:mod:`asyncore` --- Asynchronous socket handler

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst, line 1); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst, line 4)

Unknown directive type "module".

```
.. module:: asyncore
   :synopsis: A base class for developing asynchronous socket handling
           services.
   :deprecated:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst, line 9)

Unknown directive type "moduleauthor".

```
.. moduleauthor:: Sam Rushing <rushing@nightmare.com>
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst, line 10)

Unknown directive type "sectionauthor".

```
.. sectionauthor:: Christopher Petrilli <petrilli@amber.org>
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst, line 11)

Unknown directive type "sectionauthor".

```
.. sectionauthor:: Steve Holden <sholden@holdenweb.com>
```

Source code: :source:`Lib/asyncore.py`

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst, line 14); *backlink*

Unknown interpreted text role "source".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst, line 16)

Unknown directive type "deprecated".

```
.. deprecated:: 3.6
   :mod:`asyncore` will be removed in Python 3.12 (:pep:`594`).
   Please use :mod:`asyncio` instead.
```

Note

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst, line 24); *backlink*

Unknown interpreted text role "mod".

This module provides the basic infrastructure for writing asynchronous socket service clients and servers.

There are only two ways to have a program on a single processor do "more than one thing at a time." Multi-threaded programming is the simplest and most popular way to do it, but there is another very different technique, that lets you have nearly all the advantages of multi-threading, without actually using multiple threads. It's really only practical if your program is largely I/O bound. If your program is processor bound, then pre-emptive scheduled threads are probably what you really need. Network servers are rarely processor bound, however.

If your operating system supports the :c:func:`select` system call in its I/O library (and nearly all do), then you can use it to juggle multiple communication channels at once; doing other work while your I/O is taking place in the "background." Although this strategy can seem strange and complex, especially at first, it is in many ways easier to understand and control than multi-threaded programming. The :mod:`asyncore` module solves many of the difficult problems for you, making the task of building sophisticated high-performance network servers and clients a snap. For "conversational" applications and protocols the companion :mod:`asynchat` module is invaluable.

The basic idea behind both modules is to create one or more network *channels*, instances of class :class:`asyncore.dispatcher` and :class:`asynchat.async_chat`. Creating the channels adds them to a global map, used by the :func:`loop` function if you do not provide it with your own *map*.

Once the initial channel(s) is(are) created, calling the :func:`loop` function activates channel service, which continues until the last channel (including any that have been added to the map during asynchronous service) is closed.

```
channels have been closed.  All arguments are optional.  The *count*
parameter defaults to ``None``, resulting in the loop terminating only when all
channels have been closed.  The *timeout* argument sets the timeout
parameter for the appropriate :func:`~select.select` or :func:`~select.poll`
call, measured in seconds; the default is 30 seconds.  The *use_poll*
parameter, if true, indicates that :func:`~select.poll` should be used in
preference to :func:`~select.select` (the default is ``False``).

The *map* parameter is a dictionary whose items are the channels to watch.
As channels are closed they are deleted from their map.  If *map* is
omitted, a global map is used. Channels (instances of
:class:`asyncore.dispatcher`, :class:`asynchat.async_chat` and subclasses
thereof) can freely be mixed in the map.
```

The :class:`dispatcher` class is a thin wrapper around a low-level socket object. To make it more useful, it has a few methods for event-handling which are called from the asynchronous loop. Otherwise, it can be treated as a normal non-blocking socket object.

BWTgdgdg

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst, **line 81**); *backlink*

Unknown interpreted text role "class".

The firing of low-level events at certain times or in certain connection states tells the asynchronous loop that certain higher-level events have taken place. For example, if we have asked for a socket to connect to another host, we know that the connection has been made when the socket becomes writable for the first time (at this point you know that you may write to it with the expectation of success). The implied higher-level events are:

Event	Description
handle_connect()	Implied by the first read or write event
handle_close()	Implied by a read event with no data available
handle_accepted()	Implied by a read event on a listening socket

During asynchronous processing, each mapped channel's :meth:`readable` and :meth:`writable` methods are used to determine whether the channel's socket should be added to the list of channels :c:func:`select`ed or :c:func:`poll`ed for read and write events.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst, **line 107**); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst, **line 107**); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst, **line 107**); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst, **line 107**); *backlink*

Unknown interpreted text role "c:func".

Thus, the set of channel events is larger than the basic socket events. The full set of methods that can be overridden in your subclass follows:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst, **line 116**)

Unknown directive type "method".

