Unittest for `tcp_socket`

May 16, 2026

Contents

1 Test Information **2**

1.1 Testobject Information 2

1.2 Testdata Information 2

1.3 Testsystem Information 2

2 Statistic **2**

2.1 Test-Statistic for testrun: Library test 2

2.2 Coverage Statistic 3

A Trace for testrun: Library test **4**

B Test-Coverage **4**

B.1 tcp_socket 4

B.1.1 tcp_socket.__init__.py 4

1 Test Information

1.1 Testobject Information

The Module tcp_socket is designed to help with client / server tcp socket connections. For more Information read the documentation.

Information	
Name	tcp_socket
State	In test
Version	0.3.2
Dependencies	
	Python standards
	stringtools
	task

1.2 Testdata Information

Information	
Version	0.0.1
Testruns	Library test

1.3 Testsystem Information

System Information	
Architecture	64bit
Distribution	Debian GNU/Linux 13 trixie
Hostname	ahorn
Kernel	6.12.86+deb13-amd64 (#1 SMP PREEMPT_DYNAMIC Debian 6.12.86-1 (2026-05-08))
Machine	x86_64
Path	/home/dirk/work/unittest_collection/tcp_socket/unittest/src
System	Linux
Username	dirk

2 Statistic

2.1 Test-Statistic for testrun: Library test

Number of tests	**0**
Number of successfull tests	**0**
Number of possibly failed tests	**0**
Number of failed tests	**0**
Executionlevel	Full Test (all defined tests)
Time consumption	0.000s

2.2 Coverage Statistic

Module- or Filename	Line-Coverage	Branch-Coverage
tcp_socket	24.0%	
tcp_socket.__init__.py	24.0%	

A Trace for testrun: Library test

B Test-Coverage

B.1 tcp_socket

The line coverage for tcp_socket was 24.0%

B.1.1 tcp_socket.__init__.py

The line coverage for tcp_socket.__init__.py was 24.0%

```
"""
tcp_socket (TCP Socket)
=======================

**Author:**

* Dirk Alders <sudo-dirk@mount-mockery.de>

**Description:**

    This Module supports a client/ server tcp socket connection.

**Submodules:**

* :class:`tcp_socket.tcp_client`
* :class:`tcp_socket.tcp_client_stp`
* :class:`tcp_socket.tcp_server`
* :class:`tcp_socket.tcp_server_stp`

**Unittest:**

        See also the :download:`unittest <tcp_socket/_testresults_/unittest.pdf>` documentation.

**Module Documentation:**

"""

__VERSION__ = "0.3.2"
__DEPENDENCIES__ = ["Python standards", "stringtools", "task"]


import stringtools
import task

import logging
import socket
import time

try:
    from config import APP_NAME as ROOT_LOGGER_NAME
except ImportError:
    ROOT_LOGGER_NAME = "root"

logger = logging.getLogger(ROOT_LOGGER_NAME).getChild(__name__)

```

```
__DESCRIPTION__ = """The Module {\\tt %s} is designed to help with client / server tcp socket connections.
For more Information read the documentation.""" % __name__.replace("_", "\\_")
"""The Module Description"""


def tcp_send(host, port, data):
    """
    Send raw tcp socket data.

    :param host: The host IP for the TCP socket functionality
    :type host: str
    :param port: The port for the TCP socket functionality
    :type port: int
    :param data: The data to be send over the communication channel.
    :type data: bytes
    """
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((host, port))
        s.sendall(data)
        s.close()
    except:
        return False
    else:
        return True


def tcp_send_stp(host, port, data):
    """
    Send stp tcp socket data.

    :param host: The host IP for the TCP socket functionality
    :type host: str
    :param port: The port for the TCP socket functionality
    :type port: int
    :param data: The data to be send over the communication channel.
    :type data: bytes
    """
    return tcp_send(host, port, stringtools.stp.build_frame(data))


class tcp_base(object):
    """
    This is the base class for other classes in this module.

    :param host: The host IP for the TCP socket functionality
    :type host: str
    :param port: The port for the TCP socket functionality
    :type port: int
    :param channel_name: The name for the logging channel
    :type channel_name: str

    .. note:: This class is not designed for direct usage.
    """

    DEFAULT_CHANNEL_NAME = "all_others"

    RX_LENGTH = 0xFF
    COM_TIMEOUT = 0.5
    IS_CLIENT = False

