# Contents

## 1 Introduction
- 1.1 Quick Example ................................................. 3
- 1.2 Command-line .................................................. 4
- 1.3 Features ....................................................... 5
- 1.4 Further Reading ............................................... 5

## 2 API
- 2.1 accessories .................................................. 7
- 2.2 client ......................................................... 8
- 2.3 client_base .................................................. 13
- 2.4 client_shell .................................................. 15
- 2.5 server ......................................................... 15
- 2.6 server_base .................................................. 19
- 2.7 server_shell .................................................. 21
- 2.8 sle ......................................................... 21
- 2.9 stream_reader ............................................... 24
- 2.10 stream_writer ............................................. 27
- 2.11 telopt ....................................................... 43

## 3 RFCs
- 3.1 Implemented .................................................. 45
- 3.2 Not Implemented ............................................. 46
- 3.3 Additional Resources ......................................... 46

## 4 Contributing
- 4.1 Developing .................................................. 49
- 4.2 Running Tests ............................................... 49
- 4.3 Code Formatting ............................................. 50
- 4.4 Style and Static Analysis ................................... 50

## 5 History ......................................................... 51

## 6 Indexes ......................................................... 53

**Python Module Index** ......................................... 55

**Index** ......................................................... 57
Python 3 asyncio Telnet server and client Protocol library.

Contents:
telnetlib3 is a Telnet Client and Server library for python. This project requires python 3.7 and later, using the asyncio module.

1.1 Quick Example

Authoring a Telnet Server using Streams interface that offers a basic war game:

```python
import asyncio, telnetlib3

async def shell(reader, writer):
    writer.write('
Would you like to play a game? '
    inp = await reader.read(1)
    if inp:
        writer.echo(inp)
        writer.write('
They say the only way to win '
        'is to not play at all.

    await writer.drain()
    writer.close()

loop = asyncio.get_event_loop()
coro = telnetlib3.create_server(port=6023, shell=shell)
server = loop.run_until_complete(coro)
loop.run_until_complete(server.wait_closed())
```

Authoring a Telnet Client that plays the war game with this server:

```python
import asyncio, telnetlib3

async def shell(reader, writer):
    while True:
        # read stream until '?' mark is found
        outp = await reader.read(1024)
```

(continues on next page)
1.2 Command-line

Two command-line scripts are distributed with this package.

telnetlib3-client

Small terminal telnet client. Some example destinations and options:

telnetlib3-client nethack.alt.org
telnetlib3-client --encoding=cp437 --force-binary blackflag.acid.org
telnetlib3-client htc.zapto.org


telnetlib3-server

Telnet server providing the default debugging shell. This provides a simple shell server that allows introspection of the session’s values, for example:

tel:sh> help
quit, writer, slc, toggle [option|all], reader, proto

tel:sh> writer
<TelnetWriter server mode:kludge +lineflow -xon_any +slc_sim server-
˓→will:BINARY,ECHO,SGA client-will:BINARY,NAWS,NEW_ENVIRON,TTYPE>

tel:sh> reader
<TelnetReaderUnicode encoding='utf8' limit=65536 buflen=0 eof=False>

tel:sh> toggle all
wont echo.
wont suppress go-ahead.
wont outbinary.
dont inbinary.
xon-any enabled.
lineflow disabled.

tel:sh> reader
<TelnetReaderUnicode encoding='US-ASCII' limit=65536 buflen=1 eof=False>
Both command-line scripts accept argument `--shell=my_module.fn_shell` describing a python module path to a coroutine of signature `shell(reader, writer)`, just as the above examples.

### 1.3 Features

The following RFC specifications are implemented:


### 1.4 Further Reading

Further documentation available at https://telnetlib3.readthedocs.org/
2.1 accessories

Accessory functions.

**encoding_from_lang**(lang)
Parse encoding from LANG environment value.

Example:

```python
>>> encoding_from_lang('en_US.UTF-8@misc')
'UTF-8'
```

**name_unicode**(ucs)
Return 7-bit ascii printable of any string.

**eightbits**(number)
Binary representation of number padded to 8 bits.

Example:

```python
>>> eightbits(ord('a'))
'0b01100001'
```

**make_logger**(name, loglevel='info', logfile=None, logfmt='%(asctime)s %(levelname)s %(filename)s:%(lineno)d %(message)s')
Create and return simple logger for given arguments.

**repr_mapping**(mapping)
Return printable string, ‘key=value [key=value …]’ for mapping.

**function_lookup**(pymod_path)
Return callable function target from standard module.function path.

**make_reader_task**(reader, size=4096)
Return asyncio task wrapping coroutine of reader.read(size).
2.2 client

Telnet Client API for the `telnetlib3` python package.

class TelnetClient (term='unknown', cols=80, rows=25, tspeed=(38400, 38400), xdisploc='", *args, **kwargs)

Bases: telnetlib3.client_base.BaseClient

Telnet client that supports all common options.

This class is useful for automation, it appears to be a virtual terminal to the remote end, but does not require an interactive terminal to run.

DEFAULT_LOCALE = 'en_US'

On `send_env()`, the value of ‘LANG’ will be ‘C’ for binary transmission. When encoding is specified (utf8 by default), the LANG variable must also contain a locale, this value is used, providing a full default LANG value of ‘en_US.utf8’

connection_made (transport)

Callback for connection made to server.

send_ttype ()

Callback for responding to TTYPE requests.

send_tspeed ()

Callback for responding to TSPEED requests.

send_xdisploc ()

Callback for responding to XDISPLOC requests.

send_env (keys)

Callback for responding to NEW_ENVIRON requests.

Parameters keys (dict) – Values are requested for the keys specified. When empty, all environment values that wish to be volunteered should be returned.

Returns dictionary of environment values requested, or an empty string for keys not available.

A return value must be given for each key requested.

Return type dict

send_charset (offered)

Callback for responding to CHARSET requests.

Receives a list of character encodings offered by the server as `offered` such as ('LATIN-1', 'UTF-8'), for which the client may return a value agreed to use, or None to disagree to any available offers. Server offerings may be encodings or codepages.

The default implementation selects any matching encoding that python is capable of using, preferring any that matches `encoding` if matched in the offered list.

Parameters offered (list) – list of CHARSET options offered by server.

Returns character encoding agreed to be used.

Return type str or None

send_naws ()

Callback for responding to NAWS requests.

Return type (int, int)

Returns client window size as (rows, columns).
encoding \((\text{outgoing}=\text{None}, \text{incoming}=\text{None})\)

Return encoding for the given stream direction.

**Parameters**

- **outgoing** \((\text{bool})\) – Whether the return value is suitable for encoding bytes for transmission to server.
- **incoming** \((\text{bool})\) – Whether the return value is suitable for decoding bytes received by the client.

**Raises** **TypeError** – when a direction argument, either outgoing or incoming, was not set True.

**Returns** ‘US-ASCII’ for the directions indicated, unless BINARY RFC 856 has been negotiated for the direction indicated or :attr:`force_binary` is set True.

**Return type** str

**begin_negotiation**()

Begin on-connect negotiation.

A Telnet client is expected to send only a minimal amount of client session options immediately after connection, it is generally the server which dictates server option support.

Deriving implementations should always call super().begin_negotiation().

**begin_shell** \((\text{result})\)

**check_negotiation** \((\text{final}=\text{False})\)

Callback, return whether negotiation is complete.

**Parameters** **final** \((\text{bool})\) – Whether this is the final time this callback will be requested to answer regarding protocol negotiation.

**Returns** Whether negotiation is over (client end is satisfied).

**Return type** bool

Method is called on each new command byte processed until negotiation is considered final, or after connect_maxwait has elapsed, setting the _waiter_connected attribute to value self when complete.

This method returns False until connect_minwait has elapsed, ensuring the server may batch telnet negotiation demands without prematurely entering the callback shell.

Ensure super().check_negotiation() is called and conditionally combined when derived.

**connection_lost** \((\text{exc})\)

Called when the connection is lost or closed.

**Parameters** **exc** \((\text{Exception})\) – exception. None indicates a closing EOF sent by this end.

**data_received** \((\text{data})\)

Process bytes received by transport.

**duration**

Time elapsed since client connected, in seconds as float.

**eof_received**()

Called when the other end calls write_eof() or equivalent.

**get_extra_info** \((\text{name}, \text{default}=\text{None})\)

Get optional client protocol or transport information.
idle
Time elapsed since data last received, in seconds as float.

pause_writing()
Called when the transport’s buffer goes over the high-water mark.

Pause and resume calls are paired – pause_writing() is called once when the buffer goes strictly over
the high-water mark (even if subsequent writes increases the buffer size even more), and eventually re-
sume_writing() is called once when the buffer size reaches the low-water mark.

Note that if the buffer size equals the high-water mark, pause_writing() is not called – it must go strictly
over. Conversely, resume_writing() is called when the buffer size is equal or lower than the low-water
mark. These end conditions are important to ensure that things go as expected when either mark is zero.

NOTE: This is the only Protocol callback that is not called through EventLoop.call_soon() – if it were,
it would have no effect when it’s most needed (when the app keeps writing without yielding until
pause_writing() is called).

resume_writing()
Called when the transport’s buffer drains below the low-water mark.

See pause_writing() for details.

class TelnetTerminalClient (term='unknown', cols=80, rows=25, tspeed=(38400, 38400), xdis-
proc='*', *args, **kwargs)
Bases: telnetlib3.client.TelnetClient
Telnet client for sessions with a network virtual terminal (NVT).

send_naws ()
Callback reply for request with window size, NAWS RFC 1073.

Return type  (int, int)
Returns  window dimensions by lines and columns

send_env (keys)
Callback reply for request with env values, NEW_ENVIRON RFC 1572.

Return type  dict
Returns  super class value updated with window LINES and COLUMNS.

DEFAULT_LOCALE = 'en_US'

begin_negotiation ()
Begin on-connect negotiation.

A Telnet client is expected to send only a minimal amount of client session options immediately after
connection, it is generally the server which dictates server option support.

Deriving implementations should always call super().begin_negotiation().

begin_shell (result)

check_negotiation (final=False)
Callback, return whether negotiation is complete.

Parameters final (bool) – Whether this is the final time this callback will be requested to
answer regarding protocol negotiation.

Returns  Whether negotiation is over (client end is satisfied).

Return type  bool
Method is called on each new command byte processed until negotiation is considered final, or after connect_maxwait has elapsed, setting the _waiter_connected attribute to value self when complete.

This method returns False until connect_minwait has elapsed, ensuring the server may batch telnet negotiation demands without prematurely entering the callback shell.

Ensure super().check_negotiation() is called and conditionally combined when derived.

connection_lost(exc)
Called when the connection is lost or closed.

Parameters exc (Exception) – exception. None indicates a closing EOF sent by this end.

collection_made(transport)
Callback for connection made to server.

data_received(data)
Process bytes received by transport.

duration
Time elapsed since client connected, in seconds as float.

encoding(outgoing=None, incoming=None)
Return encoding for the given stream direction.

Parameters

• outgoing (bool) – Whether the return value is suitable for encoding bytes for transmission to server.

• incoming (bool) – Whether the return value is suitable for decoding bytes received by the client.

Raises TypeError – when a direction argument, either outgoing or incoming, was not set True.

Returns 'US-ASCII' for the directions indicated, unless BINARY RFC 856 has been negotiated for the direction indicated or :attr:`force_binary` is set True.

Return type str
eof_received()
Called when the other end calls write_eof() or equivalent.

get_extra_info(name, default=None)
Get optional client protocol or transport information.

dle
Time elapsed since data last received, in seconds as float.

pause_writing()
Called when the transport’s buffer goes over the high-water mark.

Pause and resume calls are paired – pause_writing() is called once when the buffer goes strictly over the high-water mark (even if subsequent writes increases the buffer size even more), and eventually resume_writing() is called once when the buffer size reaches the low-water mark.

Note that if the buffer size equals the high-water mark, pause_writing() is not called – it must go strictly over. Conversely, resume_writing() is called when the buffer size is equal or lower than the low-water mark. These end conditions are important to ensure that things go as expected when either mark is zero.
NOTE: This is the only Protocol callback that is not called through EventLoop.call_soon() – if it were, it would have no effect when it’s most needed (when the app keeps writing without yielding until pause_writing() is called).

**resume_writing()**
Called when the transport’s buffer drains below the low-water mark.
See pause_writing() for details.

**send_charset (offered)**
Callback for responding to CHARSET requests.

Receives a list of character encodings offered by the server as offered such as ('LATIN-1', 'UTF-8'), for which the client may return a value agreed to use, or None to disagree to any available offers. Server offerings may be encodings or codepages.

The default implementation selects any matching encoding that python is capable of using, preferring any that matches encoding if matched in the offered list.

**Parameters offered (list)** – list of CHARSET options offered by server.

**Returns** character encoding agreed to be used.

**Return type** str or None

**send_tspeed ()**
Callback for responding to TSPEED requests.

**send_ttype ()**
Callback for responding to TTYPE requests.

**send_xdisploc ()**
Callback for responding to XDISPLOC requests.

**open_connection**
Connect to a TCP Telnet server as a Telnet client.

**Parameters**

- **host (str)** – Remote Internet TCP Server host.
- **port (int)** – Remote Internet host TCP port.
- **client_factory (client_base.BaseClient)** – Client connection class factory.
  When None, TelnetTerminalClient is used when stdin is attached to a terminal, TelnetClient otherwise.
- **family (int)** – Same meaning as asyncio.loop.create_connection().
- **flags (int)** – Same meaning as asyncio.loop.create_connection().
- **local_addr (tuple)** – Same meaning as asyncio.loop.create_connection().
- **encoding (str)** – The default assumed encoding, or False to disable unicode support.
  This value is used for decoding bytes received by and encoding bytes transmitted to the Server. These values are preferred in response to NEW_ENVIRON RFC 1572 as environment value LANG, and by CHARSET RFC 2066 negotiation.

