Telnetlib3 Documentation

Release 1.0.3

Jeff Quast

Nov 28, 2022

CONTENTS

Python 3 asyncio Telnet server and client Protocol library.

Contents:

CHAPTER

ONE

INTRODUCTION

telnetlib3 is a Telnet Client and Server library for python. This project requires python 3.3 and later, using the asyncio module.

1.1 Quick Example

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

```
import asyncio, telnetlib3

@asyncio.coroutine
def shell(reader, writer):
    writer.write('\r\nWould you like to play a game? ')
    inp = yield from reader.read(1)
    if inp:
        writer.echo(inp)
        writer.write('\r\nThey say the only way to win '
                     'is to not play at all.\r\n')
        yield from 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:

```
import asyncio, telnetlib3

@asyncio.coroutine
def shell(reader, writer):
    while True:
        # read stream until '?' mark is found
        outp = yield from reader.read(1024)
        if not outp:
            # End of File
            break
        elif '?' in outp:
            # reply all questions with 'y'.
            writer.write('y')

        # display all server output
```

(continues on next page)

(continued from previous page)

```
        print(outp, flush=True)

    # EOF
    print()

loop = asyncio.get_event_loop()
coro = telnetlib3.open_connection('localhost', 6023, shell=shell)
reader, writer = loop.run_until_complete(coro)
loop.run_until_complete(writer.protocol.waiter_closed)
```

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>

tel:sh> writer
<TelnetWriter server mode:local -lineflow +xon_any +slc_sim client-will:NAWS,
↪NEW_ENVIRON,TTYPE>
```

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:

- rfc-727, “Telnet Logout Option,” Apr 1977.
- rfc-779, “Telnet Send-Location Option”, Apr 1981.
- rfc-854, “Telnet Protocol Specification”, May 1983.
- rfc-855, “Telnet Option Specifications”, May 1983.
- rfc-856, “Telnet Binary Transmission”, May 1983.
- rfc-857, “Telnet Echo Option”, May 1983.
- rfc-858, “Telnet Suppress Go Ahead Option”, May 1983.
- rfc-859, “Telnet Status Option”, May 1983.
- rfc-860, “Telnet Timing mark Option”, May 1983.
- rfc-885, “Telnet End of Record Option”, Dec 1983.
- rfc-1073, “Telnet Window Size Option”, Oct 1988.
- rfc-1079, “Telnet Terminal Speed Option”, Dec 1988.
- rfc-1091, “Telnet Terminal-Type Option”, Feb 1989.
- rfc-1096, “Telnet X Display Location Option”, Mar 1989.
- rfc-1123, “Requirements for Internet Hosts”, Oct 1989.
- rfc-1184, “Telnet Linemode Option (extended options)”, Oct 1990.
- rfc-1372, “Telnet Remote Flow Control Option”, Oct 1992.
- rfc-1408, “Telnet Environment Option”, Jan 1993.
- rfc-1571, “Telnet Environment Option Interoperability Issues”, Jan 1994.
- rfc-1572, “Telnet Environment Option”, Jan 1994.
- rfc-2066, “Telnet Charset Option”, Jan 1997.

1.4 Further Reading

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

CHAPTER

TWO

API

2.1 accessories

Accessory functions.

encoding_from_lang (*lang*)

Parse encoding from LANG environment value.

Example:

```
>>> 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:

```
>>> 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*)

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.

Class initializer.

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 Union[str, None]

send_naws()

Callback for responding to NAWS requests.

Return type (int, int)

Returns client window size as (rows, columns).

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

class TelnetTerminalClient (*term='unknown'*, *cols=80*, *rows=25*, *tspeed=38400, 38400*, *xdisploc=''*, **args*, ***kwargs*)
Telnet client for sessions with a network virtual terminal (NVT).

Class initializer.

send_naws ()
Callback replies to request for window size, NAWS **RFC 1073**.

