bottom

Release 2.2.0

Joe Cross

Jun 01, 2022

CONTENTS

With a very small API, bottom lets you wrap an IRC connection and handle events how you want. There are no assumptions about reconnecting, rate limiting, or even when to respond to PINGs.

Explicit is better than implicit: no magic importing or naming to remember for plugins. Extend the client with the same `@on` decorator.

Create an instance:

```
import asyncio
import bottom

host = 'chat.freenode.net'
port = 6697
ssl = True

NICK = "bottom-bot"
CHANNEL = "#bottom-dev"

bot = bottom.Client(host=host, port=port, ssl=ssl)
```

Send nick/user/join when connection is established:

```
@bot.on('CLIENT_CONNECT')
async def connect(**kwargs):
    bot.send('NICK', nick=NICK)
    bot.send('USER', user=NICK,
             realname='https://github.com/numberoverzero/bottom')

    # Don't try to join channels until the server has
    # sent the MOTD, or signaled that there's no MOTD.
    done, pending = await asyncio.wait(
        [bot.wait("RPL_ENDOFMOTD"),
         bot.wait("ERR_NOMOTD")],
        loop=bot.loop,
        return_when=asyncio.FIRST_COMPLETED
    )

    # Cancel whichever waiter's event didn't come in.
    for future in pending:
        future.cancel()

    bot.send('JOIN', channel=CHANNEL)
```

Respond to ping:

```
@bot.on('PING')
def keepalive(message, **kwargs):
    bot.send('PONG', message=message)
```

Echo messages (channel and direct messages):

```
@bot.on('PRIVMSG')
def message(nick, target, message, **kwargs):
    """ Echo all messages """

    # Don't echo ourselves
    if nick == NICK:
```

(continues on next page)

(continued from previous page)

```
        return
    # Respond directly to direct messages
    if target == NICK:
        bot.send("PRIVMSG", target=nick, message=message)
    # Channel message
    else:
        bot.send("PRIVMSG", target=target, message=message)
```

Connect and run the bot forever:

```
bot.loop.create_task(bot.connect())
bot.loop.run_forever()
```

CHAPTER

ONE

INSTALLATION

Bottom supports **Python 3.10+** and has no external dependencies.

1.1 Standard Installation

The easiest way is with pip:

```
pip install bottom
```

1.2 Alternative Installation

Sometimes the code on GitHub is ahead of the latest PyPI release. There are two ways to install from GitHub. First, by cloning the repo:

```
git clone git://github.com/numberoverzero/bottom.git
pip install ./bottom
```

Since pip supports installing from a git repo, you can also use:

```
pip install -e git://github.com/numberoverzero/bottom.git#egg=bottom
```

CHAPTER

TWO

USING ASYNC

Bottom accepts bot synchronous and async functions as callbacks. Both of these are valid handlers for the `privmsg` event:

```
@client.on('privmsg')
def synchronous_handler(**kwargs):
    print("Synchronous call")


@client.on('privmsg')
async def async_handler(**kwargs):
    await asyncio.sleep(1)
    print("Async call")
```

2.1 Connect/Disconnect

Client connect and disconnect are coroutines so that we can easily wait for their completion before performing more actions in a handler. However, we don't always want to wait for the action to complete. How can we do both?

Let's say that on disconnect we want to reconnect, then notify the room that we're back. We need to `await` for the connection before sending anything:

```
@client.on('client_disconnect')
async def reconnect(**kwargs):
    # Wait a second so we don't flood
    await asyncio.sleep(2)

    # Wait until we've reconnected
    await client.connect()

    # Notify the room
    client.send('privmsg', target='#bottom-dev',
                message="I'm baaack!")
```

What about a handler that doesn't need an established connection to finish? Instead of notifying the room, let's log the reconnect time and return:

```
import arrow
import asyncio
import logging
logger = logging.getLogger(__name__)
```

(continues on next page)