  The server’s attached reader, writer streams accept and return unicode, unless this value explicitly set False. In that case, the attached streams interfaces are bytes-only.
• **encoding_errors** (*str*) – Same meaning as `codecs.Codec.encode()`.

• **term** (*str*) – Terminal type sent for requests of TTYPE, **RFC 930** or as Environment value TERM by NEW_ENVIRON negotiation, **RFC 1672**.

• **cols** (*int*) – Client window dimension sent as Environment value COLUMNS by NEW_ENVIRON negotiation, **RFC 1672** or NAWS **RFC 1073**.

• **rows** (*int*) – Client window dimension sent as Environment value LINES by NEW_ENVIRON negotiation, **RFC 1672** or NAWS **RFC 1073**.

• **tspeed** (*tuple*) – Tuple of client BPS line speed in form `(rx, tx)` for receive and transmit, respectively. Sent when requested by TSPEED, **RFC 1079**.

• **xdisploc** (*str*) – String transmitted in response for request of XDISPLOC, **RFC 1086** by server (X11).

• **shell** – A async function that is called after negotiation completes, receiving arguments *(reader, writer)*. The reader is a `TelnetReader` instance, the writer is a `TelnetWriter` instance.

• **connect_minwait** (*float*) – The client allows any additional telnet negotiations to be demanded by the server within this period of time before launching the shell. Servers should assert desired negotiation on-connect and in response to 1 or 2 round trips. A server that does not make any telnet demands, such as a TCP server that is not a telnet server will delay the execution of `shell` for exactly this amount of time.

• **connect_maxwait** (*float*) – If the remote end is not complaint, or otherwise confused by our demands, the shell continues anyway after the greater of this value has elapsed. A client that is not answering option negotiation will delay the start of the shell by this amount.

• **limit** (*int*) – The buffer limit for reader stream.

Return *(reader, writer)* The reader is a `TelnetReader` instance, the writer is a `TelnetWriter` instance.

### 2.3 client_base

Module provides class `BaseClient`.

```python
class BaseClient (shell=None, encoding='utf8', encoding_errors='strict', force_binary=False, connect_minwait=1.0, connect_maxwait=4.0, limit=None, waiter_closed=None, _waiter_connected=None):

Base Telnet Client Protocol.

Class initializer.

default_encoding = None
    encoding for new connections

connect_minwait = None
    minimum duration for check_negotiation().

connect_maxwait = None
    maximum duration for check_negotiation().

eof_received()
    Called when the other end calls write_eof() or equivalent.
```
connection_lost(exc)
Called when the connection is lost or closed.

Parameters exc (Exception) – exception. None indicates a closing EOF sent by this end.

collection_made(transport)
Called when a connection is made.

Ensure super().connection_made(transport) is called when derived.

begin_shell(result)

data_received(data)
Process bytes received by transport.

duration
Time elapsed since client connected, in seconds as float.

idle
Time elapsed since data last received, in seconds as float.

get_extra_info(name, default=None)
Get optional client protocol or transport information.

begin_negotiation()
Begin on-connect negotiation.

A Telnet client is expected to send only a minimal amount of client session options immediately after
connection, it is generally the server which dictates server option support.

Deriving implementations should always call super().begin_negotiation().

encoding(outgoing=False, incoming=False)
Encoding that should be used for the direction indicated.

The base implementation always returns encoding argument given to class initializer or, when unset
(None), US-ASCII.

check_negotiation(final=False)
Callback, return whether negotiation is complete.

Parameters final (bool) – Whether this is the final time this callback will be requested to
answer regarding protocol negotiation.

Returns Whether negotiation is over (client end is satisfied).

Return type bool
Method is called on each new command byte processed until negotiation is considered final, or after
connect_maxwait has elapsed, setting the _waiter_connected attribute to value self when
complete.

This method returns False until connect_minwait has elapsed, ensuring the server may batch telnet
negotiation demands without prematurely entering the callback shell.

Ensure super().check_negotiation() is called and conditionally combined when derived.

pause_writing()
Called when the transport’s buffer goes over the high-water mark.

Pause and resume calls are paired – pause_writing() is called once when the buffer goes strictly over
the high-water mark (even if subsequent writes increases the buffer size even more), and eventually re-
sume_writing() is called once when the buffer size reaches the low-water mark.
Note that if the buffer size equals the high-water mark, pause_writing() is not called – it must go strictly over. Conversely, resume_writing() is called when the buffer size is equal or lower than the low-water mark. These end conditions are important to ensure that things go as expected when either mark is zero.

NOTE: This is the only Protocol callback that is not called through EventLoop.call_soon() – if it were, it would have no effect when it’s most needed (when the app keeps writing without yielding until pause_writing() is called).

resume_writing()
  Called when the transport’s buffer drains below the low-water mark.
  See pause_writing() for details.

2.4 client_shell

telnet_client_shell(telnet_reader, telnet_writer)
  Minimal telnet client shell for POSIX terminals.
  This shell performs minimal tty mode handling when a terminal is attached to standard in (keyboard), notably raw mode is often set and this shell may exit only by disconnect from server, or the escape character, ^].
  stdin or stdout may also be a pipe or file, behaving much like nc(1).

2.5 server

The main function here is wired to the command line tool by name telnetlib3-server. If this server’s PID receives the SIGTERM signal, it attempts to shutdown gracefully.

The TelnetServer class negotiates a character-at-a-time (WILL-SGA, WILL-ECHO) session with support for negotiation about window size, environment variables, terminal type name, and to automatically close connections clients after an idle period.

class TelnetServer(term=‘unknown’, cols=80, rows=25, timeout=300, *args, **kwargs)
  Bases: telnetlib3.server_base.BaseServer
  Telnet Server protocol performing common negotiation.

  TTYPE_LOOPMAX = 8
  Maximum number of cycles to seek for all terminal types. We are seeking the repeat or cycle of a terminal table, choosing the first – but when negotiated by MUD clients, we chose the most Unix TERM appropriate,

  connection_made(transport)
  Called when a connection is made.
  Sets attributes _transport, _when_connected, _last_received, reader and writer.
  Ensure super().connection_made(transport) is called when derived.

data_received(data)
  Process bytes received by transport.

begin_negotiation()
  Begin on-connect negotiation.
  A Telnet server is expected to demand preferred session options immediately after connection. Deriving implementations should always call super().begin_negotiation().
**begin_advanced_negotiation()**

Begin advanced negotiation.

Callback method further requests advanced telnet options. Called once on receipt of any \texttt{DO} or \texttt{WILL} acknowledgments received, indicating that the remote end is capable of negotiating further.

Only called if sub-classing \texttt{begin_negotiation()} causes at least one negotiation option to be affirmatively acknowledged.

**check_negotiation (final=False)**

Callback, return whether negotiation is complete.

- **Parameters** 
  - \texttt{final (bool)} – Whether this is the final time this callback will be requested to answer regarding protocol negotiation.

- **Returns** 
  - Whether negotiation is over (server end is satisfied).

- **Return type** 
  - \texttt{bool}

Method is called on each new command byte processed until negotiation is considered final, or after \texttt{connect_maxwait} has elapsed, setting attribute \texttt{_waiter_connected} to value \texttt{self} when complete.

Ensure super().\texttt{check_negotiation()} is called and conditionally combined when derived.

**encoding (outgoing=None, incoming=None)**

Return encoding for the given stream direction.

- **Parameters** 
  - \texttt{outgoing (bool)} – Whether the return value is suitable for encoding bytes for transmission to client end.
  - \texttt{incoming (bool)} – Whether the return value is suitable for decoding bytes received from the client.

- **Raises** \texttt{TypeError} – when a direction argument, either \texttt{outgoing} or \texttt{incoming}, was not set \texttt{True}.

- **Returns** 
  - \texttt{US-ASCII} for the directions indicated, unless \texttt{BINARY RFC 856} has been negotiated for the direction indicated or \texttt{:attr"force_binary" is set True}.

- **Return type** 
  - \texttt{str}

**set_timeout (duration=-1)**

Restart or unset timeout for client.

- **Parameters** 
  - \texttt{duration (int)} – When specified as a positive integer, schedules Future for callback of \texttt{on_timeout()}. When \texttt{-1}, the value of \texttt{self.get_extra_info('timeout')} is used. When non-\texttt{True}, it is canceled.

**on_timeout ()**

Callback received on session timeout.

Default implementation writes “Timeout.” bound by CRLF and closes.

This can be disabled by calling \texttt{set_timeout()} with \texttt{:paramref:`~.set_timeout.duration`} value of \texttt{0} or value of the same for keyword argument \texttt{timeout}.

**on_naws (rows, cols)**

Callback receives NAWS response, \texttt{RFC 1073}.

- **Parameters** 
  - \texttt{rows (int)} – screen size, by number of cells in height.
• **cols** (*int*) – screen size, by number of cells in width.

**on_request_environ()**
Definition for NEW_ENVIRON request of client, **RFC 1572**.

This method is a callback from *request_environ()*, first entered on receipt of (WILL, NEW_ENVIRON) by server. The return value *defines the request made to the client* for environment values.

**Rtype list** a list of unicode character strings of US-ASCII characters, indicating the environment keys the server requests of the client. If this list contains the special byte constants, `USERVAR` or `VAR`, the client is allowed to volunteer any other additional user or system values.

Any empty return value indicates that no request should be made.

The default return value is:

```python
['LANG', 'TERM', 'COLUMNS', 'LINES', 'DISPLAY', 'COLORTERM',
 VAR, USERVAR, 'COLORTERM']
```

**on_environ**(mapping)
Callback receives NEW_ENVIRON response, **RFC 1572**.

**on_request_charset()**
Definition for CHARSET request by client, **RFC 2066**.

This method is a callback from *request_charset()*, first entered on receipt of (WILL, CHARSET) by server. The return value *defines the request made to the client* for encodings.

**Rtype list** a list of unicode character strings of US-ASCII characters, indicating the encodings offered by the server in its preferred order.

Any empty return value indicates that no encodings are offered.

The default return value begins:

```python
['UTF-8', 'UTF-16', 'LATIN1', 'US-ASCII', 'BIG5', 'GBK', ...]
```

**on_charset**(charset)
Callback for CHARSET response, **RFC 2066**.

**on_tspeed**(rx, tx)
Callback for TSPEED response, **RFC 1079**.

**on_ttype**(tty)
Callback for TTYPE response, **RFC 930**.

**on_xdisploc**(xdisploc)
Callback for XDISPLOC response, **RFC 1096**.

**begin_shell**(result)

**connection_lost**(exc)
Called when the connection is lost or closed.

**Parameters** `exc` (**Exception**) – exception. `None` indicates close by EOF.

**duration**
Time elapsed since client connected, in seconds as float.

**eof_received**()
Called when the other end calls write_eof() or equivalent.

This callback may be exercised by the nc(1) client argument `-z`.

2.5. server
get_extra_info(name, default=None)
Get optional server protocol or transport information.

idle
Time elapsed since data last received, in seconds as float.

negotiation_should_advance()
Whether advanced negotiation should commence.

Return type bool

Returns True if advanced negotiation should be permitted.

The base implementation returns True if any negotiation options were affirmatively acknowledged by
client, more than likely options requested in callback begin_negotiation().

pause_writing()
Called when the transport’s buffer goes over the high-water mark.

Pause and resume calls are paired – pause_writing() is called once when the buffer goes strictly over
the high-water mark (even if subsequent writes increases the buffer size even more), and eventually re-
sume_writing() is called once when the buffer size reaches the low-water mark.

Note that if the buffer size equals the high-water mark, pause_writing() is not called – it must go strictly
over. Conversely, resume_writing() is called when the buffer size is equal or lower than the low-water
mark. These end conditions are important to ensure that things go as expected when either mark is zero.

NOTE: This is the only Protocol callback that is not called through EventLoop.call_soon() – if it were,
it would have no effect when it’s most needed (when the app keeps writing without yielding until
pause_writing() is called).

resume_writing()
Called when the transport’s buffer drains below the low-water mark.

See pause_writing() for details.

timeout_connection()

parse_server_args()

create_server(host=None, port=23, protocol_factory=<class 'telnetlib3.server.TelnetServer'>, **kwds)
Create a TCP Telnet server.

Parameters

• host (str) – The host parameter can be a string, in that case the TCP server is bound to
host and port. The host parameter can also be a sequence of strings, and in that case the TCP
server is bound to all hosts of the sequence.

• port (int) – listen port for TCP Server.

• protocol_factory (server_base.BaseServer) – An alternate protocol factory
for the server, when unspecified, TelnetServer is used.

• shell (Callable) – A asyncio.coroutine() that is called after negotiation com-
pletes, receiving arguments (reader, writer). The reader is a TelnetReader in-
stance, the writer is a TelnetWriter instance.

• encoding (str) – The default assumed encoding, or False to disable unicode support.
Encoding may be negotiation to another value by the client through NEW_ENVIRON RFC
1572 by sending environment value of LANG, or by any legal value for CHARSET RFC
2066 negotiation.
The server's attached reader, writer streams accept and return unicode, unless this value explicitly set False. In that case, the attached streams interfaces are bytes-only.

- **encoding_errors** *(str)* – Same meaning as `codecs.Codec.encode()`. Default value is `strict`.

- **force_binary** *(bool)* – When True, the encoding specified is used for both directions even when BINARY mode, RFC 856, is not negotiated for the direction specified. This parameter has no effect when encoding=False.