```
.. method:: handle_read()

   Called when the asynchronous loop detects that a :meth:`read` call on the
   channel's socket will succeed.
```

```
.. method:: handle_write()

   Called when the asynchronous loop detects that a writable socket can be
   written.  Often this method will implement the necessary buffering for
   performance.  For example::

      def handle_write(self):
          sent = self.send(self.buffer)
          self.buffer = self.buffer[sent:]
```

```
.. method:: handle_expt()

   Called when there is out of band (OOB) data for a socket connection.  This
   will almost never happen, as OOB is tenuously supported and rarely used.
```

```
.. method:: handle_connect()

   Called when the active opener's socket actually makes a connection.  Might
   send a "welcome" banner, or initiate a protocol negotiation with the
   remote endpoint, for example.
```

```
.. method:: handle_close()

   Called when the socket is closed.
```

```
.. method:: handle_error()

   Called when an exception is raised and not otherwise handled.  The default
   version prints a condensed traceback.
```

```
.. method:: handle_accept()
```

Called on listening channels (passive openers) when a connection can be
established with a new remote endpoint that has issued a :meth:`connect`
call for the local endpoint. Deprecated in version 3.2; use
:meth:`handle_accepted` instead.

.. deprecated:: 3.2

Unknown directive type "method".

```
.. method:: handle_accepted(sock, addr)

    Called on listening channels (passive openers) when a connection has been
    established with a new remote endpoint that has issued a :meth:`connect`
    call for the local endpoint.  *sock* is a *new* socket object usable to
    send and receive data on the connection, and *addr* is the address
    bound to the socket on the other end of the connection.

    .. versionadded:: 3.2
```

Unknown directive type "method".

```
.. method:: readable()

    Called each time around the asynchronous loop to determine whether a
    channel's socket should be added to the list on which read events can
    occur.  The default method simply returns ``True``, indicating that by
    default, all channels will be interested in read events.
```

Unknown directive type "method".

```
.. method:: writable()

    Called each time around the asynchronous loop to determine whether a
    channel's socket should be added to the list on which write events can
    occur.  The default method simply returns ``True``, indicating that by
    default, all channels will be interested in write events.
```

In addition, each channel delegates or extends many of the socket methods. Most of these are nearly identical to their socket partners.

Unknown directive type "method".

```
.. method:: create_socket(family=socket.AF_INET, type=socket.SOCK_STREAM)

    This is identical to the creation of a normal socket, and will use the
    same options for creation.  Refer to the :mod:`socket` documentation for
    information on creating sockets.

    .. versionchanged:: 3.3
       *family* and *type* arguments can be omitted.
```

Unknown directive type "method".

```
.. method:: connect(address)

   As with the normal socket object, *address* is a tuple with the first
   element the host to connect to, and the second the port number.
```

Unknown directive type "method".

```
.. method:: send(data)

   Send *data* to the remote end-point of the socket.
```

Unknown directive type "method".

```
.. method:: recv(buffer_size)

   Read at most *buffer_size* bytes from the socket's remote end-point.  An
   empty bytes object implies that the channel has been closed from the
   other end.

   Note that :meth:`recv` may raise :exc:`BlockingIOError` , even though
   :func:`select.select` or :func:`select.poll` has reported the socket
   ready for reading.
```

Unknown directive type "method".

```
.. method:: listen(backlog)

   Listen for connections made to the socket.  The *backlog* argument
   specifies the maximum number of queued connections and should be at least
   1; the maximum value is system-dependent (usually 5).
```

Unknown directive type "method".

