on 3.x. All typechecking in PyZMQ is done against these types:

| Explicit Type | 2.x | 3.x |
|---|---|---|
| bytes | str | bytes |
| unicode | unicode | str |
| basestring | basestring | (str, bytes) |

**Note:** 2.5 specific

Where we really noticed the issue of bytes vs strings coming up for users was in updating the tests to run on every version. Since the b'bytes literal' syntax was not backported to 2.5, we must call "message".encode() for *every* string in the test suite.

**See also:**

*Unicode discussion* for more information on strings/bytes.

#### PyBytes_*

The standard C-API function for turning a C-string into a Python string was a set of functions with the prefix PyString_*. However, with the Unicode changes made in Python3, this was broken into PyBytes_* for bytes objects and PyUnicode_* for unicode objects. We changed all our PyString_* code to PyBytes_*, which was backported to 2.6.

**Note:** 2.5 Specific:

Since Python 2.5 doesn't support the PyBytes_* functions, we had to alias them to the PyString_* methods in utils/pyversion_compat.h.

```
#define PyBytes_FromStringAndSize PyString_FromStringAndSize
#define PyBytes_FromString PyString_FromString
#define PyBytes_AsString PyString_AsString
#define PyBytes_Size PyString_Size
```

### 3.1.3 Buffers

The layer that is most complicated for developers, but shouldn't trouble users, is the Python C-Buffer APIs. These are the methods for converting between Python objects and C buffers. The reason it is complicated is that it keeps changing.

There are two buffer interfaces for converting an object to a C-buffer, known as new-style and old-style. Old-style buffers were introduced long ago, but the new-style is only backported to 2.6. The old-style buffer interface is not available in 3.x. There is also an old- and new-style interface for creating Python objects that view C-memory. The old-style object is called a buffer, and the new-style object is memoryview. Unlike the new-style buffer interface for objects, memoryview has only been backported to *2.7*. This means that the available buffer-related functions are not the same in any two versions of Python 2.5, 2.6, 2.7, or 3.1.

We have a utils/buffers.pxd file that defines our asbuffer() and frombuffer() functions. utils/buffers.pxd was adapted from mpi4py's asbuffer.pxi. The frombuffer() functionality was added. These functions internally switch based on Python version to call the appropriate C-API functions.

**See also:**

Python Buffer API

### 3.1.4 __str__

As discussed, `str` is not a platform independent type. The two places where we are required to return native str objects are `error.strerror()`, and `Message.__str__()`. In both of these cases, the natural return is actually a `bytes` object. In the methods, the native `str` type is checked, and if the native str is actually unicode, then we decode the bytes into unicode:

```
# ...
b = natural_result()
if str is unicode:
    return b.decode()
else:
    return b
```

### 3.1.5 Exceptions

**Note:** This section is only relevant for supporting Python 2.5 and 3.x, not for 2.6-3.x.

The syntax for handling exceptions has changed in Python 3. The old syntax:

```
try:
    s.send(msg)
except zmq.ZMQError, e:
    handle(e)
```

is no longer valid in Python 3. Instead, the new syntax for this is:

```
try:
    s.send(msg)
except zmq.ZMQError as e:
    handle(e)
```

This new syntax is backported to Python 2.6, but is invalid on 2.5. For 2.6-3.x compatible code, we could just use the new syntax. However, the only method we found to catch an exception for handling on both 2.5 and 3.1 is to get the exception object inside the exception block:

```
try:
    s.send(msg)
except zmq.ZMQError:
    e = sys.exc_info()[1]
    handle(e)
```

This is certainly not as elegant as either the old or new syntax, but it's the only way we have found to work everywhere.

**See also:**

PEP-3110

## 3.2 PyZMQ and Unicode

PyZMQ is built with an eye towards an easy transition to Python 3, and part of that is dealing with unicode strings. This is an overview of some of what we found, and what it means for PyZMQ.