- **term** *(str)* – Value returned for `writer.get_extra_info('term')` until negotiated by TTYPE RFC 930, or NAWS RFC 1572. Default value is 'unknown'.

- **cols** *(int)* – Value returned for `writer.get_extra_info('cols')` until negotiated by NAWS RFC 1572. Default value is 80 columns.

- **rows** *(int)* – Value returned for `writer.get_extra_info('rows')` until negotiated by NAWS RFC 1572. Default value is 25 rows.

- **timeout** *(int)* – Causes clients to disconnect if idle for this duration, in seconds. This ensures resources are freed on busy servers. When explicitly set to False, clients will not be disconnected for timeout. Default value is 300 seconds (5 minutes).

- **connect_maxwait** *(float)* – If the remote end is not complaint, or otherwise confused by our demands, the shell continues anyway after the greater of this value has elapsed. A client that is not answering option negotiation will delay the start of the shell by this amount.

- **limit** *(int)* – The buffer limit for the reader stream.

Return **asyncio.Server** The return value is the same as `asyncio.loop.create_server()`, An object which can be used to stop the service.

This function is a coroutine().

```python
run_server (host='localhost', port=6023, loglevel='info', logfile=None, logfmt='%(asctime)s %(levelname)s %(filename)s:%(lineno)d %(message)s', shell=<function telnet_server_shell>, encoding='utf8', force_binary=False, timeout=300, connect_maxwait=4.0)
```

Program entry point for server daemon.

This function configures a logger and creates a telnet server for the given keyword arguments, serving forever, completing only upon receipt of SIGTERM.

### 2.6 server_base

Module provides class `BaseServer`.

```python
class BaseServer (shell=None, _waiter_connected=None, _waiter_closed=None, encoding='utf8', encoding_errors='strict', force_binary=False, connect_maxwait=4.0, limit=None, reader_factory=<class 'telnetlib3.stream_reader.TelnetReader'>, reader_factory_encoding=<class 'telnetlib3.stream_reader.TelnetReaderUnicode'>, writer_factory=<class 'telnetlib3.stream_writer.TelnetWriter'>, writer_factory_encoding=<class 'telnetlib3.stream_writer.TelnetWriterUnicode'>)
```

**Bases:** `asyncio.streams.FlowControlMixin, asyncio.protocols.Protocol`

Base Telnet Server Protocol.

Class initializer.

```python
connect_maxwait = None
```

maximum duration for `check_negotiation()`.
timeout_connection()

eof_received()
    Called when the other end calls write_eof() or equivalent.
    
    This callback may be exercised by the nc(1) client argument -z.

connection_lost(exc)
    Called when the connection is lost or closed.

        Parameters exc (Exception) – exception. None indicates close by EOF.

collection_made(transport)
    Called when a connection is made.
    
    Sets attributes _transport, _when_connected, _last_received, reader and writer.
    
    Ensure super().connection_made(transport) is called when derived.

begin_shell(result)

data_received(data)
    Process bytes received by transport.

duration
    Time elapsed since client connected, in seconds as float.

idle
    Time elapsed since data last received, in seconds as float.

general_info(name, default=None)
    Get optional server protocol or transport information.

begin_negotiation()
    Begin on-connect negotiation.
    
    A Telnet server is expected to demand preferred session options immediately after connection. Deriving
    implementations should always call super().begin_negotiation().

begin_advanced_negotiation()
    Begin advanced negotiation.
    
    Callback method further requests advanced telnet options. Called once on receipt of any DO or WILL
    acknowledgments received, indicating that the remote end is capable of negotiating further.
    
    Only called if sub-classing begin_negotiation() causes at least one negotiation option to be affirm-
    atively acknowledged.

coding (outgoing=False, incoming=False)
    Encoding that should be used for the direction indicated.
    
    The base implementation always returns the encoding given to class initializer, or, when unset (None),
    US-ASCII.

negotiation_should_advance()
    Whether advanced negotiation should commence.

        Return type bool

        Returns True if advanced negotiation should be permitted.

        The base implementation returns True if any negotiation options were affirmatively acknowledged by
        client, more than likely options requested in callback begin_negotiation().

check_negotiation(final=False)
    Callback, return whether negotiation is complete.
Parameters final (bool) – Whether this is the final time this callback will be requested to answer regarding protocol negotiation.

Returns Whether negotiation is over (server end is satisfied).

Return type bool

Method is called on each new command byte processed until negotiation is considered final, or after connect_maxwait has elapsed, setting attribute _waiter_connected to value self when complete.

Ensure super().check_negotiation() is called and conditionally combined when derived.

pause_writing()
Called when the transport’s buffer goes over the high-water mark.

Pause and resume calls are paired – pause_writing() is called once when the buffer goes strictly over the high-water mark (even if subsequent writes increases the buffer size even more), and eventually resume_writing() is called once when the buffer size reaches the low-water mark.

Note that if the buffer size equals the high-water mark, pause_writing() is not called – it must go strictly over. Conversely, resume_writing() is called when the buffer size is equal or lower than the low-water mark. These end conditions are important to ensure that things go as expected when either mark is zero.

NOTE: This is the only Protocol callback that is not called through EventLoop.call_soon() – if it were, it would have no effect when it’s most needed (when the app keeps writing without yielding until pause_writing() is called).

resume_writing()
Called when the transport’s buffer drains below the low-water mark.

See pause_writing() for details.

2.7 server_shell

telnet_server_shell (reader, writer)
A default telnet shell, appropriate for use with telnetlib3.create_server.

This shell provides a very simple REPL, allowing introspection and state toggling of the connected client session.

2.8 slc

Special Line Character support for Telnet Linemode Option (RFC 1184).

class SLC (mask=b’x03’, value=b’x00’)
Bases: object

Defines the willingness to support a Special Linemode Character.

Defined by its SLC support level, mask and default keyboard ASCII byte value (may be negotiated by client).

level
Returns SLC level of support.

nosupport
Returns True if SLC level is SLC_NOSUPPORT.
cantchange
    Returns True if SLC level is SLC_CANTCHANGE.

variable
    Returns True if SLC level is SLC_VARIABLE.

default
    Returns True if SLC level is SLC_DEFAULT.

ack
    Returns True if SLC_ACK bit is set.

flushin
    Returns True if SLC_FLUSHIN bit is set.

flushout
    Returns True if SLC_FLUSHOUT bit is set.

set_value(value)
    Set SLC keyboard ascii value to byte.

set_mask(mask)
    Set SLC option mask, mask.

set_flag(flag)
    Set SLC option flag, flag.

class SLC(mask=b'x03', value=b'x00')
Bases: object

Defines the willingness to support a Special Linemode Character.
Defined by its SLC support level, mask and default keyboard ASCII byte value (may be negotiated by client).

level
    Returns SLC level of support.

nosupport
    Returns True if SLC level is SLC_NOSUPPORT.

cantchange
    Returns True if SLC level is SLC_CANTCHANGE.

variable
    Returns True if SLC level is SLC_VARIABLE.

default
    Returns True if SLC level is SLC_DEFAULT.

ack
    Returns True if SLC_ACK bit is set.

flushin
    Returns True if SLC_FLUSHIN bit is set.

flushout
    Returns True if SLC_FLUSHOUT bit is set.

set_value(value)
    Set SLC keyboard ascii value to byte.

set_mask(mask)
    Set SLC option mask, mask.
set_flag (flag)
Set SLC option flag, flag.

class SLC_nosupport
Bases: telnetlib3.slc.SLC
SLC definition inferring our unwillingness to support the option.

ack
Returns True if SLC_ACK bit is set.

cantchange
Returns True if SLC level is SLC_CANTCHANGE.

default
Returns True if SLC level is SLC_DEFAULT.

flushin
Returns True if SLC_FLUSHIN bit is set.

flushout
Returns True if SLC_FLUSHIN bit is set.

level
Returns SLC level of support.

nosupport
Returns True if SLC level is SLC_NOSUPPORT.

set_flag (flag)
Set SLC option flag, flag.

set_mask (mask)
Set SLC option mask, mask.

set_value (value)
Set SLC keyboard ascii value to byte.

variable
Returns True if SLC level is SLC_VARIABLE.

generate_slctab (tabset={b'\x01': <telnetlib3.slc.SLC object>, b'\x02': <telnetlib3.slc.SLC object>, b'\x03': <telnetlib3.slc.SLC object>, b'\x05': <telnetlib3.slc.SLC object>, b'\x07': <telnetlib3.slc.SLC object>, b'\x08': <telnetlib3.slc.SLC object>, b'\x0b': <telnetlib3.slc.SLC object>, b'\x0c': <telnetlib3.slc.SLC object>, b'\x0d': <telnetlib3.slc.SLC object>, b'\x0f': <telnetlib3.slc.SLC object>, b'\x10': <telnetlib3.slc.SLC object>, b'\x12': <telnetlib3.slc.SLC_nosupport object>})
Returns full ‘SLC Tab’ for definitions found using tabset. Functions not listed in tabset are set as SLC_NOSUPPORT.

generate_forwardmask (binary_mode, tabset, ack=False)
Generate a Forwardmask instance.

Generate a 32-byte (binary_mode is True) or 16-byte (False) Forwardmask instance appropriate for the specified slctab. A Forwardmask is formed by a bitmask of all 256 possible 8-bit keyboard ascii input, or, when not ‘outbinary’, a 16-byte 7-bit representation of each value, and whether or not they should be “forwarded” by the client on the transport stream.

snoop (byte, slctab, slc_callbacks)
Scan slctab for matching byte values.
Returns (callback, func_byte, slc_definition) on match. Otherwise, (None, None, None). If no callback is assigned, the value of callback is always None.

class Linemode (mask=b'x00')
    Bases: object
    A mask of LMODE_MODE_LOCAL means that all line editing is performed on the client side (default). A mask of NULL () indicates that editing is performed on the remote side. Valid bit flags of mask are: LMODE_MODE_TRAPSIG, LMODE_MODE_ACK, LMODE_MODE_SOFT_TAB, and LMODE_MODE_LIT_ECHO.

    local
        True if linemode is local.

    remote
        True if linemode is remote.

    trapsig
        True if signals are trapped by client.

    ack
        Returns True if mode has been acknowledged.

    soft_tab
        Returns True if client will expand horizontal tab ( ).

    lit_echo
        Returns True if non-printable characters are displayed as-is.

class Forwardmask (value, ack=False)
    Bases: object
    Forwardmask object using the bytemask value received by server.

    Parameters value (bytes) – bytemask value received by server after IAC SB LINEMODE DO FORWARDMASK. It must be a bytearray of length 16 or 32.

    description_table ()
        Returns list of strings describing obj as a tabular ASCII map.

    name_slc_command (byte)
        Given an SLC byte, return global mnemonic as string.

2.9 stream_reader

Module provides class TelnetReader and TelnetReaderUnicode.

class TelnetReader (limit=65536)
    Bases: object
    This is a copy of asyncio.StreamReader, with a little care for telnet-like readline(), and something about _waiter which I don’t really

    exception ()

    set_exception (exc)

    set_transport (transport)

    feed_eof ()
at_eof()
Return True if the buffer is empty and ‘feed_eof’ was called.

feed_data(data)

connection_closed

close()

read(n=-1)
Read up to n bytes from the stream.
If n is not provided, or set to -1, read until EOF and return all read bytes. If the EOF was received and the internal buffer is empty, return an empty bytes object.
If n is zero, return empty bytes object immediately.
If n is positive, this function try to read n bytes, and may return less or equal bytes than requested, but at least one byte. If EOF was received before any byte is read, this function returns empty byte object.
Returned value is not limited with limit, configured at stream creation.
If stream was paused, this function will automatically resume it if needed.

readexactly(n)
Read exactly n bytes.
Raise an IncompleteReadError if EOF is reached before n bytes can be read. The IncompleteReadError.partial attribute of the exception will contain the partial read bytes.
if n is zero, return empty bytes object.
Returned value is not limited with limit, configured at stream creation.
If stream was paused, this function will automatically resume it if needed.

readline()
Read one line.
Where “line” is a sequence of characters ending with CR LF, LF, or CR NUL. This readline function is a strict interpretation of Telnet Protocol RFC 854.
The sequence “CR LF” must be treated as a single “new line” character and used whenever their combined action is intended; The sequence “CR NUL” must be used where a carriage return alone is actually desired; and the CR character must be avoided in other contexts.
And therefor, a line does not yield for a stream containing a CR if it is not succeeded by NUL or LF.

<table>
<thead>
<tr>
<th>Given stream</th>
<th>readline() yields</th>
</tr>
</thead>
<tbody>
<tr>
<td>--\r\x00---</td>
<td>--\r,---...</td>
</tr>
<tr>
<td>--\r\n---</td>
<td>--\r\n,---...</td>
</tr>
<tr>
<td>--\n---</td>
<td>--\n,---...</td>
</tr>
<tr>
<td>--\r---</td>
<td>--\r,---...</td>
</tr>
</tbody>
</table>

If EOF is received before the termination of a line, the method will yield the partially read string.

readuntil(separator=b‘\n’)
Read data from the stream until separator is found.
On success, the data and separator will be removed from the internal buffer (consumed). Returned data will include the separator at the end.
Configured stream limit is used to check result. Limit sets the maximal length of data that can be returned, not counting the separator.
If an EOF occurs and the complete separator is still not found, an IncompleteReadError exception will be raised, and the internal buffer will be reset. The IncompleteReadError.partial attribute may contain the separator partially.