```
.. method:: bind(address)

   Bind the socket to *address*.  The socket must not already be bound.  (The
   format of *address* depends on the address family --- refer to the
   :mod:`socket` documentation for more information.)  To mark
   the socket as re-usable (setting the :const:`SO_REUSEADDR` option), call
   the :class:`dispatcher` object's :meth:`set_reuse_addr` method.
```

Unknown directive type "method".

```
.. method:: accept()

   Accept a connection.  The socket must be bound to an address and listening
   for connections.  The return value can be either ``None`` or a pair
   ``(conn, address)`` where *conn* is a *new* socket object usable to send
   and receive data on the connection, and *address* is the address bound to
```

```
        the socket on the other end of the connection.
        When ``None`` is returned it means the connection didn't take place, in
        which case the server should just ignore this event and keep listening
        for further incoming connections.
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst`, line 258)

Unknown directive type "method".

```
    .. method:: close()

        Close the socket.  All future operations on the socket object will fail.
        The remote end-point will receive no more data (after queued data is
        flushed).  Sockets are automatically closed when they are
        garbage-collected.
```

A :class:`dispatcher` subclass which adds simple buffered output capability, useful for simple clients. For more sophisticated usage use :class:`asynchat.async_chat`.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst`, line 268); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst`, line 268); *backlink*

Unknown interpreted text role "class".

A file_dispatcher takes a file descriptor or :term:`file object` along with an optional map argument and wraps it for use with the :c:func:`poll` or :c:func:`loop` functions. If provided a file object or anything with a :c:func:`fileno` method, that method will be called and passed to the :class:`file_wrapper` constructor.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst`, line 274); *backlink*

Unknown interpreted text role "term".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst`, line 274); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst`, line 274); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst`, line 274); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst`, line 274); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst`, line 280)

Unknown directive type "availability".

```
    .. availability:: Unix.
```

A file_wrapper takes an integer file descriptor and calls :func:`os.dup` to duplicate the handle so that the original handle may be closed independently of the file_wrapper. This class implements sufficient methods to emulate a socket for use by the :class:`file_dispatcher` class.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst`, **line 284**); *backlink*
>
> Unknown interpreted text role "func".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst`, **line 284**); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst`, **line 289**)
>
> Unknown directive type "availability".
>
> ```
> .. availability:: Unix.
> ```

asyncore Example basic HTTP client

Here is a very basic HTTP client that uses the :class:`dispatcher` class to implement its socket handling:

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst`, **line 297**); *backlink*
>
> Unknown interpreted text role "class".

```
import asyncore

class HTTPClient(asyncore.dispatcher):

    def __init__(self, host, path):
        asyncore.dispatcher.__init__(self)
        self.create_socket()
        self.connect( (host, 80) )
        self.buffer = bytes('GET %s HTTP/1.0\r\nHost: %s\r\n\r\n' %
                            (path, host), 'ascii')

    def handle_connect(self):
        pass

    def handle_close(self):
        self.close()

    def handle_read(self):
        print(self.recv(8192))

    def writable(self):
        return (len(self.buffer) > 0)

    def handle_write(self):
        sent = self.send(self.buffer)
        self.buffer = self.buffer[sent:]


client = HTTPClient('www.python.org', '/')
asyncore.loop()
```

asyncore Example basic echo server

Here is a basic echo server that uses the :class:`dispatcher` class to accept connections and dispatches the incoming connections to a handler:

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncore.rst`, **line 336**); *backlink*
>
> Unknown interpreted text role "class".

```python
import asyncore

class EchoHandler(asyncore.dispatcher_with_send):

    def handle_read(self):
        data = self.recv(8192)
        if data:
            self.send(data)

class EchoServer(asyncore.dispatcher):

    def __init__(self, host, port):
        asyncore.dispatcher.__init__(self)
        self.create_socket()
        self.set_reuse_addr()
        self.bind((host, port))
        self.listen(5)

    def handle_accepted(self, sock, addr):
        print('Incoming connection from %s' % repr(addr))
        handler = EchoHandler(sock)

server = EchoServer('localhost', 8080)
asyncore.loop()
```