(continued from previous page)

```
@client.on('client_disconnect')
async def reconnect(**kwargs):
    # Wait a second so we don't flood
    await asyncio.sleep(2)

    # Schedule a connection when the loop's next available
    asyncio.create_task(client.connect())

    # Record the time of the disconnect event
    now = arrow.now()
    logger.info("Reconnect started at " + now.isoformat())
```

We can also wait for the `client_connect` event to trigger, which is slightly different than waiting for client.connect to complete:

```
@client.on('client_disconnect')
async def reconnect(**kwargs):
    # Wait a second so we don't flood
    await asyncio.sleep(2)

    # Schedule a connection when the loop's next available
    asyncio.create_task(client.connect())

    # Wait until client_connect has triggered
    await client.wait("client_connect")

    # Notify the room
    client.send('privmsg', target='#bottom-dev',
                message="I'm baaack!")
```

2.2 Debugging

You can get more asyncio debugging info by running python with the `-X dev` flag:

```
python -X dev my_bot.py
```

For more information, see: Python Development Mode.

CHAPTER

THREE

API

3.1 `Client.on`

```
client.on(event)(func)
```

This decorator is the main way you'll interact with a `Client`. For a given event name, it registers the decorated function to be invoked when that event occurs. Your decorated functions should always accept `**kwargs`, in case unexpected kwargs are included when the event is triggered.

The usual IRC commands sent from a server are triggered automatically, or can be manually invoked with `trigger`. You may register handlers for any string, making it easy to extend bottom with your own signals.

Not all available arguments need to be used. Both of the following are valid:

```
@bot.on('PRIVMSG')
def event(nick, message, target, **kwargs):
    """ Doesn't use user, host.  argument order is different """
    # message sent to bot - echo message
    if target == bot.nick:
        bot.send('PRIVMSG', target, message=message)
    # Some channel we're watching
    elif target == bot.monitored_channel:
        logger.info("{} -> {}: {}".format(nick, target, message))


@bot.on('PRIVMSG')
def func(message, target, **kwargs):
    """ Just waiting for the signal """
    if message == codeword && target == secret_channel:
        execute_heist()
```

Handlers do not need to be async functions - non async will be wrapped prior to the bot running. For example, both of these are valid:

```
@bot.on('PRIVMSG')
def handle(message, **kwargs):
    print(message)

@bot.on('PRIVMSG')
async def handle(message, **kwargs):
    await async_logger.log(message)
```

Finally, you can create your own events to trigger and handle. For example, let's catch SIGINT and gracefully shut down the event loop:

```
import signal

def handle_sigint(signum, frame):
    print("SIGINT handler")
    bot.trigger("my.sigint.event")
signal.signal(signal.SIGINT, handle_sigint)


@bot.on("my.sigint.event")
async def handle(**kwargs):
    print("SIGINT trigger")
    await bot.disconnect()

    # Signal a stop before disconnecting so that any reconnect
    # coros aren't run by the last run_forever sweep.
    bot.loop.stop()


bot.loop.create_task(bot.connect())
bot.loop.run_forever()  # Ctrl + C here
```

3.2 `Client.trigger`

```
client.trigger(event, **kwargs)
```

Manually inject a command or reply as if it came from the server. This is useful for invoking other handlers. Because `trigger` doesn't block, registered callbacks for the event won't run until the event loop yields to them.

Events don't need to be valid irc commands; any string is available.

```
# Manually trigger `PRIVMSG` handlers:
bot.trigger('privmsg', nick="always_says_no", message="yes")
```

```
# Rename !commands to !help
@bot.on('privmsg')
def parse(nick, target, message, **kwargs):
    if message == '!commands':
        bot.send('privmsg', target=nick,
                 message="!commands was renamed to !help in 1.2")
        # Don't make them retype it, trigger the correct command
        bot.trigger('privmsg', nick=nick,
                    target=target, message="!help")
```