### 3.2.1 First, Unicode in Python 2 and 3

In Python < 3, a `str` object is really a C string with some sugar - a specific series of bytes with some fun methods like `endswith()` and `split()`. In 2.0, the `unicode` object was added, which handles different methods of encoding. In Python 3, however, the meaning of `str` changes. A `str` in Python 3 is a full unicode object, with encoding and everything. If you want a C string with some sugar, there is a new object called `bytes`, that behaves much like the 2.x `str`. The idea is that for a user, a string is a series of *characters*, not a series of bytes. For simple ascii, the two are interchangeable, but if you consider accents and non-Latin characters, then the character meaning of byte sequences can be ambiguous, since it depends on the encoding scheme. They decided to avoid the ambiguity by forcing users who want the actual bytes to specify the encoding every time they want to convert a string to bytes. That way, users are aware of the difference between a series of bytes and a collection of characters, and don't confuse the two, as happens in Python 2.x.

The problems (on both sides) come from the fact that regardless of the language design, users are mostly going to use `str` objects to represent collections of characters, and the behavior of that object is dramatically different in certain aspects between the 2.x `bytes` approach and the 3.x `unicode` approach. The `unicode` approach has the advantage of removing byte ambiguity - it's a list of characters, not bytes. However, if you really do want the bytes, it's very inefficient to get them. The `bytes` approach has the advantage of efficiency. A `bytes` object really is just a char* pointer with some methods to be used on it, so when interacting with, so interacting with C code, etc is highly efficient and straightforward. However, understanding a bytes object as a string with extended characters introduces ambiguity and possibly confusion.

To avoid ambiguity, hereafter we will refer to encoded C arrays as 'bytes' and abstract unicode objects as 'strings'.

#### Unicode Buffers

Since unicode objects have a wide range of representations, they are not stored as the bytes according to their encoding, but rather in a format called UCS (an older fixed-width Unicode format). On some platforms (OS X, Windows), the storage is UCS-2, which is 2 bytes per character. On most *ix systems, it is UCS-4, or 4 bytes per character. The contents of the *buffer* of a `unicode` object are not encoding dependent (always UCS-2 or UCS-4), but they are *platform* dependent. As a result of this, and the further insistence on not interpreting `unicode` objects as bytes without specifying encoding, `str` objects in Python 3 don't even provide the buffer interface. You simply cannot get the raw bytes of a `unicode` object without specifying the encoding for the bytes. In Python 2.x, you can get to the raw buffer, but the platform dependence and the fact that the encoding of the buffer is not the encoding of the object makes it very confusing, so this is probably a good move.

The efficiency problem here comes from the fact that simple ascii strings are 4x as big in memory as they need to be (on most Linux, 2x on other platforms). Also, to translate to/from C code that works with char*, you always have to copy data and encode/decode the bytes. This really is horribly inefficient from a memory standpoint. Essentially, Where memory efficiency matters to you, you should never ever use strings; use bytes. The problem is that users will almost always use `str`, and in 2.x they are efficient, but in 3.x they are not. We want to make sure that we don't help the user make this mistake, so we ensure that zmq methods don't try to hide what strings really are.

### 3.2.2 What This Means for PyZMQ

PyZMQ is a wrapper for a C library, so it really should use bytes, since a string is not a simple wrapper for `char *` like it used to be, but an abstract sequence of characters. The representations of bytes in Python are either the `bytes` object itself, or any object that provides the buffer interface (aka memoryview). In Python 2.x, unicode objects do provide the buffer interface, but as they do not in Python 3, where pyzmq requires bytes, we specifically reject unicode objects.

The relevant methods here are `socket.send/recv`, `socket.get/setsockopt`, `socket.bind/connect`. The important consideration for send/recv and set/getsockopt is that when you put in something, you really should get the same object back with its partner method. We can easily coerce unicode objects to bytes with send/setsockopt,

but the problem is that the pair method of recv/getsockopt will always be bytes, and there should be symmetry. We certainly shouldn't try to always decode on the retrieval side, because if users just want bytes, then we are potentially using up enormous amounts of excess memory unnecessarily, due to copying and larger memory footprint of unicode strings.