If the data cannot be read because of over limit, a LimitOverrunError exception will be raised, and the data will be left in the internal buffer, so it can be read again.

class TelnetReaderUnicode (fn_encoding, *, limit=65536, encoding_errors='replace')

Bases: telnetlib3.stream_reader.TelnetReader

A Unicode StreamReader interface for Telnet protocol.

Parameters fn_encoding (Callable) – function callback, receiving boolean keyword argument, incoming=True, which is used by the callback to determine what encoding should be used to decode the value in the direction specified.

at_eof()
Return True if the buffer is empty and ‘feed_eof’ was called.

close()

collection_closed

exception()

feed_data (data)

feed_eof()

read (n=-1)
Read up to n bytes.

If the EOF was received and the internal buffer is empty, return an empty string.

Parameters n (int) – If n is not provided, or set to -1, read until EOF and return all characters as one large string.

Return type  str

readexactly (n)
Read exactly n unicode characters.

Raises asyncio.IncompleteReadError – if the end of the stream is reached before n can be read. the asyncio.IncompleteReadError.partial attribute of the exception contains the partial read characters.

Return type  str

readline()
Read one line.

See ancestor method, readline() for details.

readuntil (separator=b’n’)
Read data from the stream until separator is found.

On success, the data and separator will be removed from the internal buffer (consumed). Returned data will include the separator at the end.

Configured stream limit is used to check result. Limit sets the maximal length of data that can be returned, not counting the separator.

If an EOF occurs and the complete separator is still not found, an IncompleteReadError exception will be raised, and the internal buffer will be reset. The IncompleteReadError.partial attribute may contain the separator partially.
If the data cannot be read because of over limit, a LimitOverrunError exception will be raised, and the data will be left in the internal buffer, so it can be read again.

```python
set_exception(exc)
set_transport(transport)
decode(buf, final=False)
```

Decode bytes `buf` using preferred encoding.

---

### 2.10 stream_writer

Module provides `TelnetWriter` and `TelnetWriterUnicode`.

#### class `TelnetWriter`

```python
class TelnetWriter(transport, protocol, *, client=False, server=False, reader=None)
```

This is a copy of `asyncio.StreamWriter`, except that it is a Telnet IAC Interpreter implementing the telnet protocol.

Almost all negotiation actions are performed through the writer interface, as any action requires writing bytes to the underlying stream. This class implements `feed_byte()`, which acts as a Telnet *Is-A-Command* (IAC) interpreter.

The significance of the last byte passed to this method is tested by instance attribute `is_oob`, following the call to `feed_byte()` to determine whether the given byte is in or out of band.

A minimal Telnet Protocol method, `asyncio.Protocol.data_received()`, should forward each byte to `feed_byte()`, which returns True to indicate the given byte should be forwarded to a Protocol reader method.

**Parameters**

- `client (bool)`: Whether the IAC interpreter should react from the client point of view.
- `server (bool)`: Whether the IAC interpreter should react from the server point of view.

```python
byte_count = 0
```

Total bytes sent to `feed_byte()`

```python
lflow = True
```

Whether flow control is enabled.

```python
xon_any = False
```

Whether flow control enabled by Transmit-Off (XOFF) (Ctrl-s), should re-enable Transmit-On (XON) only on receipt of XON (Ctrl-q). When False, any keypress from client re-enables transmission.

```python
iac_received = None
```

Whether the last byte received by `feed_byte()` is the beginning of an IAC command.

```python
cmd_received = None
```

Whether the last byte received by `feed_byte()` begins an IAC command sequence.

```python
slc_received = None
```

Whether the last byte received by `feed_byte()` is a matching special line character value, if negotiated.

```python
slc_simulated = True
```

SLC function values and callbacks are fired for clients in Kludge mode not otherwise capable of negotiating LINEMODE, providing transport remote editing function callbacks for dumb clients.

```python
default_slc_tab = {b'\x01': <telnetlib3.slc.SLC object>, b'\x02': <telnetlib3.slc.SLC object>}
```
default_linemode = <b>'\x10':
    lit_echo=True, soft_tab=False, ack=False, trapsig=False,
Initial line mode requested by server if client supports LINEMODE negotiation (remote line editing and literal echo of control chars)

pending_option = None
    Dictionary of telnet option byte(s) that follow an IAC-DO or IAC-DONT command, and contains a value of True until IAC-WILL or IAC-WONT has been received by remote end.

local_option = None
    Dictionary of telnet option byte(s) that follow an IAC-WILL or IAC-WONT command, sent by our end, indicating state of local capabilities.

remote_option = None
    Dictionary of telnet option byte(s) that follow an IAC-WILL or IAC-WONT command received by remote end, indicating state of remote capabilities.

slctab = None
    SLC Tab (SLC Functions and their support level, and ascii value)

collection_closed

transport

close()

is_closing()

write(data)
    Write a bytes object to the protocol transport.

    Return type  None

writelines(lines)
    Write unicode strings to transport.

    Note that newlines are not added. The sequence can be any iterable object producing strings. This is equivalent to calling write() for each string.

write_eof()

can_write_eof()

feed_byte(byte)
    Feed a single byte into Telnet option state machine.

    Parameters byte(int) – an 8-bit byte value as integer (0-255), or a bytes array. When a bytes array, it must be of length 1.

    Rtype  bool  Whether the given byte is “in band”, that is, should be duplicated to a connected terminal or device. False is returned for an IAC command for each byte until its completion.

get_extra_info(name, default=None)
    Get optional server protocol information.

protocol
    The (Telnet) protocol attached to this stream.

server
    Whether this stream is of the server’s point of view.

client
    Whether this stream is of the client’s point of view.
inbinary
Whether binary data is expected to be received on reader, RFC 856.

outbinary
Whether binary data may be written to the writer, RFC 856.

**echo** *(data)*
Conditionally write data to transport when “remote echo” enabled.

**Parameters** data *(bytes)* – string received as input, conditionally written.

**Return type** None
The default implementation depends on telnet negotiation willingness for local echo, only an RFC-compliant telnet client will correctly set or unset echo accordingly by demand.

**will_echo**
Whether Server end is expected to echo back input sent by client.

From server perspective: the server should echo (duplicate) client input back over the wire, the client is awaiting this data to indicate their input has been received.

From client perspective: the server will not echo our input, we should chose to duplicate our input to standard out ourselves.

**mode**
String describing NVT mode.

**Rtype** str One of:

- **kludge**: Client acknowledges WILL-ECHO, WILL-SGA. character-at-time and remote line editing may be provided.
- **local**: Default NVT half-duplex mode, client performs line editing and transmits only after pressing send (usually CR)
- **remote**: Client supports advanced remote line editing, using mixed-mode local line buffering (optionally, echoing) until send, but also transmits buffer up to and including special line characters (SLCs).

**is_oob**
The previous byte should not be received by the API stream.

**linemode**
Linemode instance for stream.

**Note**: value is meaningful after successful LINEMODE negotiation, otherwise does not represent the linemode state of the stream.

Attributes of the stream’s active linemode may be tested using boolean instance attributes, edit, trapsig, soft_tab, lit_echo, remote, local.

**send_iac**(buf)
Send a command starting with IAC (base 10 byte value 255).

No transformations of bytes are performed. Normally, if the byte value 255 is sent, it is escaped as IAC + IAC. This method ensures it is not escaped.

**iac**(cmd, opt=b")"
Send Is-A-Command 3-byte negotiation command.
Returns True if command was sent. Not all commands are legal in the context of client, server, or pending negotiation state, emitting a relevant debug warning to the log handler if not sent.

**send_ga()**
Transmit IAC GA (Go-Ahead).
Returns True if sent. If IAC-DO-SGA has been received, then False is returned and IAC-GA is not transmitted.

**send_eor()**
Transmit IAC CMD_EOR (End-of-Record), RFC 885.
Returns True if sent. If IAC-DO-EOR has not been received, False is returned and IAC-CMD_EOR is not transmitted.

**request_status()**
Send IAC-SB-STATUS-SEND sub-negotiation (RFC 859).
This method may only be called after IAC-WILL-STATUS has been received. Returns True if status request was sent.

**request_tspeed()**
Send IAC-SB-TSPEED-SEND sub-negotiation, RFC 1079.
This method may only be called after IAC-WILL-TSPEED has been received. Returns True if TSPEED request was sent.

**request_charset()**
Request sub-negotiation CHARSET, RFC 2066.
Returns True if request is valid for telnet state, and was sent.

The sender requests that all text sent to and by it be encoded in one of character sets specified by string list codepages, which is determined by function value returned by callback registered using `set_ext_send_callback()` with value CHARSET.

**request_environ()**
Request sub-negotiation NEW_ENVIRON, RFC 1572.
Returns True if request is valid for telnet state, and was sent.

**request_xdisploc()**
Send XDISPLOC, SEND sub-negotiation, RFC 1086.
Returns True if request is valid for telnet state, and was sent.

**request_ttype()**
Send TTYPE SEND sub-negotiation, RFC 930.
Returns True if request is valid for telnet state, and was sent.

**request_forwardmask(fmask=None)**
Request the client forward their terminal control characters.
Characters are indicated in the Forwardmask instance fmask. When fmask is None, a forwardmask is generated for the SLC characters registered by slctab.

**send_lineflow_mode()**
Send LFLOW mode sub-negotiation, RFC 1372.
Returns True if request is valid for telnet state, and was sent.

**send_linemode(linemode=None)**
Set and Inform other end to agree to change to linemode, linemode.
An instance of the Linemode class, or self.linemode when unset.

**set_iac_callback** *(cmd, func)*
Register callable `func` as callback for IAC `cmd`.
- BRK, IP, AO, AYT, EC, EL, CMD_EOR, EOF, SUSP, ABORT, and NOP.
  These callbacks receive a single argument, the IAC `cmd` which triggered it.

**handle_nop** *(cmd)*
Handle IAC No-Operation (NOP).

**handle_ga** *(cmd)*
Handle IAC Go-Ahead (GA).

**handle_dm** *(cmd)*
Handle IAC Data-Mark (DM).

**handle_el** *(byte)*
Handle IAC Erase Line (EL, SLC_EL).
  Provides a function which discards all the data ready on current line of input. The prompt should be
  re-displayed.

**handle_eor** *(byte)*
Handle IAC End of Record (CMD_EOR, SLC_EOR).

**handle_abort** *(byte)*
Handle IAC Abort (ABORT, SLC_ABORT).
  Similar to Interrupt Process (IP), but means only to abort or terminate the process to which the NVT is
  connected.

**handle_eof** *(byte)*
Handle IAC End of Record (EOF, SLC_EOF).

**handle_susp** *(byte)*
Handle IAC Suspend Process (SUSP, SLC_SUSP).
  Suspends the execution of the current process attached to the NVT in such a way that another process will
  take over control of the NVT, and the suspended process can be resumed at a later time.
  If the receiving system does not support this functionality, it should be ignored.

**handle_brk** *(byte)*
Handle IAC Break (BRK, SLC_BRK).
  Sent by clients to indicate BREAK keypress. This is not the same as IP (^c), but a means to map
  sysystem-dependent break key such as found on an IBM Systems.

**handle_ayt** *(byte)*
Handle IAC Are You There (AYT, SLC_AYT).
  Provides the user with some visible (e.g., printable) evidence that the system is still up and running.

**handle_ip** *(byte)*
Handle IAC Interrupt Process (IP, SLC_IP).

**handle_ao** *(byte)*
Handle IAC Abort Output (AO) or SLC_AO.
  Discards any remaining output on the transport buffer.
    [… ] a reasonable implementation would be to suppress the remainder of the text string, but
  transmit the prompt character and the preceding <CR><LF>.
**handle_ec** *(byte)*  
Handle IAC Erase Character (EC, SLC_EC).

Provides a function which deletes the last preceding undeleted character from data ready on current line of input.

**handle_tm** *(cmd)*  
Handle IAC (WILL, WONT, DO, DONT) Timing Mark (TM).

TM is essentially a NOP that any IAC interpreter must answer, if at least it answers WONT to unknown options (required), it may still be used as a means to accurately measure the “ping” time.

**set_slc_callback** *(slc_byte, func)*  
Register func as callable for receipt of slc_byte.

**Parameters**

- **slc_byte**(bytes) – any of SLC_SYNCH, SLC_BRK, SLC_IP, SLC_AO, SLC_AYT, SLC_EOL, SLC_ABORT, SLC_EOF, SLC_SUSP, SLC_EC, SLC_EL, SLC_EW, SLC_RP, SLC_XON, SLC_XOFF…
- **func**(Callable) – These callbacks receive a single argument: the SLC function byte that fired it. Some SLC and IAC functions are intermixed; which signaling mechanism used by client can be tested by evaluating this argument.

**handle_ew** *(slc)*  
Handle SLC_EW (Erase Word).

Provides a function which deletes the last preceding undeleted character, and any subsequent bytes until next whitespace character from data ready on current line of input.