Because the `@on` decorator returns the original function, you can register a handler for multiple events. It's especially important to use `**kwargs` correctly here, to handle different keywords for each event.

```
# Simple recursive-style countdown
@bot.on('privmsg')
@bot.on('countdown')
async def handle(target, message, remaining=None, **kwargs):
    # Entry point, verify command and parse from message
    if remaining is None:
        if not message.startswith("!countdown"):
            return
```

(continues on next page)

(continued from previous page)

```
        # !countdown 10
        remaining = int(message.split(" ")[-1])

    if remaining == 0:
        message = "Countdown complete!"
    else:
        message = "{}...".format(remaining)
    # Assume for now that target is always a channel
    bot.send("privmsg", target=target, message=message)

    if remaining:
        # After a second trigger another countdown event
        await asyncio.sleep(1, loop=bot.loop)
        bot.trigger('countdown', target=target,
                    message=message, remaining=remaining - 1)
```

3.3 Client.wait

```
await client.wait(event)
```

Wait for an event to trigger:

```
@bot.on("client_disconnect")
async def reconnect(**kwargs):
    # Trigger an event that may cascade to a client_connect.
    # Don't continue until a client_connect occurs,
    # which may be never.

    bot.trigger("some.plugin.connection.lost")

    await client.wait("client_connect")

    # If we get here, one of the plugins handled connection lost by
    # reconnecting, and we're back.  Send some messages, etc.
    client.send("privmsg", target=bot.CHANNEL,
                message="Happy Birthday!")
```

3.4 Client.connect

```
await client.connect()
```

Connect to the client's host, port.

```
@bot.on('client_disconnect')
async def reconnect(**kwargs):
    # Wait a few seconds
    await asyncio.sleep(3, loop=bot.loop)
    await bot.connect()
    # Now that we're connected, let everyone know
    bot.send('privmsg', target=bot.channel, message="I'm back.")
```

You can schedule a non-blocking connect with the client's event loop:

```
@bot.on('client_disconnect')
def reconnect(**kwargs):
    # Wait a few seconds

    # Note that we're not in a coroutine, so we don't have access
    # to await and asyncio.sleep
    time.sleep(3)

    # After this line we won't necessarily be connected.
    # We've simply scheduled the connect to happen in the future
    bot.loop.create_task(bot.connect())

    print("Reconnect scheduled.")
```

3.5 Client.disconnect

```
await client.disconnect()
```

Immediately disconnect from the server.

```
@bot.on('privmsg')
async def disconnect_bot(nick, message, **kwargs):
    if nick == "myNick" and message == "disconnect:hunter2":
        await bot.disconnect()
        logger.log("disconnected bot.")
```

Like `connect`, use the bot's event loop to schedule a disconnect:

```
bot.loop.create_task(bot.disconnect())
```

3.6 Client.send

```
client.send(command, **kwargs)
```

Send a command to the server. See Commands.

3.7 Client.handle_raw

New in version 2.1.0.

```
client.handle_raw(message)
```

Manually inject a raw command. The client's `raw_handlers` will process the message. By default, every `Client` is configured with a `rfc2812_handler` which unpacks a conforming rfc 2812 message into an event and calls `client.trigger`.

You can disable this functionality by removing the handler:

```
client = Client(host="localhost", port=443)
client.raw_handlers.clear()
```

3.8 Client.send_raw

New in version 2.1.0.

```
client.send_raw(message)
```

Send a complete IRC line without the Client reconstructing or modifying the message. To easily send an rfc 2812 message, you should instead consider `Client.send`.