Still, we recognize the fact that users will quite frequently have unicode strings that they want to send, so we have added `socket.<method>_string()` wrappers. These methods simply wrap their bytes counterpart by encoding to/decoding from bytes around them, and they all take an *encoding* keyword argument that defaults to utf-8. Since encoding and decoding are necessary to translate between unicode and bytes, it is impossible to perform non-copying actions with these wrappers.

`socket.bind/connect` methods are different from these, in that they are strictly setters and there is not corresponding getter method. As a result, we feel that we can safely coerce unicode objects to bytes (always to utf-8) in these methods.

---

**Note:** For cross-language symmetry (including Python 3), the `_unicode` methods are now `_string`. Many languages have a notion of native strings, and the use of `_unicode` was wedded too closely to the name of such objects in Python 2. For the time being, anywhere you see `_string`, `_unicode` also works, and is the only option in pyzmq 2.1.11.

---

### The Methods

Overview of the relevant methods:

`socket.`**`bind`**(*self*, *addr*)
*addr* is `bytes` or `unicode`. If `unicode`, encoded to utf-8 `bytes`

`socket.`**`connect`**(*self*, *addr*)
*addr* is `bytes` or `unicode`. If `unicode`, encoded to utf-8 `bytes`

`socket.`**`send`**(*self*, *object obj*, *flags=0*, *copy=True*)
*obj* is `bytes` or provides buffer interface.

if *obj* is `unicode`, raise `TypeError`

`socket.`**`recv`**(*self*, *flags=0*, *copy=True*)
returns `bytes` if *copy=True*

returns `zmq.Message` if *copy=False*:

*message.buffer* is a buffer view of the `bytes`

*str(message)* provides the `bytes`

*unicode(message)* decodes *message.buffer* with utf-8

`socket.`**`send_string`**(*self*, *unicode s*, *flags=0*, *encoding='utf-8'*)
takes a `unicode` string *s*, and sends the `bytes` after encoding without an extra copy, via:

*socket.send(s.encode(encoding), flags, copy=False)*

`socket.`**`recv_string`**(*self*, *flags=0*, *encoding='utf-8'*)
always returns `unicode` string

there will be a `UnicodeError` if it cannot decode the buffer

performs non-copying *recv*, and decodes the buffer with *encoding*

`socket.`**`setsockopt`**(*self*, *opt*, *optval*)
only accepts `bytes` for *optval* (or `int`, depending on *opt*)

`TypeError` if `unicode` or anything else

`socket.`**`getsockopt`**(*self*, *opt*)
returns `bytes` (or `int`), never `unicode`

`socket.`**`setsockopt_string`**(*self*, *opt*, *unicode optval*, *encoding='utf-8'*)
accepts `unicode` string for *optval*

encodes *optval* with *encoding* before passing the `bytes` to *setsockopt*

`socket.`**`getsockopt_string`**(*self*, *opt*, *encoding='utf-8'*)
always returns `unicode` string, after decoding with *encoding*

note that *zmq.IDENTITY* is the only *sockopt* with a string value that can be queried with *getsockopt*

CHAPTER 4

# Indices and tables

- genindex
- modindex
- search

CHAPTER 5

# Links

- ØMQ Home
- The ØMQ Guide
- PyZMQ Installation notes on the ZeroMQ website
- PyZMQ on GitHub
- Issue Tracker

# Bibliography

[msgpack] Message Pack serialization library http://msgpack.org

[protobuf] Google Protocol Buffers http://code.google.com/p/protobuf

[zlib] Python stdlib module for zip compression: `zlib`

[blosc] Blosc: A blocking, shuffling and loss-less (and crazy-fast) compression library http://blosc.pytables.org/trac

## Z

## Symbols



## A



## B



## C

## D



## E



## F



## G



## H



## I

## T



## U



## W

## Z