Return type (int, int)

Returns window dimensions by lines and columns

send_env (*keys*)
Callback replies to request for env values, NEW_ENVIRON **RFC 1572**.

Return type dict

Returns super class value updated with window LINES and COLUMNS.

open_connection (*host=None*, *port=23*, ***, *client_factory=None*, *loop=None*, *family=0*, *flags=0*, *local_addr=None*, *log=None*, *encoding='utf8'*, *encoding_errors='replace'*, *force_binary=False*, *term='unknown'*, *cols=80*, *rows=25*, *tspeed=38400, 38400*, *xdisploc=''*, *shell=None*, *connect_minwait=2.0*, *connect_maxwait=3.0*, *waiter_closed=None*, *_waiter_connected=None*, *limit=None*)
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.
- **loop** (*asyncio.AbstractEventLoop*) – set the event loop to use. The return value of `asyncio.get_event_loop()` is used when unset.
- **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()`.

- **log** (*logging.Logger*) – target logger, if None is given, one is created using the namespace `'telnetlib3.server'`.
- **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.
- **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** (*Callable*) – A `asyncio.coroutine()` 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 and failing to reply to pending negotiations, the shell continues anyway after the greater of this value or `connect_minwait` elapsed.
- **force_binary** (*bool*) – When `True`, the encoding specified is used for both directions even when failing `BINARY` negotiation, **RFC 856**. This parameter has no effect when `encoding=False`.
- **encoding_errors** (*str*) – Same meaning as `codecs.Codec.encode()`.
- **connect_minwait** – XXX
- **connect_maxwait** – 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.

This function is a `coroutine()`.

2.3 client_base

Module provides class BaseClient.

class BaseClient(*shell=None, log=None, loop=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

encoding for new connections

connect_minwait

minimum duration for `check_negotiation()`.

connect_maxwait

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.

connection_made(*transport*)

Called when a connection is made.

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

data_received(*data*)

Process bytes received by transport.

property duration

Time elapsed since client connected, in seconds as float.

property 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.

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).

This function is a `coroutine()`.

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*)

Telnet Server protocol performing common negotiation.

Class initializer.

`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 must 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 `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 affirmatively acknowledged.

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.

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 client end.
- **incoming** (*bool*) – Whether the return value is suitable for decoding bytes received from 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

set_timeout(*duration=- 1*)

Restart or unset timeout for client.

Parameters duration (*int*) – When specified as a positive integer, schedules Future for callback of `on_timeout()`. When `-1`, the value of `self.get_extra_info('timeout')` is used. When non-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 `set_timeout()` with `duration` value of `0` or value of the same for keyword argument `timeout`.

on_naws(*rows*, *cols*)

Callback receives NAWS response, **RFC 1073**.

Parameters

- **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:

```
['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:

```
['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(*ttype*)
Callback for TTYPE response, **RFC 930**.

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

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 completes, receiving arguments `(reader, writer)`. The reader is a *`TelnetReader`* instance, the writer is a *`TelnetWriter`* instance.
- **`log`** (*`logging.Logger`*) – target logger, if None is given, one is created using the namespace `'telnetlib3.server'`.
- **`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()`.

`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.

class BaseServer(*shell=None*, *log=None*, *loop=None*, *_waiter_connected=None*, *_waiter_closed=None*, *encoding='utf8'*, *encoding_errors='strict'*, *force_binary=False*, *connect_maxwait=4.0*, *limit=None*)

Base Telnet Server Protocol.

Class initializer.

connect_maxwait

maximum duration for `check_negotiation()`.

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.

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.

property duration

Time elapsed since client connected, in seconds as float.

property idle

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

get_extra_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 affirmatively acknowledged.

encoding(*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.

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.

This function is a `coroutine()`.