CHAPTER

FOUR

EVENTS

In bottom, an event is simply a string and set of `**kwargs` to be passed to any handlers listening for that event:

```
@client.on('any string is fine')
def handle(**kwargs):
    if 'potato' in kwargs:
        print("Found a potato!")
```

4.1 IRC Events

While connected, a client will trigger events for valid IRC commands that it receives, with kwargs according to that command's structure. For example, the "part" event will always include the nickmask (`nick`, `user`, `host`), `message`, and `channel` kwargs, even if the message was empty:

```
@client.on("part")
def handle(nick, user, host, message, channel, **kwargs):
    out = "User {}!{}@{} left {} with '{}'"
    print(out.format(nick, user, host, channel, message))
```

Because `kwargs` contains those fields, we could also use:

```
@client.on("part")
def handle(**kwargs):
    out = ("User {nick}!{user}@{host} left"
           " {channel} with '{message}'")
    print(out.format(**kwargs))
```

4.2 Triggering

The same mechanism that the client uses to dispatch events can be invoked manually, either for custom events or to simulate receiving an irc command:

```
@client.on("privmsg")
def handle(**kwargs):
    print("Someone sent a message!")

client.trigger("privmsg")
```

Running the above won't print anything, however. Triggering an event only schedules the registered handlers (like the function we defined) to run *in the future*. Until we run the event loop, the triggered handlers won't be invoked. Let's see that print statement:

```
asyncio.get_event_loop().run_forever()
```

We can pass arbitrary kwargs to handlers through `trigger`:

```
client.trigger("event")
client.trigger("event", **some_dict)
client.trigger("event", nick="bot", message="hello, world")
```

4.3 Waiting

Sometimes we need to wait for another event to occur before continuing. For example, consider a reconnect handler that wants to trigger the "reconnect" event for some plugins, but only after the connection has actually been established. The following will incorrectly signal that the reconnect has completed, while in reality the client has only scheduled a connection for the future:

```
@client.on("client_disconnect")
def reconnect(**kwargs):
    client.connect()
    client.trigger("reconnect", reconnect_msg="May not be connected!")


@client.on("reconnect")
def handle_reconnect(reconnect_msg="", **kwargs):
    if reconnect_msg:
        client.send("privmsg", target=CHANNEL, message=reconnect_msg)
```

Because both `client.send` and `client.connect` schedule coroutines, the event loop may reorder (or process out of order). In `reconnect` what we really want to do is wait until the client_connect event is emitted, and then trigger the reconnect event:

```
@client.on("client_disconnect")
async def reconnect(**kwargs):
    client.connect()
    await client.wait("client_connect")
    client.trigger("reconnect", reconnect_msg="May not be connected!")
```

Whenever an event triggers, an `asyncio.Event` is set and cleared, which allows any code that is waiting on that event to continue. Be careful using `client.wait` - because we can call trigger with any string, `wait` will allow us to wait (forever) for events that may never trigger.

4.4 Supported Events

```
# Local only events
client.trigger('CLIENT_CONNECT')
client.trigger('CLIENT_DISCONNECT')
```

- PING
- JOIN
- PART
- PRIVMSG
- NOTICE
- USERMODE (renamed from MODE)
- CHANNELMODE (renamed from MODE)
- RPL_WELCOME (001)
- RPL_YOURHOST (002)
- RPL_CREATED (003)
- RPL_MYINFO (004)
- RPL_BOUNCE (005)
- RPL_MOTDSTART (375)
- RPL_MOTD (372)
- RPL_ENDOFMOTD (376)
- RPL_LUSERCLIENT (251)
- RPL_LUSERME (255)
- RPL_LUSEROP (252)
- RPL_LUSERUNKNOWN (253)
- RPL_LUSERCHANNELS (254)
- ERR_NOMOTD (422)