**handle_rp** *(slc)*  
Handle SLC Repaint (RP).

**handle_lnext** *(slc)*  
Handle SLC Literal Next (LNEXT) (Next character is received raw).

**handle_xon** *(byte)*  
Handle SLC Transmit-On (XON).

**handle_xoff** *(byte)*  
Handle SLC Transmit-Off (XOFF).

**set_ext_send_callback** *(cmd, func)*  
Register callback for inquiries of sub-negotiation of cmd.

**Parameters**

- **func**(Callable) – A callable function for the given cmd byte. Note that the return type must match those documented.
- **cmd**(bytes) – These callbacks must return any number of arguments, for each registered cmd byte, respectively:
  - SNDLOC: for clients, returning one argument: the string describing client location, such as b'ROOM  641-A', RFC 779.
  - NAWS: for clients, returning two integer arguments (width, height), such as (80, 24), RFC 1073.
  - TSPEED: for clients, returning two integer arguments (rx, tx) such as (57600, 57600), RFC 1079.
- TTYPE: for clients, returning one string, usually the terminfo(5) database capability name, such as ‘xterm’, RFC 1091.
- XDISPLC: for clients, returning one string, the DISPLAY host value, in form of <host>:<dispnum>[.<screennum>], RFC 1096.
- NEW_ENVIRON: for clients, returning a dictionary of (key, val) pairs of environment item values, RFC 1408.
- CHARSET: for clients, receiving iterable of strings of character sets requested by server, callback must return one of those strings given, RFC 2066.

```
set_ext_callback(cmd, func)
```

Register func as callback for receipt of cmd negotiation.

**Parameters cmd (bytes)** – One of the following listed bytes:

- **LOGOUT**: for servers and clients, receiving one argument. Server end may receive DO or DONT as argument cmd, indicating client’s wish to disconnect, or a response to WILL, LOGOUT, indicating it’s wish not to be automatically disconnected. Client end may receive WILL or WONT, indicating server’s wish to disconnect, or acknowledgment that the client will not be disconnected.
- **SNDLOC**: for servers, receiving one argument: the string describing the client location, such as 'ROOM 641-A', RFC 779.
- **NAWS**: for servers, receiving two integer arguments (width, height), such as (80, 24), RFC 1073.
- **TSPEED**: for servers, receiving two integer arguments (rx, tx) such as (57600, 57600), RFC 1079.
- **TTYPE**: for servers, receiving one string, usually the terminfo(5) database capability name, such as ‘xterm’, RFC 1091.
- **XDISPLC**: for servers, receiving one string, the DISPLAY host value, in form of <host>:<dispnum>[.<screennum>], RFC 1096.
- **NEW_ENVIRON**: for servers, receiving a dictionary of (key, val) pairs of remote client environment item values, RFC 1408.
- **CHARSET**: for servers, receiving one string, the character set negotiated by client. RFC 2066.

```
handle_xdisploc(xdisploc)
```

Receive XDISPLAY value xdisploc, RFC 1096.

```
handle_send_xdisploc()
```

Send XDISPLAY value xdisploc, RFC 1096.

```
handle_sndloc(location)
```

Receive LOCATION value location, RFC 779.

```
handle_send_sndloc()
```

Send LOCATION value location, RFC 779.

```
handle_ttype(ttype)
```

Receive TTYPE value ttype, RFC 1091.

A string value that represents client’s emulation capability.

Some example values: VT220, VT100, ANSITERM, ANSI, TTY, and 5250.

```
handle_send_ttype()
```

Send TTYPE value ttype, RFC 1091.

```
handle_naws(width, height)
```

Receive window size width and height, RFC 1073.
handle_send_naws()
Send window size width and height, RFC 1073.

handle_environ(env)
Receive environment variables as dict, RFC 1572.

handle_send_client_environ(keys)
Send environment variables as dict, RFC 1572.
If argument keys is empty, then all available values should be sent. Otherwise, keys is a set of environment keys explicitly requested.

handle_send_server_environ()
Server requests environment variables as list, RFC 1572.

handle_tspeed(rx, tx)
Receive terminal speed from TSPEED as int, RFC 1079.

handle_send_tspeed()
Send terminal speed from TSPEED as int, RFC 1079.

handle_charset(charset)
Receive character set as string, RFC 2066.

handle_send_client_charset(charsets)
Send character set selection as string, RFC 2066.
Given the available encodings presented by the server, select and return only one. Returning an empty string indicates that no selection is made (request is ignored).

handle_send_server_charset(charsets)
Send character set (encodings) offered to client, RFC 2066.

handle_logout(cmd)
Handle (IAC, (DO | DONT | WILL | WONT), LOGOUT), RFC 727.
Only the server end may receive (DO, DONT). Only the client end may receive (WILL, WONT).

handle_do(opt)
Process byte 3 of series (IAC, DO, opt) received by remote end.
This method can be derived to change or extend protocol capabilities, for most cases, simply returning True if supported, False otherwise.
In special cases of various RFC statutes, state is stored and answered in willing affirmative, with the exception of:
• DO TM is always answered WILL TM, even if it was already replied to. No state is stored ("Timing Mark"), and the IAC callback registered by set_ext_callback() for cmd TM is called with argument byte DO.
• DO LOGOUT executes extended callback registered by cmd LOGOUT with argument DO (indicating a request for voluntary logoff).
• DO STATUS sends state of all local, remote, and pending options.

handle_dont(opt)
Process byte 3 of series (IAC, DONT, opt) received by remote end.
This only results in self.local_option[opt] set to False, with the exception of (IAC, DONT, LOGOUT), which only signals a callback to handle_logout(DONT).

handle_will(opt)
Process byte 3 of series (IAC, DONT, opt) received by remote end.
The remote end requests we perform any number of capabilities. Most implementations require an answer in the affirmative with DO, unless DO has meaning specific for only client or server end, and dissenting with DONT.

WILL ECHO may only be received for clients, answered with DO. WILL NAWS may only be received for servers, answered with DO. BINARY and SGA are answered with DO. STATUS, NEW_ENVIRON, XDISPLOC, and TTYPE is answered with sub-negotiation SEND. The env variables requested in response to WILL NEW_ENVIRON is “SEND ANY”. All others are replied with DONT.

The result of a supported capability is a response of (IAC, DO, opt) and the setting of self.remote_option[opt] of True. For unsupported capabilities, RFC specifies a response of (IAC, DONT, opt). Similarly, set self.remote_option[opt] to False.

handle_wont (opt)
Process byte 3 of series (IAC, WONT, opt) received by remote end.
(IAC, WONT, opt) is a negative acknowledgment of (IAC, DO, opt) sent.
The remote end requests we do not perform a telnet capability.
It is not possible to decline a WONT. T.remote_option[opt] is set False to indicate the remote end’s refusal to perform opt.

handle_subnegotiation (buf)
Callback for end of sub-negotiation buffer.
SB options handled here are TTYPE, XDISPLOC, NEW_ENVIRON, NAWS, and STATUS, and are delegated to their handle_ equivalent methods. Implementors of additional SB options should extend this method.

drain()
Flush the write buffer.
The intended use is to write
w.write(data) await w.drain()

class TelnetWriterUnicode (transport, protocol, fn_encoding, *, encoding_errors='strict', **kwds)
Bases: telnetlib3.stream_writer.TelnetWriter
A Unicode StreamWriter interface for Telnet protocol.
See ancestor class, TelnetWriter for details.
Requires the fn_encoding callback, receiving mutually boolean keyword argument outgoing=True to determine what encoding should be used to decode the value in the direction specified.
The encoding may be conditionally negotiated by CHARSET, RFC 2066, or discovered by LANG environment variables by NEW_ENVIRON, RFC 1572.

encode (string, errors)
Encode string using protocol-preferred encoding.

Parameters

- errors (str) – same as meaning in codecs.Codec.encode(). When None, value of encoding_errors given to class initializer is used.
- errors – same as meaning in codecs.Codec.encode(), when None (default), value of class initializer keyword argument, encoding_errors.

write (string, errors=None)
Write unicode string to transport, using protocol-preferred encoding.
If the connection is closed, nothing is done.
Parameters

- **string** *(str)* - unicode string text to write to endpoint using the protocol’s preferred encoding. When the protocol encoding keyword is explicitly set to False, the given string should be only raw `b'bytes'`.

- **errors** *(str)* - same as meaning in `codecs.Codec.encode()`, when None (default), value of class initializer keyword argument, `encoding_errors`.

**Return type** None

**writelines** *(lines, errors=None)*

Write unicode strings to transport.

Note that newlines are not added. The sequence can be any iterable object producing strings. This is equivalent to calling write() for each string.

**echo** *(string, errors=None)*

Conditionally write `string` to transport when “remote echo” enabled.

Parameters

- **string** *(str)* - string received as input, conditionally written.

- **errors** *(str)* - same as meaning in `codecs.Codec.encode()`.

This method may only be called from the server perspective. The default implementation depends on telnet negotiation willingness for local echo: only an RFC-compliant telnet client will correctly set or unset echo accordingly by demand.

byte_count = 0

can_write_eof()

client

Whether this stream is of the client’s point of view.

close()

cmd_received = None

connection_closed

default_linemode = <b'\x10': lit_echo:True, soft_tab:False, ack:False, trapsig:False,>
default_slc_tab = {b'\x01': <telnetlib3.slc.SLC object>, b'\x02': <telnetlib3.slc.SLC object>}
drain()

Flush the write buffer.

The intended use is to write

w.write(data) await w.drain()

**feed_byte** *(byte)*

Feed a single byte into Telnet option state machine.

Parameters **byte** *(int)* - an 8-bit byte value as integer (0-255), or a bytes array. When a bytes array, it must be of length 1.

Rtype bool Whether the given byte is “in band”, that is, should be duplicated to a connected terminal or device. False is returned for an IAC command for each byte until its completion.

get_extra_info** *(name, default=None)*

Get optional server protocol information.
**handle_abort**(byte)
Handle IAC Abort (ABORT, SLC_ABORT).

Similar to Interrupt Process (IP), but means only to abort or terminate the process to which the NVT is connected.

**handle_ao**(byte)
Handle IAC Abort Output (AO) or SLC_AO.
Discards any remaining output on the transport buffer.

[...]
a reasonable implementation would be to suppress the remainder of the text string, but transmit the prompt character and the preceding <CR><LF>.

**handle_ayt**(byte)
Handle IAC Are You There (AYT, SLC_AYT).
Provides the user with some visible (e.g., printable) evidence that the system is still up and running.

**handle_brk**(byte)
Handle IAC Break (BRK, SLC_BRK).
Sent by clients to indicate BREAK keypress. This is not the same as IP (^c), but a means to map system-dependent break key such as found on an IBM Systems.

**handle_charset**(charset)
Receive character set as string, RFC 2066.

**handle_dm**(cmd)
Handle IAC Data-Mark (DM).

**handle_do**(opt)
Process byte 3 of series (IAC, DO, opt) received by remote end.
This method can be derived to change or extend protocol capabilities, for most cases, simply returning True if supported, False otherwise.

In special cases of various RFC statutes, state is stored and answered in willing affirmative, with the exception of:

- DO TM is always answered WILL TM, even if it was already replied to. No state is stored (“Timing Mark”), and the IAC callback registered by set_ext_callback() for cmd TM is called with argument byte DO.
- DO LOGOUT executes extended callback registered by cmd LOGOUT with argument DO (indicating a request for voluntary logoff).
- DO STATUS sends state of all local, remote, and pending options.

**handle_dont**(opt)
Process byte 3 of series (IAC, DONT, opt) received by remote end.
This only results in self.local_option[opt] set to False, with the exception of (IAC, DONT, LOGOUT), which only signals a callback to handle_logout(DONT).

**handle_ec**(byte)
Handle IAC Erase Character (EC, SLC_EC).
Provides a function which deletes the last preceding undeleted character from data ready on current line of input.

**handle_el**(byte)
Handle IAC Erase Line (EL, SLC_EL).
Provides a function which discards all the data ready on current line of input. The prompt should be re-displayed.

`handle_environ(env)`  
Receive environment variables as dict, RFC 1572.

`handle_eof(byte)`  
Handle IAC End of Record (EOF, SLC_EOF).

`handle_eor(byte)`  
Handle IAC End of Record (CMD_EOR, SLC_EOR).

`handle_ew(slc)`  
Handle SLC_EW (Erase Word).  
Provides a function which deletes the last preceding undeleted character, and any subsequent bytes until next whitespace character from data ready on current line of input.

`handle_ga(cmd)`  
Handle IAC Go-Ahead (GA).

`handle_ip(byte)`  
Handle IAC Interrupt Process (IP, SLC_IP).

`handle_lnext(slc)`  
Handle SLC Literal Next (LNEXT) (Next character is received raw).

`handle_logout(cmd)`  
Handle (IAC, (DO | DONT | WILL | WONT), LOGOUT), RFC 727.  
Only the server end may receive (DO, DONT). Only the client end may receive (WILL, WONT).

`handle_naws(width, height)`  
Receive window size width and height, RFC 1073.

`handle_nop(cmd)`  
Handle IAC No-Operation (NOP).

`handle_rp(slc)`  
Handle SLC Repaint (RP).

`handle_send_client_charset(charsets)`  
Send character set selection as string, RFC 2066.  
Given the available encodings presented by the server, select and return only one. Returning an empty string indicates that no selection is made (request is ignored).

`handle_send_client_environ(keys)`  
Send environment variables as dict, RFC 1572.  
If argument keys is empty, then all available values should be sent. Otherwise, keys is a set of environment keys explicitly requested.

`handle_send_naws()`  
Send window size width and height, RFC 1073.

`handle_send_server_charset(charsets)`  
Send character set (encodings) offered to client, RFC 2066.

`handle_send_server_environ()`  
Server requests environment variables as list, RFC 1572.

`handle_send_sndloc()`  
Send LOCATION value location, RFC 779.
### Telnetlib3 Documentation, Release 2.0.1

#### handle_send_tspeed()

Send terminal speed from TSPEED as int, **RFC 1079**.

#### handle_send_ttype()

Send TTYPE value `ttype`, **RFC 1091**.

#### handle_send_xdisploc()

Send XDISPLAY value `xdisploc`, **RFC 1096**.

#### handle_sndloc (location)

Receive LOCATION value `location`, **RFC 779**.

#### handle_send_sessioninfo (sessioninfo)

Send SESSIONINFO value `sessioninfo`, **RFC 1096**.