2.8 slc

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

generate_slctab(*tabset={b'\x01': <telnetlib3.slc.SLC object>, b'\x02': <telnetlib3.slc.SLC object>, b'\x03': <telnetlib3.slc.SLC object>, b'\x04': <telnetlib3.slc.SLC object>, b'\x05': <telnetlib3.slc.SLC object>, b'\x06': <telnetlib3.slc.SLC object>, b'\x07': <telnetlib3.slc.SLC object>, b'\x08': <telnetlib3.slc.SLC object>, b'\t': <telnetlib3.slc.SLC object>, b'\n': <telnetlib3.slc.SLC object>, b'\x0b': <telnetlib3.slc.SLC object>, b'\x0c': <telnetlib3.slc.SLC object>, b'\r': <telnetlib3.slc.SLC object>, b'\x0e': <telnetlib3.slc.SLC object>, b'\x0f': <telnetlib3.slc.SLC object>, b'\x10': <telnetlib3.slc.SLC object>, b'\x11': <telnetlib3.slc.SLC_nosupport 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.

class Linemode(*mask=b'\x00'*)
A mask of `LMODE_MODE_LOCAL` means that all line editing is performed on the client side (default). A mask of theNULL () 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`.

property local
True if linemode is local.

property remote
True if linemode is remote.

property trapsig
True if signals are trapped by client.

property ack
Returns True if mode has been acknowledged.

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

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

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.

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

class Forwardmask(*value*, *ack=False*)
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(**args*, ***kwargs*)
A reader interface for the telnet protocol.

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.

Given stream	readline() yields
`--\r\x00---`	`--\r`, `---` …
`--\r\n---`	`--\r\n`, `---` …
`--\n---`	`--\n`, `---` …
`--\r---`	`--\r`, `---` …

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

This method is a `coroutine()`.

class TelnetReaderUnicode(*fn_encoding*, ***, *limit=65536*, *loop=None*, *encoding_errors='replace'*)

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.

decode(*buf*, *final=False*)

Decode bytes `buf` using preferred encoding.

readline()

Read one line.

See ancestor method, *readline()* for details.

This method is a `coroutine()`.

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

This method is a `coroutine()`.

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

This method is a `coroutine()`.

2.10 stream_writer

Module provides `TelnetWriter` and `TelnetWriterUnicode`.

class TelnetWriter(*transport*, *protocol*, ***, *client=False*, *server=False*, *reader=None*, *loop=None*, *log=None*)

A writer interface for the telnet protocol.

Telnet IAC Interpreter.

Almost all negotiation actions are performed through the writer interface, as any action requires writing bytes to the underling 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.
- **log** (*logging.Logger*) – target logger, if None is given, one is created using the namespace `'telnetlib3.stream_writer'`.
- **loop** (*asyncio.AbstractEventLoop*) – set the event loop to use. The return value of `asyncio.get_event_loop()` is used when unset.

byte_count = 0

Total bytes sent to `feed_byte()`

lflow = True

Whether flow control is enabled.

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.

iac_received = None

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

cmd_received = None

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

slc_received = None

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

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.

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

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
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
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
SLC Tab (SLC Functions and their support level, and ascii value)

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.

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.

property protocol
The protocol attached to this stream.

property server
Whether this stream is of the server's point of view.

property client
Whether this stream is of the client's point of view.

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

property 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.

property 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.

`property 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).

`property is_oob`

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

`property 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_EOR, 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 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**.

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**.
- `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**.

`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.

class `TelnetWriterUnicode`(*transport*, *protocol*, *fn_encoding*, ***, *encoding_errors='strict'*, ***kwds*)
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**.

A writer interface for the telnet protocol.

Telnet IAC Interpreter.

Almost all negotiation actions are performed through the writer interface, as any action requires writing bytes to the underling 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.
- **`log`** (*`logging.Logger`*) – target logger, if None is given, one is created using the namespace `'telnetlib3.stream_writer'`.
- **`loop`** (*`asyncio.AbstractEventLoop`*) – set the event loop to use. The return value of `asyncio.get_event_loop()` is used when unset.

