18.7. `asynchat` — Asynchronous socket command/response handler

Source code: Lib/asynchat.py

Deprecated since version 3.6: Please use `asyncio` instead.

> **Note:** This module exists for backwards compatibility only. For new code we recommend using `asyncio`.

This module builds on the `asyncore` infrastructure, simplifying asynchronous clients and servers and making it easier to handle protocols whose elements are terminated by arbitrary strings, or are of variable length. `asynchat` defines the abstract class `async_chat` that you subclass, providing implementations of the `collect_incoming_data()` and `found_terminator()` methods. It uses the same asynchronous loop as `asyncore`, and the two types of channel, `asyncore.dispatcher` and `asynchat.async_chat`, can freely be mixed in the channel map. Typically an `asyncore.dispatcher` server channel generates new `asynchat.async_chat` channel objects as it receives incoming connection requests.

class `asynchat.`**`async_chat`**

> This class is an abstract subclass of `asyncore.dispatcher`. To make practical use of the code you must subclass `async_chat`, providing meaningful `collect_incoming_data()` and `found_terminator()` methods. The `asyncore.dispatcher` methods can be used, although not all make sense in a message/response context.
>
> Like `asyncore.dispatcher`, `async_chat` defines a set of events that are generated by an analysis of socket conditions after a `select()` call. Once the polling loop has been started the `async_chat` object's methods are called by the event-processing framework with no action on the part of the programmer.
>
> Two class attributes can be modified, to improve performance, or possibly even to conserve memory.

> **`ac_in_buffer_size`**
>
> > The asynchronous input buffer size (default `4096`).

> **`ac_out_buffer_size`**
>
> > The asynchronous output buffer size (default `4096`).

Unlike `asyncore.dispatcher`, `async_chat` allows you to define a FIFO queue of *producers*. A producer need have only one method, `more()`, which should return data to be transmitted on the channel. The producer indicates exhaustion (*i.e.* that it contains no more data) by having its `more()` method return the empty bytes object. At this point the `async_chat` object removes the producer from the queue and starts using the next producer, if any. When the producer queue is empty the `handle_write()` method does nothing. You use the channel object's `set_terminator()` method to describe how to recognize the end of, or an important breakpoint in, an incoming transmission from the remote endpoint.

To build a functioning `async_chat` subclass your input methods `collect_incoming_data()` and `found_terminator()` must handle the data that the channel receives asynchronously. The methods are described below.

async_chat. **close_when_done**()

Pushes a `None` on to the producer queue. When this producer is popped off the queue it causes the channel to be closed.

async_chat. **collect_incoming_data**(*data*)

Called with *data* holding an arbitrary amount of received data. The default method, which must be overridden, raises a `NotImplementedError` exception.

async_chat. **discard_buffers**()

In emergencies this method will discard any data held in the input and/or output buffers and the producer queue.

async_chat. **found_terminator**()

Called when the incoming data stream matches the termination condition set by `set_terminator()`. The default method, which must be overridden, raises a `NotImplementedError` exception. The buffered input data should be available via an instance attribute.

async_chat. **get_terminator**()

Returns the current terminator for the channel.

async_chat. **push**(*data*)

Pushes data on to the channel's queue to ensure its transmission. This is all you need to do to have the channel write the data out to the network, although it is possible to use your own producers in more complex schemes to implement encryption and chunking, for example.

async_chat. **push_with_producer**(*producer*)

Takes a producer object and adds it to the producer queue associated with the channel. When all currently-pushed producers have been exhausted the chan-

nel will consume this producer's data by calling its `more()` method and send the data to the remote endpoint.

`async_chat.` **`set_terminator`**(*term*)

Sets the terminating condition to be recognized on the channel. `term` may be any of three types of value, corresponding to three different ways to handle incoming protocol data.

term	Description
string	Will call `found_terminator()` when the string is found in the input stream
integer	Will call `found_terminator()` when the indicated number of characters have been received
None	The channel continues to collect data forever

Note that any data following the terminator will be available for reading by the channel after `found_terminator()` is called.

18.7.1. asynchat Example

The following partial example shows how HTTP requests can be read with `async_chat`. A web server might create an `http_request_handler` object for each incoming client connection. Notice that initially the channel terminator is set to match the blank line at the end of the HTTP headers, and a flag indicates that the headers are being read.

Once the headers have been read, if the request is of type POST (indicating that further data are present in the input stream) then the `Content-Length:` header is used to set a numeric terminator to read the right amount of data from the channel.

The `handle_request()` method is called once all relevant input has been marshalled, after setting the channel terminator to `None` to ensure that any extraneous data sent by the web client are ignored.

```python
import asynchat

class http_request_handler(asynchat.async_chat):

    def __init__(self, sock, addr, sessions, log):
        asynchat.async_chat.__init__(self, sock=sock)
        self.addr = addr
        self.sessions = sessions
        self.ibuffer = []
        self.obuffer = b""
```

```python
        self.set_terminator(b"\r\n\r\n")
        self.reading_headers = True
        self.handling = False
        self.cgi_data = None
        self.log = log

    def collect_incoming_data(self, data):
        """Buffer the data"""
        self.ibuffer.append(data)

    def found_terminator(self):
        if self.reading_headers:
            self.reading_headers = False
            self.parse_headers(b"".join(self.ibuffer))
            self.ibuffer = []
            if self.op.upper() == b"POST":
                clen = self.headers.getheader("content-length")
                self.set_terminator(int(clen))
            else:
                self.handling = True
                self.set_terminator(None)
                self.handle_request()
        elif not self.handling:
            self.set_terminator(None)  # browsers sometimes over-send
            self.cgi_data = parse(self.headers, b"".join(self.ibuffer)
            self.handling = True
            self.ibuffer = []
            self.handle_request()
```