#### handle_send_username (username)

Send USER value `username`, **RFC 1096**.

#### handle_subnegotiation (buf)

Callback for end of sub-negotiation buffer.

SB options handled here are TTYPE, XDISPLOC, NEW_ENVIRON, NAWS, and STATUS, and are delegated to their handle_ equivalent methods. Implementors of additional SB options should extend this method.

#### handle_suspend (byte)

Handle IAC Suspend Process (SUSP, SLC_SUSP).

Suspends the execution of the current process attached to the NVT in such a way that another process will take over control of the NVT, and the suspended process can be resumed at a later time.

If the receiving system does not support this functionality, it should be ignored.

#### handle_tm (cmd)

Handle IAC (WILL, WONT, DO, DONT) Timing Mark (TM).

TM is essentially a NOP that any IAC interpreter must answer, if at least it answers WONT to unknown options (required), it may still be used as a means to accurately measure the “ping” time.

#### handle_tspeed (rx, tx)

Receive terminal speed from TSPEED as int, **RFC 1079**.

#### handle_ttype (tttype)

Receive TTYPE value `tttype`, **RFC 1091**.

A string value that represents client’s emulation capability.

Some example values: VT220, VT100, ANSITERM, ANSI, TTY, and 5250.

#### handle_will (opt)

Process byte 3 of series (IAC, DONT, opt) received by remote end.

The remote end requests we perform any number of capabilities. Most implementations require an answer in the affirmative with DO, unless DO has meaning specific for only client or server end, and dissenting with DONT.

WILL ECHO may only be received for clients, answered with DO. WILL NAWS may only be received for servers, answered with DO. BINARY and SGA are answered with DO. STATUS, NEW_ENVIRON, XDISPLOC, and TTYPE is answered with sub-negotiation SEND. The env variables requested in response to WILL NEW_ENVIRON is “SEND ANY”. All others are replied with DONT.

The result of a supported capability is a response of (IAC, DO, opt) and the setting of self.remote_option[opt] of True. For unsupported capabilities, RFC specifies a response of (IAC, DONT, opt). Similarly, set self.remote_option[opt] to False.

#### handle_wont (opt)

Process byte 3 of series (IAC, WONT, opt) received by remote end.

(IAC, WONT, opt) is a negative acknowledgment of (IAC, DO, opt) sent.
The remote end requests we do not perform a telnet capability.

It is not possible to decline a WONT. `T.remote_option[opt]` is set False to indicate the remote end’s refusal to perform `opt`.

### handle_xdisploc (xdisploc)
Receive XDISPLAY value `xdisploc`, RFC 1096.

### handle_xoff (byte)
Handle SLC Transmit-Off (XOFF).

### handle_xon (byte)
Handle SLC Transmit-On (XON).

### iac (cmd, opt=b")"
Send Is-A-Command 3-byte negotiation command.

Returns True if command was sent. Not all commands are legal in the context of client, server, or pending negotiation state, emitting a relevant debug warning to the log handler if not sent.

```python
iac_received = None
```

### inbinary
Whether binary data is expected to be received on reader, RFC 856.

### is_closing()

### is_oob
The previous byte should not be received by the API stream.

### lflow = True

### linemode
Linemode instance for stream.

### Note: value is meaningful after successful LINEMODE negotiation, otherwise does not represent the linemode state of the stream.

Attributes of the stream’s active linemode may be tested using boolean instance attributes, `edit`, `trapsig`, `soft_tab`, `lit_echo`, `remote`, `local`.

### mode
String describing NVT mode.

#### Rtype str
One of:

- **kludge**: Client acknowledges WILL-ECHO, WILL-SGA. character-at-a-time and remote line editing may be provided.

- **local**: Default NVT half-duplex mode, client performs line editing and transmits only after pressing send (usually CR)

- **remote**: Client supports advanced remote line editing, using mixed-mode local line buffering (optionally, echoing) until send, but also transmits buffer up to and including special line characters (SLCs).

### outbinary
Whether binary data may be written to the writer, RFC 856.

### protocol
The (Telnet) protocol attached to this stream.
**request_charset()**
Request sub-negotiation CHARSET, RFC 2066.

Returns True if request is valid for telnet state, and was sent.

The sender requests that all text sent to and by it be encoded in one of character sets specified by string list codepages, which is determined by function value returned by callback registered using `set_ext_send_callback()` with value CHARSET.

**request_environ()**
Request sub-negotiation NEW_ENVIRON, RFC 1572.

Returns True if request is valid for telnet state, and was sent.

**request_forwardmask**(fmask=None)
Request the client forward their terminal control characters.

Characters are indicated in the Forwardmask instance fmask. When fmask is None, a forwardmask is generated for the SLC characters registered by slctab.

**request_status()**
Send IAC-SB-STATUS-SEND sub-negotiation (RFC 859).

This method may only be called after IAC-WILL-STATUS has been received. Returns True if status request was sent.

**request_tspeed()**
Send IAC-SB-TSPEED-SEND sub-negotiation, RFC 1079.

This method may only be called after IAC-WILL-TSPEED has been received. Returns True if TSPEED request was sent.

**request_ttype()**
Send TTYPE SEND sub-negotiation, RFC 930.

Returns True if request is valid for telnet state, and was sent.

**request_xdisploc()**
Send XDISPLOC, SEND sub-negotiation, RFC 1086.

Returns True if request is valid for telnet state, and was sent.

**send_eor()**
Transmit IAC CMD_EOR (End-of-Record), RFC 885.

Returns True if sent. If IAC-DO-EOR has not been received, False is returned and IAC-CMD_EOR is not transmitted.

**send_ga()**
Transmit IAC GA (Go-Ahead).

Returns True if sent. If IAC-DO-SGA has been received, then False is returned and IAC-GA is not transmitted.

**send_iac**(buf)
Send a command starting with IAC (base 10 byte value 255).

No transformations of bytes are performed. Normally, if the byte value 255 is sent, it is escaped as IAC + IAC. This method ensures it is not escaped.

**send_lineflow_mode()**
Send LFLOW mode sub-negotiation, RFC 1372.

Returns True if request is valid for telnet state, and was sent.
send_linemode (linemode=None)
Set and Inform other end to agree to change to linemode, linemode.
An instance of the Linemode class, or self.linemode when unset.

server
Whether this stream is of the server’s point of view.

set_ext_callback (cmd, func)
Register func as callback for receipt of cmd negotiation.

Parameters

- cmd (bytes) – One of the following listed bytes:
  
  - LOGOUT: for servers and clients, receiving one argument. Server end may receive DO or DON'T as argument cmd, indicating client’s wish to disconnect, or a response to WILL, LOGOUT, indicating it’s wish not to be automatically disconnected. Client end may receive WILL or WONT, indicating server’s wish to disconnect, or acknowledgment that the client will not be disconnected.
  
  - SNDLOC: for servers, receiving one argument: the string describing the client location, such as 'ROOM 641-A', RFC 779.
  
  - NAWS: for servers, receiving two integer arguments (width, height), such as (80, 24), RFC 1073.
  
  - TSPEED: for servers, receiving two integer arguments (rx, tx) such as (57600, 57600), RFC 1079.
  
  - TTYPE: for servers, receiving one string, usually the terminfo(5) database capability name, such as ‘xterm’, RFC 1091.
  
  - XDISPLOC: for servers, receiving one string, the DISPLAY host value, in form of <host>:<dispnum>[.<screennum>], RFC 1096.
  
  - NEW_ENVIRON: for servers, receiving a dictionary of (key, val) pairs of remote client environment item values, RFC 1408.
  
  - CHARSET: for servers, receiving one string, the character set negotiated by client. RFC 2066.

set_ext_send_callback (cmd, func)
Register callback for inquires of sub-negotiation of cmd.

Parameters

- func (Callable) – A callable function for the given cmd byte. Note that the return type must match those documented.
- cmd (bytes) – These callbacks must return any number of arguments, for each registered cmd byte, respectively:
  
  - SNDLOC: for clients, returning one argument: the string describing client location, such as b'ROOM 641-A', RFC 779.
  
  - NAWS: for clients, returning two integer arguments (width, height), such as (80, 24), RFC 1073.
  
  - TSPEED: for clients, returning two integer arguments (rx, tx) such as (57600, 57600), RFC 1079.
  
  - TTYPE: for clients, returning one string, usually the terminfo(5) database capability name, such as ‘xterm’, RFC 1091.
  
  - XDISPLOC: for clients, returning one string, the DISPLAY host value, in form of <host>:<dispnum>[.<screennum>], RFC 1096.
  
  - NEW_ENVIRON: for clients, returning a dictionary of (key, val) pairs of environment item values, RFC 1408.
- CHARSET: for clients, receiving iterable of strings of character sets requested by server, callback must return one of those strings given, RFC 2066.

```python
set_iac_callback(cmd, func)
```
Register callable `func` as callback for IAC `cmd`.

BRK, IP, AO, AYT, EC, EL, CMD_EOR, EOF, SUSP, ABORT, and NOP.

These callbacks receive a single argument, the IAC `cmd` which triggered it.

```python
set_slc_callback(slc_byte, func)
```
Register `func` as callable for receipt of `slc_byte`.

**Parameters**

- `func(Callable)` - These callbacks receive a single argument: the SLC function byte that fired it. Some SLC and IAC functions are intermixed; which signaling mechanism used by client can be tested by evaluating this argument.

```python
slc_received = None
slc_simulated = True
```

```python
transport
```
`will_echo`
Whether Server end is expected to echo back input sent by client.

From server perspective: the server should echo (duplicate) client input back over the wire, the client is awaiting this data to indicate their input has been received.

From client perspective: the server will not echo our input, we should chose to duplicate our input to standard out ourselves.

```python
write_eof()
```

```python
xon_any = False
```

## 2.11 telopt

```python
name_command(byte)
```
Return string description for (maybe) telnet command byte.

```python
name_commands(cmds, sep=’,’)
```
Return string description for array of (maybe) telnet command bytes.
3.1 Implemented

3.2 Not Implemented

- **RFC 861**, “Telnet Extended Options List”, May 1983. describes a method of negotiating options after all possible 255 option bytes are exhausted by future implementations. This never happened (about 100 remain), it was perhaps, ambitious in thinking more protocols would incorporate Telnet (such as FTP did).

- **RFC 927**, “TACACS User Identification Telnet Option”, describes a method of identifying terminal clients by a 32-bit UUID, providing a form of ‘rlogin’. This system, published in 1984, was designed for MILNET by BBN, and the actual TACACS implementation is undocumented, though partially re-imagined by Cisco in RFC 1492. Essentially, the user’s credentials are forwarded to a TACACS daemon to verify that the client does in fact have access. The UUID is a form of an early Kerberos token.

- **RFC 933**, “Output Marking Telnet Option”, describes a method of sending banners, such as displayed on login, with an associated ID to be stored by the client. The server may then indicate at which time during the session the banner is relevant. This was implemented by Mitre for DOD installations that might, for example, display various levels of “TOP SECRET” messages each time a record is opened – preferably on the top, bottom, left or right of the screen.

- **RFC 946**, “Telnet Terminal Location Number Option”, only known to be implemented at Carnegie Mellon University in the mid-1980’s, this was a mechanism to identify a Terminal by ID, which would then be read and forwarded by gatewaying hosts. So that user traveling from host A -> B -> C appears as though his “from” address is host A in the system “who” and “finger” services. There exists more appropriate solutions, such as the “Report Terminal ID” sequences $\text{CSI} + c$ and $\text{CSI} + 0c$ for vt102, and $\text{ESC} + z$ (vt52), which sends a terminal ID in-band as ASCII.

- **RFC 1041**, “Telnet 3270 Regime Option”, Jan 1988
- **RFC 1043**, “Telnet Data Entry Terminal Option”, Feb 1988
- **RFC 1143**, “The Q Method of Implementing .. Option Negotiation”, Feb 1990
- **RFC 1205**, “5250 Telnet Interface”, Feb 1991
- **RFC 1412**, “Telnet Authentication: SPX”
- **RFC 1416**, “Telnet Authentication Option”
- **RFC 2217**, “Telnet Com Port Control Option”, Oct 1997

3.3 Additional Resources

These RFCs predate, or are superseded by, **RFC 854**, but may be relevant for study of the telnet protocol.

- **RFC 97** A First Cut at a Proposed Telnet Protocol
- **RFC 137** Telnet Protocol.
- **RFC 139** Discussion of Telnet Protocol.
- **RFC 318** Telnet Protocol.
- **RFC 328** Suggested Telnet Protocol Changes.
- **RFC 340** Proposed Telnet Changes.
• RFC 393 Comments on TELNET Protocol Changes.
• RFC 435 Telnet Issues.
• RFC 513 Comments on the new Telnet Specifications.
• RFC 529 A Note on Protocol Synch Sequences.
• RFC 559 Comments on the new Telnet Protocol and its Implementation.
• RFC 563 Comments on the RCTE Telnet Option.
• RFC 593 Telnet and FTP Implementation Schedule Change.
• RFC 595 Some Thoughts in Defense of the Telnet Go-Ahead.
• RFC 596 Second Thoughts on Telnet Go-Ahead.
• RFC 652 Telnet Output Carriage-Return Disposition Option.
• RFC 653 Telnet Output Horizontal Tabstops Option.
• RFC 654 Telnet Output Horizontal Tab Disposition Option.
• RFC 655 Telnet Output Formfeed Disposition Option.
• RFC 656 Telnet Output Vertical Tabstops Option.
• RFC 657 Telnet Output Vertical Tab Disposition Option.
• RFC 658 Telnet Output Linefeed Disposition.
• RFC 659 Announcing Additional Telnet Options.
• RFC 698 Telnet Extended ASCII Option.
• RFC 702 September, 1974, Survey of New-Protocol Telnet Servers.
• RFC 718 Comments on RCTE from the TENEX Implementation Experience.
• RFC 719 Discussion on RCTE.
• RFC 726 Remote Controlled Transmission and Echoing Telnet Option.
• RFC 728 A Minor Pitfall in the Telnet Protocol.
• RFC 732 Telnet Data Entry Terminal Option (Obsoletes: RFC 731)
• RFC 734 SUPDUP Protocol.
• RFC 735 Revised Telnet Byte Macro Option (Obsoletes: RFC 729, RFC 736)
• RFC 749 Telnet SUPDUP-Output Option.
• RFC 818 The Remote User Telnet Service.