`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.

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.

2.11 telopt

`name_command`(*byte*)

Return string description for (maybe) telnet command byte.

`name_commands`(*cmds*, *sep=' '*)

Return string description for array of (maybe) telnet command bytes.

CHAPTER

THREE

RFCS

3.1 Implemented

- **RFC 727**, “Telnet Logout Option,” Apr 1977.
- **RFC 779**, “Telnet Send-Location Option”, Apr 1981.
- **RFC 854**, “Telnet Protocol Specification”, May 1983.
- **RFC 855**, “Telnet Option Specifications”, May 1983.
- **RFC 856**, “Telnet Binary Transmission”, May 1983.
- **RFC 857**, “Telnet Echo Option”, May 1983.
- **RFC 858**, “Telnet Suppress Go Ahead Option”, May 1983.
- **RFC 859**, “Telnet Status Option”, May 1983.
- **RFC 860**, “Telnet Timing mark Option”, May 1983.
- **RFC 885**, “Telnet End of Record Option”, Dec 1983.
- **RFC 1073**, “Telnet Window Size Option”, Oct 1988.
- **RFC 1079**, “Telnet Terminal Speed Option”, Dec 1988.
- **RFC 1091**, “Telnet Terminal-Type Option”, Feb 1989.
- **RFC 1096**, “Telnet X Display Location Option”, Mar 1989.
- **RFC 1123**, “Requirements for Internet Hosts”, Oct 1989.
- **RFC 1184**, “Telnet Linemode Option (extended options)”, Oct 1990.
- **RFC 1372**, “Telnet Remote Flow Control Option”, Oct 1992.
- **RFC 1408**, “Telnet Environment Option”, Jan 1993.
- **RFC 1571**, “Telnet Environment Option Interoperability Issues”, Jan 1994.
- **RFC 1572**, “Telnet Environment Option”, Jan 1994.
- **RFC 2066**, “Telnet Charset Option”, Jan 1997.

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 `CSI + c` and `CSI + 0c` for vt102, and `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 1097**, "Telnet Subliminal-Message Option", Apr 1989
- **RFC 1143**, "The Q Method of Implementing .. Option Negotiation", Feb 1990
- **RFC 1205**, "5250 Telnet Interface", Feb 1991
- **RFC 1411**, "Telnet Authentication: Kerberos Version 4", Jan 1993
- **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 701** August, 1974, Survey of New-Protocol Telnet Servers.
- **RFC 702** September, 1974, Survey of New-Protocol Telnet Servers.
- **RFC 703** July, 1975, 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:

- "Telnet Protocol," MIL-STD-1782, U.S. Department of Defense, May 1984.
- "Mud Terminal Type Standard," http://tintin.sourceforge.net/mtts/
- "Mud Client Protocol, Version 2.1," http://www.moo.mud.org/mcp/mcp2.html
- "Telnet Protocol in C-Kermit 8.0 and Kermit 95 2.0," http://www.columbia.edu/kermit/telnet80.html
- "Telnet Negotiation Concepts," http://lpc.psyc.eu/doc/concepts/negotiation
- "Telnet RFCs," http://www.omnifarious.org/~hopper/telnet-rfc.html"
- "Telnet Options", http://www.iana.org/assignments/telnet-options/telnet-options.xml

CHAPTER

FOUR

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

Install and run tox

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

Py.test <https://pytest.org> is the test runner. tox commands pass through positional arguments, so you may for example use *looponfailing <https://pytest.org/latest/xdist.html#running-tests-in-looponfailing-mode>* with python 3.5, stopping at the first failing test case:

```
tox -epy35 -- -fx
```

4.3 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
```

CHAPTER

FIVE

HISTORY

1.0.4

- bugfix a 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

SIX

INDEXES

- genindex
- modindex

PYTHON MODULE INDEX

INDEX