CHAPTER

FIVE

COMMANDS

```
client.send('PASS', password='hunter2')
```

```
client.send('NICK', nick='WiZ')
```

```
# mode is optional, default is 0
client.send('USER', user='WiZ-user', realname='Ronnie')
client.send('USER', user='WiZ-user', mode='8', realname='Ronnie')
```

```
client.send('OPER', user='WiZ', password='hunter2')
```

```
# Renamed from MODE
client.send('USERMODE', nick='WiZ')
client.send('USERMODE', nick='WiZ', modes='+io')
```

```
client.send('SERVICE', nick='CHANSERV', distribution='*.en',
            type='0', info='manages channels')
```

```
client.send('QUIT')
client.send('QUIT', message='Gone to Lunch')
```

```
client.send('SQUIT', server='tolsun.oulu.fi')
client.send('SQUIT', server='tolsun.oulu.fi', message='Bad Link')
```

```
client.send('JOIN', channel='#foo-chan')
client.send('JOIN', channel='#foo-chan', key='foo-key')
client.send('JOIN', channel=['#foo-chan', '#other'],
            key='foo-key') # other has no key
client.send('JOIN', channel=['#foo-chan', '#other'],
            key=['foo-key', 'other-key'])

# this will cause you to LEAVE all currently joined channels
client.send('JOIN', channel='0')
```

```
client.send('PART', channel='#foo-chan')
client.send('PART', channel=['#foo-chan', '#other'])
client.send('PART', channel='#foo-chan', message='I lost')
```

```
# Renamed from MODE
client.send('CHANNELMODE', channel='#foo-chan', modes='+b')
```

(continues on next page)

(continued from previous page)

```
client.send('CHANNELMODE', channel='#foo-chan', modes='+l',
            params='10')
```

```
client.send('TOPIC', channel='#foo-chan')
client.send('TOPIC', channel='#foo-chan',  # Clear channel message
            message='')
client.send('TOPIC', channel='#foo-chan',
            message='Yes, this is dog')
```

```
# target requires channel
client.send('NAMES')
client.send('NAMES', channel='#foo-chan')
client.send('NAMES', channel=['#foo-chan', '#other'])
client.send('NAMES', channel=['#foo-chan', '#other'],
            target='remote.*.edu')
```

```
# target requires channel
client.send('LIST')
client.send('LIST', channel='#foo-chan')
client.send('LIST', channel=['#foo-chan', '#other'])
client.send('LIST', channel=['#foo-chan', '#other'],
            target='remote.*.edu')
```

```
client.send('INVITE', nick='WiZ-friend', channel='#bar-chan')
```

```
# nick and channel must have the same number of elements
client.send('KICK', channel='#foo-chan', nick='WiZ')
client.send('KICK', channel='#foo-chan', nick='WiZ',
            message='Spamming')
client.send('KICK', channel='#foo-chan', nick=['WiZ', 'WiZ-friend'])
client.send('KICK', channel=['#foo', '#bar'],
            nick=['WiZ', 'WiZ-friend'])
```

```
client.send('PRIVMSG', target='WiZ-friend', message='Hello, friend!')
```

```
client.send('NOTICE', target='#foo-chan',
            message='Maintenance in 5 mins')
```

```
client.send('MOTD')
client.send('MOTD', target='remote.*.edu')
```

```
client.send('LUSERS')
client.send('LUSERS', mask='*.edu')
client.send('LUSERS', mask='*.edu', target='remote.*.edu')
```

```
client.send('VERSION')
```

```
# target requires query
client.send('STATS')
client.send('STATS', query='m')
client.send('STATS', query='m', target='remote.*.edu')
```

```
# remote requires mask
client.send('LINKS')
client.send('LINKS', mask='*.bu.edu')
client.send('LINKS', mask='*.bu.edu', remote='*.edu')
```

```
client.send('TIME')
client.send('TIME', target='remote.*.edu')
```

```
client.send('CONNECT', target='tolsun.oulu.fi', port=6667)
client.send('CONNECT', target='tolsun.oulu.fi', port=6667,
            remote='*.edu')
```

```
client.send('TRACE')
client.send('TRACE', target='remote.*.edu')
```

```
client.send('ADMIN')
client.send('ADMIN', target='remote.*.edu')
```

```
client.send('INFO')
client.send('INFO', target='remote.*.edu')
```

```
# type requires mask
client.send('SERVLIST', mask='*SERV')
client.send('SERVLIST', mask='*SERV', type=3)
```

```
client.send('SQUERY', target='irchelp', message='HELP privmsg')
```

```
client.send('WHO')
client.send('WHO', mask='*.fi')
client.send('WHO', mask='*.fi', o=True)
```

```
client.send('WHOIS', mask='*.fi')
client.send('WHOIS', mask=['*.fi', '*.edu'], target='remote.*.edu')
```

```
# target requires count
client.send('WHOWAS', nick='WiZ')
client.send('WHOWAS', nick='WiZ', count=10)
client.send('WHOWAS', nick=['WiZ', 'WiZ-friend'], count=10)
client.send('WHOWAS', nick='WiZ', count=10, target='remote.*.edu')
```

```
client.send('KILL', nick='WiZ', message='Spamming Joins')
```

```
# PING the server you are connected to
client.send('PING')
client.send('PING', message='Test..')
```

```
# when replying to a PING, the message should be the same
client.send('PONG')
client.send('PONG', message='Test..')
```

```
client.send('AWAY')
client.send('AWAY', message='Gone to Lunch')
```

```
client.send('REHASH')
```

```
client.send('DIE')
```

```
client.send('RESTART')
```

```
# target requires channel
client.send('SUMMON', nick='WiZ')
client.send('SUMMON', nick='WiZ', target='remote.*.edu')
client.send('SUMMON', nick='WiZ', target='remote.*.edu',
            channel='#foo-chan')
```

```
client.send('USERS')
client.send('USERS', target='remote.*.edu')
```

```
client.send('WALLOPS', message='Maintenance in 5 minutes')
```

```
client.send('USERHOST', nick='WiZ')
client.send('USERHOST', nick=['WiZ', 'WiZ-friend'])
```

```
client.send('ISON', nick='WiZ')
client.send('ISON', nick=['WiZ', 'WiZ-friend'])
```