The following further describe the telnet protocol and various extensions of related interest:

• “Mud Terminal Type Standard,” http://tintin.sourceforge.net/mtts/

3.3. Additional Resources
• “Telnet RFCs,” http://www.omnifarious.org/~hopper/telnet-rfc.html
• “Telnet Options”, http://www.iana.org/assignments/telnet-options/telnet-options.xml
Contributing

We welcome contributions via GitHub pull requests:

- Fork a Repo
- Creating a pull request

4.1 Developing

Prepare a developer environment. Then, from the telnetlib3 code folder:

```
pip install --editable .
```

Any changes made in this project folder are then made available to the python interpreter as the ‘telnetlib3’ module irregardless of the current working directory.

4.2 Running Tests

`Py.test <https://pytest.org>` is the test runner. Install and run tox

```
pip install --upgrade tox
tox
```

A convenience target, ‘develop’ is provided, which adds -vv and --looponfail arguments, where the tests automatically re-trigger on any file change:

```
tox -e develop
```
4.3 Code Formatting

To make code formatting easy on developers, and to simplify the conversation around pull request reviews, this project has adopted the black code formatter. This formatter must be run against any new code written for this project. The advantage is, you no longer have to think about how your code is styled; it’s all handled for you!

To make this even easier on you, you can set up most editors to auto-run black for you. We have also set up a pre-commit hook to run automatically on every commit, with just a small bit of extra setup:

```
pip install pre-commit
pre-commit install --install-hooks
```

Now, before each git commit is accepted, this hook will run to ensure the code has been properly formatted by black.

4.4 Style and Static Analysis

All standards enforced by the underlying tools are adhered to by this project, with the declarative exception of those found in landscape.yml, or inline using pylint: disable= directives.

Perform static analysis using tox target sa:

```
tox -esa
```
2.0.1

- bugfix: “write after close” is disregarded, caused many errors logged in socket.send()
- bugfix: in accessories.repr_mapping() about using shlex.quote on non-str, TypeError: expected string or bytes-like object, got ‘int’
- bugfix: about fn_encoding using repr() on TelnetReaderUnicode
- bugfix: TelnetReader.is_closing() raises AttributeError
- deprecation: TelnetReader.close and TelnetReader.connection_closed emit warning, use at_eof() and feed_eof() instead.
- deprecation: the loop argument are is no longer accepted by TelnetReader.
- enhancement: Add Generic Mud Communication Protocol support #63 by gtaylor!
- change: TelnetReader and TelnetWriter no longer derive from asyncio.StreamReader and asyncio.StreamWriter, this fixes some TypeError in signatures and runtime

2.0.0

- change: Support Python 3.9, 3.10, 3.11. Drop Python 3.6 and earlier, All code and examples have been updated to the new-style PEP-492 syntax.
- change: the loop, event_loop, and log arguments are no longer accepted to any class initializers.
- note: This release has a known memory leak when using the _waiter_connected and _waiter_closed arguments to Client or Shell class initializers, please do not use them A replacement “wait_for_negotiation” awaitable will be provided in a future release.
- enhancement: Add COM-PORT-OPTION subnegotiation support #57 by albireox

1.0.4

- bugfix: NoneType error on EOF/Timeout, introduced in previous version 1.0.3, #51 by zofy.

1.0.3

- bugfix: circular reference between transport and protocol, #43 by fried.
1.0.2
- add –speed argument to telnet client #35 by hughpyle.

1.0.1
- add python3.7 support, drop python 3.4 and earlier, #33 by AndrewNelis.

1.0.0
- First general release for standard API: Instead of encouraging twisted-like override of protocol methods, we provide a “shell” callback interface, receiving argument pairs (reader, writer).

0.5.0
- bugfix: linemode MODE is now acknowledged.
- bugfix: default stream handler sends 80 x 24 in cols x rows, not 24 x 80.
- bugfix: waiter_closed future on client defaulted to wrong type.
- bugfix: telnet shell (TelSh) no longer paints over final exception line.

0.4.0
- bugfix: cannot connect to IPv6 address as client.
- change: TelnetClient.CONNECT_DEFERED class attribute renamed DEFERRED. Default value changed to 50ms from 100ms.
- change: TelnetClient.waiter renamed to TelnetClient.waiter_closed.
- enhancement: TelnetClient.waiter_connected future added.

0.3.0
- bugfix: cannot bind to IPv6 address #5.
- enhancement: Futures waiter_connected, and waiter_closed added to server.
- change: TelSh.feed_slc merged into TelSh.feed_byte as slc_function keyword.
- change: TelnetServer.CONNECT_DEFERED class attribute renamed DEFERRED. Default value changed to 50ms from 100ms.
- enhancement: Default TelnetServer.PROMPT_IMMEDIATELY = False ensures prompt is not displayed until negotiation is considered final. It is no longer “aggressive”.
- enhancement: TelnetServer.pause_writing and resume_writing callback wired.
- enhancement: TelSh.pause_writing and resume_writing methods added.

0.2.4
- bugfix: pip installation issue #8.

0.2
- enhancement: various example programs were included in this release.

0.1
- Initial release.
CHAPTER 6

Indexes

- genindex
- modindex
Python Module Index

t

telnetlib3.accessories, 7
telnetlib3.client, 8
telnetlib3.client_base, 13
telnetlib3.client_shell, 15
telnetlib3.server, 15
telnetlib3.server_base, 19
telnetlib3.server_shell, 21
telnetlib3.slc, 21
telnetlib3.stream_reader, 24
telnetlib3.stream_writer, 27
telnetlib3.telopt, 43
| A | ack (Linemode attribute), 24 |
|   | ack (SLC attribute), 22 |
|   | ack (SLC_nosupport attribute), 23 |
|   | at_eof() (TelnetReader method), 24 |
|   | at_eof() (TelnetReaderUnicode method), 26 |
| B | BaseClient (class in telnetlib3.client_base), 13 |
|   | BaseServer (class in telnetlib3.server_base), 19 |
|   | begin_advanced_negotiation() (BaseServer method), 20 |
|   | begin_advanced_negotiation() (TelnetServer method), 15 |
|   | begin_negotiation() (BaseClient method), 14 |
|   | begin_negotiation() (BaseServer method), 20 |
|   | begin_negotiation() (TelnetClient method), 9 |
|   | begin_negotiation() (TelnetServer method), 15 |
|   | begin_negotiation() (TelnetTerminalClient method), 10 |
|   | begin_shell() (BaseClient method), 14 |
|   | begin_shell() (BaseServer method), 20 |
|   | begin_shell() (TelnetClient method), 9 |
|   | begin_shell() (TelnetServer method), 17 |
|   | begin_shell() (TelnetTerminalClient method), 10 |
|   | byte_count (TelnetWriter attribute), 27 |
|   | byte_count (TelnetWriterUnicode attribute), 36 |
| C | can_write_eof() (TelnetWriter method), 28 |
|   | can_write_eof() (TelnetWriterUnicode method), 36 |
|   | cantchange (SLC attribute), 21, 22 |
|   | cantchange (SLC_nosupport attribute), 23 |
|   | check_negotiation() (BaseClient method), 14 |
|   | check_negotiation() (BaseServer method), 20 |
|   | check_negotiation() (TelnetClient method), 9 |
|   | check_negotiation() (TelnetServer method), 16 |
|   | check_negotiation() (TelnetTerminalClient method), 10 |
|   | client (TelnetWriter attribute), 28 |
|   | client (TelnetWriterUnicode attribute), 36 |
|   | close() (TelnetReader method), 25 |
|   | close() (TelnetReaderUnicode method), 26 |
|   | close() (TelnetWriter method), 28 |
|   | close() (TelnetWriterUnicode method), 36 |
|   | cmd_received (TelnetWriter attribute), 27 |
|   | cmd_received (TelnetWriterUnicode attribute), 36 |
|   | connect_maxwait (BaseClient attribute), 13 |
|   | connect_maxwait (BaseServer attribute), 19 |
|   | connect_minwait (BaseClient attribute), 13 |
|   | connection_closed (TelnetReader attribute), 25 |
|   | connection_closed (TelnetWriterUnicode attribute), 26 |
|   | connection_closed (TelnetWriter attribute), 28 |
|   | connection_closed (TelnetWriterUnicode attribute), 36 |
|   | connection_lost() (BaseClient method), 13 |
|   | connection_lost() (BaseServer method), 20 |
|   | connection_lost() (TelnetClient method), 9 |
|   | connection_lost() (TelnetServer method), 17 |
|   | connection_lost() (TelnetTerminalClient method), 11 |
|   | connection_made() (BaseClient method), 14 |
|   | connection_made() (BaseServer method), 20 |
|   | connection_made() (TelnetClient method), 8 |
|   | connection_made() (TelnetServer method), 15 |
|   | connection_made() (TelnetTerminalClient method), 11 |
|   | create_server() (in module telnetlib3.server), 18 |
| D | data_received() (BaseClient method), 14 |
|   | data_received() (BaseServer method), 20 |
|   | data_received() (TelnetClient method), 9 |
|   | data_received() (TelnetServer method), 15 |
|   | data_received() (TelnetTerminalClient method), 11 |
|   | decode() (TelnetReaderUnicode method), 27 |
|   | default (SLC attribute), 22 |
|   | default (SLC_nosupport attribute), 23 |
default_encoding (BaseClient attribute), 13
default_linemode (TelnetWriter attribute), 27
default_linemode (TelnetWriterUnicode attribute),
36
DEFAULT_LOCALE (TelnetClient attribute), 8
DEFAULT_LOCALE (TelnetTerminalClient attribute), 10
default_slc_tab (TelnetWriter attribute), 27
default_slc_tab (TelnetWriterUnicode attribute),
36
description_table () (Forwardmask method), 24
drain () (TelnetWriter method), 35
drain () (TelnetWriterUnicode method), 36
duration (BaseClient attribute), 14
duration (BaseServer attribute), 20
duration (TelnetClient attribute), 9
duration (TelnetServer attribute), 17
duration (TelnetTerminalClient attribute), 11

E
echo () (TelnetWriter method), 29
echo () (TelnetWriterUnicode method), 36
eightbits () (in module telnetlib3.accessories), 7
encode () (TelnetWriterUnicode method), 35
encoding () (BaseClient attribute), 14
encoding () (BaseServer method), 20
encoding () (TelnetClient method), 8
encoding () (TelnetServer method), 16
encoding () (TelnetTerminalClient method), 11
encoding_from_lang () (in module telnetlib3.accessories), 7
eof_received () (BaseClient method), 13
eof_received () (BaseServer method), 20
eof_received () (TelnetClient method), 9
eof_received () (TelnetServer method), 17
eof_received () (TelnetTerminalClient method), 11
exception () (TelnetReader method), 24
exception () (TelnetReaderUnicode method), 26

F
feed_byte () (TelnetWriter method), 28
feed_byte () (TelnetWriterUnicode method), 36
feed_data () (TelnetReader method), 25
feed_data () (TelnetReaderUnicode method), 26
feed_eof () (TelnetReader method), 24
feed_eof () (TelnetReaderUnicode method), 26
flushin (SLC attribute), 22
flushin (SLC_nosupport attribute), 23
flushout (SLC attribute), 22
flushout (SLC_nosupport attribute), 23
Forwardmask (class in telnetlib3.slc), 23
function_lookup () (in module telnetlib3.accessories), 7

G
generate_forwardmask () (in module telnetlib3.slc), 23
generate_sltctab () (in module telnetlib3.slc), 23
get_extra_info () (BaseClient method), 14
get_extra_info () (BaseServer method), 20
get_extra_info () (TelnetClient method), 9
get_extra_info () (TelnetServer method), 17
get_extra_info () (TelnetTerminalClient method), 11
get_extra_info () (TelnetWriter method), 28
get_extra_info () (TelnetWriterUnicode method), 36