CHAPTER

SIX

EXTENSIONS

bottom doesn't have any clever import hooks to identify plugins based on name, shape, or other significant denomination. Instead, we can create extensions by using `client.on` on a `Client` instance.

6.1 Keepalive

Instead of writing the same `PING` handler everywhere, a reusable plugin:

```
# my_plugin.py
def keepalive(client):
    @client.on("ping")
    def handle(message=None, **kwargs):
        message = message or ""
        client.send("pong", message=message)
```

That's it! And to use it:

```
import bottom
from my_plugin import keepalive

client = bottom.Client(...)
keepalive(client)
```

6.2 Returning new objects

Aside from subclassing `bottom.Client`, we can use a class to expose additional behavior around a client. This can be useful when we're worried about other plugins assigning different meaning to the same attributes:

```
# irc_logging.py
class Logger:
    def __init__(self, client, local_logger):
        self.client = client
        self.local = local_logger
        client.on("client_disconnect", self._on_disconnect)

    def log(self, level, message):
        try:
            self.client.send("{}: {}".format(level.upper(), message))
        catch RuntimeError:
            self.local.warning("Failed to log to remote")
```

(continues on next page)

(continued from previous page)

```
            # Get the local logger's method by name
            # ex. `self.local.info`
            method = getattr(self.local, level.lower())
            method(message)

    def _on_disconnect(self):
        self.local.warning("Remote logging client disconnected!")


    def debug(self, message):
        self.log("debug", message)

    # Same for info, warning, ...
    ...
```

And its usage:

```
import asyncio
import bottom
import logging
from irc_logging import Logger

local_logger = logging.getLogger(__name__)

client = bottom.Client(...)
remote_logger = Logger(client, local_logger)


@client.on("client_connect")
def log_connect(**kwargs):
    remote_logger.info("Connected!")

# Connect and send "INFO: Connected!"
asyncio.create_task(client.connect())
asyncio.get_event_loop().run_forever()
```

Notice that the logging functionality is part of a different object, not the client. This keeps the namespace clean, and reduces the attribute contention that can occur when multiple plugins store their information directly on the client instance.