H
handle_abort () (TelnetWriter method), 31
handle_abort () (TelnetWriterUnicode method), 36
handle_ao () (TelnetWriter method), 31
handle_ao () (TelnetWriterUnicode method), 37
handle_ayt () (TelnetWriter method), 31
handle_ayt () (TelnetWriterUnicode method), 37
handle_brk () (TelnetWriter method), 31
handle_brk () (TelnetWriterUnicode method), 37
handle_charset () (TelnetWriter method), 34
handle_charset () (TelnetWriterUnicode method), 37
handle_dm () (TelnetWriter method), 31
handle_dm () (TelnetWriterUnicode method), 37
handle_do () (TelnetWriter method), 34
handle_do () (TelnetWriterUnicode method), 37
handle_dont () (TelnetWriter method), 34
handle_dont () (TelnetWriterUnicode method), 37
handle_ec () (TelnetWriter method), 31
handle_ec () (TelnetWriterUnicode method), 37
handle_el () (TelnetWriter method), 31
handle_el () (TelnetWriterUnicode method), 37
handle_envir () (TelnetWriter method), 34
handle_envir () (TelnetWriterUnicode method), 38
handle_eof () (TelnetWriter method), 31
handle_eof () (TelnetWriterUnicode method), 38
handle_eor () (TelnetWriter method), 31
handle_eor () (TelnetWriterUnicode method), 38
handle_ew () (TelnetWriter method), 32
handle_ew () (TelnetWriterUnicode method), 38
handle_ga () (TelnetWriter method), 31
handle_ga () (TelnetWriterUnicode method), 38
handle_ip () (TelnetWriter method), 31
handle_ip () (TelnetWriterUnicode method), 38
handle_lnext () (TelnetWriter method), 32
handle_lnext () (TelnetWriterUnicode method), 38
handle_logout () (TelnetWriter method), 34
handle_logout () (TelnetWriterUnicode method), 38
handle_naws () (TelnetWriter method), 33
handle_naws() (TelnetWriterUnicode method), 38
handle_nop() (TelnetWriter method), 31
handle_nop() (TelnetWriterUnicode method), 38
handle rp() (TelnetWriter method), 32
handle rp() (TelnetWriterUnicode method), 38
handle_send_client_charset() (TelnetWriter method), 34
handle_send_client_charset() (TelnetWriterUnicode method), 38
handle_send_client_envir() (TelnetWriter method), 34
handle_send_client_envir() (TelnetWriterUnicode method), 38
handle_send_naws() (TelnetWriter method), 33
handle_send_naws() (TelnetWriterUnicode method), 38
handle_send_server_charset() (TelnetWriter method), 34
handle_send_server_charset() (TelnetWriterUnicode method), 38
handle_send_server_envir() (TelnetWriter method), 34
handle_send_server_envir() (TelnetWriterUnicode method), 38
handle_send_sndloc() (TelnetWriter method), 33
handle_send_sndloc() (TelnetWriterUnicode method), 38
handle_send_tspeed() (TelnetWriter method), 34
handle_send_tspeed() (TelnetWriterUnicode method), 38
handle_send_ttype() (TelnetWriter method), 33
handle_send_ttype() (TelnetWriterUnicode method), 39
handle_xdisploc() (TelnetWriter method), 33
handle_xdisploc() (TelnetWriterUnicode method), 39
handle_sndloc() (TelnetWriter method), 33
handle_sndloc() (TelnetWriterUnicode method), 39
handle_subnegotiation() (TelnetWriter method), 35
handle_subnegotiation() (TelnetWriterUnicode method), 39
handle_susp() (TelnetWriter method), 31
handle_susp() (TelnetWriterUnicode method), 39
handle_tm() (TelnetWriter method), 32
handle_tm() (TelnetWriterUnicode method), 39
handle_tspeed() (TelnetWriter method), 34
handle_tspeed() (TelnetWriterUnicode method), 39
handle_ttype() (TelnetWriter method), 33
handle_ttype() (TelnetWriterUnicode method), 39
handle_will() (TelnetWriter method), 34
handle_will() (TelnetWriterUnicode method), 39
handle_wont() (TelnetWriterUnicode method), 35

iac() (TelnetWriter method), 29
iac() (TelnetWriterUnicode method), 40
iac_received (TelnetWriter attribute), 27
iac_received (TelnetWriterUnicode attribute), 40
done (BaseClient attribute), 14
done (BaseServer attribute), 20
done (TelnetClient attribute), 9
done (TelnetServer attribute), 18
done (TelnetTerminalClient attribute), 11
inbinary (TelnetWriter attribute), 28
inbinary (TelnetWriterUnicode attribute), 40
is_closing() (TelnetWriter method), 28
is_closing() (TelnetWriterUnicode method), 40
is_oob (TelnetWriter attribute), 29
is_oob (TelnetWriterUnicode attribute), 40

level (SLC attribute), 21, 22
level (SLC_nosupport attribute), 23
1flow (TelnetWriter attribute), 27
1flow (TelnetWriterUnicode attribute), 40
Linemode (class in telnetlib3.slc), 24
linemode (TelnetWriter attribute), 29
linemode (TelnetWriterUnicode attribute), 40
lit echo (Linemode attribute), 24
local (Linemode attribute), 24
local_option (TelnetWriter attribute), 28

make_logger() (in module telnetlib3.accessories), 7
make_reader_task() (in module telnetlib3.accessories), 7
mode (TelnetWriter attribute), 29
mode (TelnetWriterUnicode attribute), 40

name_command() (in module telnetlib3.telopt), 43
name_commands() (in module telnetlib3.telopt), 43
name_scl_command() (in module telnetlib3.slc), 24
name_unicode() (in module telnetlib3.accessories), 7
negotiation should advance (BaseServer method), 20
negotiation should advance (TelnetServer method), 18
nosupport (SLC attribute), 21, 22
nosupport (SLC_nosupport attribute), 23

O
on_charset() (TelnetServer method), 17
on_environ() (TelnetServer method), 17
on_naws() (TelnetServer method), 16
on_request_charset() (TelnetServer method), 17
on_request_environ() (TelnetServer method), 17
on_timeout() (TelnetServer method), 16
on_tspeed() (TelnetServer method), 17
on_xdisploc() (TelnetServer method), 17
open_connection() (in module telnetlib3.client), 12
on_charset() (TelnetServer method), 17
on_environ() (TelnetServer method), 17
on_naws() (TelnetServer method), 16
on_request_charset() (TelnetServer method), 17
on_request_environ() (TelnetServer method), 17
on_timeout() (TelnetServer method), 16
on_tspeed() (TelnetServer method), 17
on_xdisploc() (TelnetServer method), 17

P
parse_server_args() (in module telnetlib3.server), 18
pause_writing() (BaseClient method), 14
pause_writing() (BaseServer method), 21
pause_writing() (TelnetClient method), 10
pause_writing() (TelnetServer method), 18
pause_writing() (TelnetTerminalClient method), 11
pending_option (TelnetWriter attribute), 28
protocol (TelnetWriter attribute), 28
protocol (TelnetWriterUnicode attribute), 40

R
read() (TelnetReader method), 25
read() (TelnetReaderUnicode method), 26
readexactly() (TelnetReader method), 25
readexactly() (TelnetReaderUnicode method), 26
readline() (TelnetReader method), 25
readline() (TelnetReaderUnicode method), 26
readuntil() (TelnetReader method), 25
readuntil() (TelnetReaderUnicode method), 26
remote (Linemode attribute), 24
remote_option (TelnetWriter attribute), 28
repr_mapping() (in module telnetlib3.accessories), 7
request_charset() (TelnetWriter method), 30
request_charset() (TelnetWriter method), 30
request_environ() (TelnetWriter method), 30
request_environ() (TelnetWriterUnicode method), 41
request_forwardmask() (TelnetWriter method), 30
request_forwardmask() (TelnetWriterUnicode method), 41
request_status() (TelnetWriter method), 30
request_status() (TelnetWriterUnicode method), 41
request_tspeed() (TelnetWriter method), 30
request_tspeed() (TelnetWriterUnicode method), 41
request_ttype() (TelnetWriter method), 30
request_ttype() (TelnetWriterUnicode method), 41
resume_writing() (BaseClient method), 15
resume_writing() (BaseServer method), 21
resume_writing() (TelnetClient method), 10
resume_writing() (TelnetServer method), 18
resume_writing() (TelnetTerminalClient method), 12

RFC
RFC 1041, 46
RFC 1043, 46
RFC 1073, 10, 13, 16, 32–34, 38, 42, 45
RFC 1079, 13, 17, 30, 32–34, 39, 41, 42, 45
RFC 1086, 13, 30, 41
RFC 1091, 33, 39, 42, 45
RFC 1096, 17, 33, 39, 40, 42, 45
RFC 1097, 46
RFC 1123, 45
RFC 1143, 46
RFC 1184, 21, 45
RFC 1205, 46
RFC 137, 46
RFC 1372, 30, 41, 45
RFC 139, 46
RFC 1408, 33, 42, 45
RFC 1411, 46
RFC 1412, 46
RFC 1416, 46
RFC 1492, 46
RFC 1571, 45
RFC 1572, 10, 12, 17–19, 30, 34, 35, 38, 41, 45
RFC 1672, 13
RFC 2066, 12, 17, 18, 30, 33–35, 37, 38, 41–43, 46
RFC 2217, 46
RFC 318, 46
RFC 328, 46
RFC 340, 46
RFC 393, 47
RFC 435, 47
RFC 513, 47
RFC 529, 47
RFC 559, 47
RFC 563, 47
RFC 593, 47
RFC 595, 47
RFC 596, 47
RFC 652, 47

Index
RFC 653, 47
RFC 654, 47
RFC 655, 47
RFC 656, 47
RFC 657, 47
RFC 658, 47
RFC 659, 47
RFC 698, 47
RFC 701, 47
RFC 702, 47
RFC 703, 47
RFC 718, 47
RFC 719, 47
RFC 726, 47
RFC 727, 34, 38, 45
RFC 728, 47
RFC 729, 47
RFC 731, 47
RFC 732, 47
RFC 734, 47
RFC 735, 47
RFC 736, 47
RFC 749, 47
RFC 779, 32, 33, 38, 39, 42, 45
RFC 818, 47
RFC 854, 25, 45, 46
RFC 855, 45
RFC 856, 9, 11, 16, 19, 29, 40, 45
RFC 857, 45
RFC 858, 45
RFC 859, 30, 41, 45
RFC 860, 45
RFC 861, 46
RFC 885, 30, 41, 45
RFC 927, 46
RFC 930, 13, 17, 19, 30, 41
RFC 933, 46
RFC 946, 46
RFC 97, 46

run_server() (in module telnetlib3.server), 19

S

send_charset() (TelnetClient method), 8
send_charset() (TelnetTerminalClient method), 12
send_env() (TelnetClient method), 8
send_env() (TelnetTerminalClient method), 10
send_eor() (TelnetWriter method), 30
send_eor() (TelnetWriterUnicode method), 41
send_ga() (TelnetWriter method), 30
send_ga() (TelnetWriterUnicode method), 41
send_iac() (TelnetWriter method), 29
send_iac() (TelnetWriterUnicode method), 41
send_lineflow_mode() (TelnetWriterUnicode method), 41
send_lineflow_mode() (TelnetWriter method), 30
send_linemode() (TelnetWriter method), 30
send_linemode() (TelnetWriterUnicode method), 41
send_naws() (TelnetClient method), 8
send_naws() (TelnetTerminalClient method), 10
send_tspeed() (TelnetClient method), 8
send_tspeed() (TelnetTerminalClient method), 12
send_ttype() (TelnetClient method), 8
send_ttype() (TelnetTerminalClient method), 12
send_xdisploc() (TelnetClient method), 8
send_xdisploc() (TelnetTerminalClient method), 12
server (TelnetWriter attribute), 28
server (TelnetWriterUnicode attribute), 42
set_exception() (TelnetReader method), 24
set_exception() (TelnetReaderUnicode method), 27
set_ext_callback() (TelnetWriter method), 33
set_ext_callback() (TelnetWriterUnicode method), 42
set_ext_send_callback() (TelnetWriter method), 32
set_ext_send_callback() (TelnetWriterUnicode method), 42
set_flag() (SLC method), 22
set_flag() (SLC_nosupport method), 23
set_iac_callback() (TelnetWriter method), 31
set_iac_callback() (TelnetWriterUnicode method), 43
set_mask() (SLC method), 22
set_mask() (SLC_nosupport method), 23
set_slc_callback() (TelnetWriter method), 32
set_slc_callback() (TelnetWriterUnicode method), 43
set_timeout() (TelnetServer method), 16
set_transport() (TelnetReader method), 24
set_transport() (TelnetReaderUnicode method), 27
set_value() (SLC method), 22
set_value() (SLC_nosupport method), 23
SLC (class in telnetlib3.slc), 21, 22
SLC_nosupport (class in telnetlib3.slc), 23
slc_received (TelnetWriter attribute), 27
slc_received (TelnetWriterUnicode attribute), 43
slc_simulated (TelnetWriter attribute), 27
slc_simulated (TelnetWriterUnicode attribute), 43
slctab (TelnetWriter attribute), 28
snoop() (in module telnetlib3.slc), 23
soft_tab (Linemode attribute), 24

T
telnet_client_shell() (in module telnetlib3.client_shell), 15
TelnetServer
(class in telnetlib3.server), 15
TelnetTerminalClient
(class in telnetlib3.client), 10
TelnetTerminalReader
(class in telnetlib3.stream_reader), 24
TelnetTerminalWriter
(class in telnetlib3.stream_writer), 27
TelnetTerminalWriterUnicode
(class in telnetlib3.stream_writer), 35
timeout_connection
(BaseServer method), 19
timeout_connection
(TelnetServer method), 18
transport
(TelnetWriter attribute), 28
transport
(TelnetWriterUnicode attribute), 43
trapsig
(Linemode attribute), 24
TTYTYPE_LOOPMAX
(TelnetServer attribute), 15

V
variable
(SLC attribute), 22
variable
(SLC_nosupport attribute), 23

W
will_echo
(TelnetWriter attribute), 29
will_echo
(TelnetWriterUnicode attribute), 43
write
(TelnetWriter method), 28
write
(TelnetWriterUnicode method), 35
write_eof
(TelnetWriter method), 28
write_eof
(TelnetWriterUnicode method), 43
writelines
(TelnetWriter method), 28
writelines
(TelnetWriterUnicode method), 36

X
xon_any
(TelnetWriter attribute), 27
xon_any
(TelnetWriterUnicode attribute), 43