This line hooked the logger's disconnect handler to the client:

```
def __init__(self, client, ...):
    ...
    client.on("client_disconnect", self._on_disconnect)
```

6.3 Pattern matching

We can write a simple wrapper class to annotate functions to handle PRIVMSG matching a regex. To keep the interface simple, we can use bottom's annotation pattern and pass the regex to match.

In the following example, we'll define a handler that echos whatever a user asks for, if it's in the correct format:

```
import bottom

client = bottom.Client(host=host, port=port, ssl=ssl)
router = Router(client)


@router.route("^bot, say (\w+)\.$")
def echo(self, nick, target, message, match, **kwargs):
    if target == router.nick:
        # respond in a direct message
        target = nick
    client.send("privmsg", target=target, message=match.group(1))
```

Now, the Router class needs to manage the regex -> handler mapping and connect an event handler to PRIVMSG on its client:

```
import asyncio
import functools
import re


class Router(object):
    def __init__(self, client):
        self.client = client
        self.routes = {}
        client.on("PRIVMSG")(self._handle)

    def _handle(self, nick, target, message, **kwargs):
        """ client callback entrance """
        for regex, (func, pattern) in self.routes.items():
            match = regex.match(message)
            if match:
                self.client.loop.create_task(func(nick, target, message, match,␣
↪**kwargs))

    def route(self, pattern, func=None, **kwargs):
        if func is None:
            return functools.partial(self.route, pattern)

        # Decorator should always return the original function
        wrapped = func
        if not asyncio.iscoroutinefunction(wrapped):
            wrapped = asyncio.coroutine(wrapped)

        compiled = re.compile(pattern)
        self.routes[compiled] = (wrapped, pattern)
        return func
```

6.4 Wait for any events

Use `Client.wait()` to pause until one or all signals have fired. For example, after sending NICK/USER during CLIENT_CONNECT, some servers will ignore subsequent commands until they have finished sending RPL_ENDOFMOTD. This can be used to wait for any signal that the MOTD has been sent (eg. ERR_NOMOTD may be sent instead of RPL_ENDOFMOTD).

```
import asyncio


def waiter(client):
    async def wait_for(*events, return_when=asyncio.FIRST_COMPLETED):
        if not events:
            return
        done, pending = await asyncio.wait(
            [bot.wait(event) for event in events],
            return_when=return_when)

        # Cancel any events that didn't come in.
        for future in pending:
            future.cancel()
    return wait_for
```

To use in the CLIENT_CONNECT process:

```
import bottom
client = bottom.Client(...)
wait_for = waiter(client)


@client.on("CLIENT_CONNECT")
async def on_connect(**kwargs):
    client.send('nick', ...)
    client.send('user', ...)

    await wait_for('RPL_ENDOFMOTD', 'ERR_NOMOTD')

    client.send('join', ...)
```

6.5 Send and trigger raw messages

New in version 2.1.0.

Extensions do not need to strictly conform to rfc 2812. You can send or trigger custom messages with `Client.send_raw` and `Client.handle_raw`. For example, the following can be used to request Twitch.tv's Membership capability using IRC v3's capabilities registration:

```
client = MyTwitchClient(...)
client.send_raw("CAP REQ :twitch.tv/membership")
```

Just as `Client.trigger` can be used to manually invoke handlers for a specific event, `Client.handle_raw` can be called to manually invoke raw handlers for a given message. For the above example, you can ensure you handle the response from Twitch.tv with the following:

```
response = ":tmi.twitch.tv CAP * ACK :twitch.tv/membership"
client = MyTwitchClient(...)
client.handle_raw(response)
```

6.6 Raw handlers

New in version 2.1.0.

Clients can extend or replace the default message handler by modifying the `Client.raw_handlers` list. This is a list of async functions that take a `(next_handler, message)` tuple. To allow the next handler to process a message, call `next_handler(message)` within your handler. You may also send a different message to the subsequent handler, or not invoke it at all.

The following listens for responses from twitch.tv about capabilities and logs them. Otherwise, it passes the message on to the next handler.

```
import re
CAPABILITY_RESPONSE_PATTERN = re.compile(
    "^:tmi\.twitch\.tv CAP \* ACK :twitch\.tv/\w+$")


async def capability_handler(next_handler, message):
    if CAPABILITY_RESPONSE_PATTERN.match(message):
        print("Capability granted: " + message)
    else:
        await next_handler(message)
```

And to ensure it runs before the default handler:

```
client = Client(...)
client.raw_handlers.insert(0, capability_handler)
```

Unlike `Client.on`, raw handlers must be async functions.

Handlers may send a different message than they receive. The following can be used to forward messages from one chat room to another:

```
from bottom.pack import pack_command
from bottom.unpack import unpack_command


def forward(old_room, new_room):
    async def handle(next_handler, message):
        try:
            event, kwargs = unpack_command(message)
        except ValueError:
            # pass message unchanged
            pass
        else:
            if event.lower() == "privmsg":
                if kwargs["target"].lower() == old_room.lower():
                    kwargs["target"] = new_room
                    message = pack_command("privmsg", **kwargs)
        await next_handler(message)
    return handle
```

And its usage:

```
client = Client(...)

forwarding = forward("bottom-legacy", "bottom-dev")
client.raw_handlers.insert(0, forwarding)
```

6.7 Full message encryption

This is a more complex example of a raw handler where messages are encrypted and then base64 encoded. On the wire their only similarity with the IRC protocol is a newline terminating character. This is enough to build an extension to transparently encrypt data.

Assume you have implemented a class with the following interface:

```
class EncryptionContext:
    def encrypt(self, data: bytes) -> bytes:
        ...

    def decrypt(self, data: bytes) -> bytes:
        ...
```

the following extension can be written:

```
import base64

def encryption_handler(context: EncryptionContext):
    async def handle_decrypt(next_handler, message):
        message = context.decrypt(
            base64.b64decode(
                message.encode("utf-8")
            )
        ).decode("utf-8")
        await next_handler(message)
    return handle_decrypt
```

to encrypt messages as they are sent, the class can override `Client.send_raw`. Adding in the encryption handler above:

```
class EncryptedClient(Client):
    def __init__(self, encryption_context, **kwargs):
        super().__init__(**kwargs)
        self.raw_handlers.append(
            encryption_handler(encryption_context))
        self.context = encryption_context

    def send_raw(self, message: str) -> None:
        message = base64.b64encode(
            self.context.encrypt(
                message.encode("utf-8")
            )
        ).decode("utf-8")
        super().send_raw(message)
```

CHAPTER

SEVEN

DEVELOPMENT

7.1 Versioning and RFC2812

- Bottom follows semver for its **public** API.
  - Currently, `Client` is the only public member of bottom.
  - IRC replies/codes which are not yet implemented may be added at any time, and will correspond to a patch - the function contract of `@on` method does not change.
  - You should not rely on the internal api staying the same between minor versions.
  - Over time, private apis may be raised to become public. The reverse will never occur.

7.2 Contributing

Contributions welcome! Please make sure `tox` passes (including flake8 and docs build) before submitting a PR.

Pull requests that decrease coverage will not be merged.

7.2.1 Development

bottom uses `tox`, `pytest`, `coverage`, and `flake8`. To get everything set up in a new virtualenv:

```
git clone https://github.com/numberoverzero/bottom.git
cd bottom
python3.10 -m venv --copies .venv
source .venv/bin/activate
pip install -r requirements.txt
pip install -e .
tox
```

7.2.2 Documentation

Documentation improvements are especially appreciated. For small changes, open a pull request! If there's an area you feel is lacking and will require more than a small change, open an issue to discuss the problem - others are probably also confused, and may have suggestions to improve the same area.

7.3 TODO

- Better `Client` docstrings
- Add missing replies/errors to `unpack.py:unpack_command`
  - Add reply/error parameters to `unpack.py:parameters`
  - Document events, client.send