```

```
107     def __init__(self, host, port, channel_name=None, rx_tx_log_lvl=logging.INFO):
108         self.host = host
109         self.port = port
110         self.init_channel_name(channel_name)
111         self.__rx_tx_log_lvl__ = rx_tx_log_lvl
112         self.__socket__ = None
113         self.__data_available_callback__ = None
114         self.__supress_data_available_callback__ = False
115         self.__connect_callback__ = None
116         self.__disconnect_callback__ = None
117         self.__clean_receive_buffer__()
118         self.__connection__ = None
119         self.__listening_message_displayed__ = False
120         self.__client_address__ = None
121         self.__queue__ = task.threaded_queue()
122         self.__queue__.enqueue(5, self.__receive_task__)
123         self.__queue__.run()
124 
125     def __call_data_available_callback__(self):
126         if len(self.__receive_buffer__) > 0 and not self.__supress_data_available_callback__ and
    self.__data_available_callback__ is not None:
127             self.__supress_data_available_callback__ = True
128             self.__data_available_callback__(self)
129             self.__supress_data_available_callback__ = False
130 
131     def __clean_receive_buffer__(self):
132         self.logger.debug("%s Cleaning up receive-buffer", self.__log_prefix__())
133         self.__receive_buffer__ = b""
134 
135     def __connection_lost__(self):
136         self.__connection__.close()
137         self.__connection__ = None
138         self.__client_address__ = None
139         self.logger.debug("%s Connection lost...", self.__log_prefix__())
140         if self.__disconnect_callback__ is not None:
141             self.__disconnect_callback__()
142 
143     def __del__(self):
144         self.close()
145 
146     def __log_prefix__(self):
147         return "comm-client:" if self.IS_CLIENT else "comm-server:"
148 
149     def __receive_task__(self, queue_inst):
150         if self.__connection__ is not None:
151             try:
152                 data = self.__connection__.recv(self.RX_LENGTH)
153             except socket.error as e:
154                 if e.errno != 11:
155                     raise
156                 else:
157                     time.sleep(0.05)
158             else:
159                 if len(data) > 0:
160                     self.logger.log(self.__rx_tx_log_lvl__, '%s RX <- "%s"', self.__log_prefix__
    (), stringtools.hexlify(data))
161                     self.__receive_buffer__ += data
162                 else:
163                     self.__connection_lost__()
164             self.__call_data_available_callback__()
165         else:
166             self.__connect__()
167         queue_inst.enqueue(5, self.__receive_task__)
```

```
168
169     def client_address(self):
170         """
171         This method returns the address of the connected client.
172
173         :return: The client address.
174         :rtype: str
175         """
176         return self.__client_address__[0]
177
178     def close(self):
179         """
180         This method closes the connected communication channel, if exists.
181         """
182         self.__queue__.stop()
183         self.__queue__.join()
184         if self.__connection__ is not None:
185             self.__connection_lost__()
186         if self.__socket__ is not None:
187             self.__socket__.close()
188
189     def init_channel_name(self, channel_name=None):
190         """
191         With this Method, the channel name for logging can be changed.
192
193         :param channel_name: The name for the logging channel
194         :type channel_name: str
195         """
196         if channel_name is None:
197             self.logger = logger.getChild(self.DEFAULT_CHANNEL_NAME)
198         else:
199             self.logger = logger.getChild(channel_name)
200
201     def is_connected(self):
202         """
203         With this Method the connection status can be identified.
204
205         :return: True, if a connection is established, otherwise False.
206         :rtype: bool
207         """
208         return self.__connection__ is not None
209
210     def receive(self, timeout=1, num=None):
211         """
212         This method returns received data.
213
214         :param timeout: The timeout for receiving data (at least after the timeout the method
    returns data or None).
215         :type timeout: float
216         :param num: the number of bytes to receive (use None to get all available data).
217         :type num: int
218         :return: The received data.
219         :rtype: bytes
220         """
221         rv = None
222         if self.__connection__ is not None:
223             tm = time.time()
224             while (num is not None and len(self.__receive_buffer__) < num) or (num is None and
    len(self.__receive_buffer__) < 1):
225                 if self.__connection__ is None:
226                     return None
227                 if time.time() > tm + timeout:
228                     self.logger.log(
```

```
229                         self.__rx_tx_log_lvl__,
230                         "%s TIMEOUT (%ss): Not enough data in buffer. Requested %s and buffer
    size is %d.",
231                         self.__log_prefix__(),
232                         repr(timeout),
233                         repr(num or "all"),
234                         len(self.__receive_buffer__),
235                     )
236                     return None
237                 time.sleep(0.05)
238             if num is None:
239                 rv = self.__receive_buffer__
240                 self.__clean_receive_buffer__()
241             else:
242                 rv = self.__receive_buffer__[:num]
243                 self.__receive_buffer__ = self.__receive_buffer__[num:]
244         return rv
245
246     def register_callback(self, callback):
247         """
248         This method stores the callback which is executed, if data is available. You need to
    execute :func:`receive` of this instance
249         given as first argument.
250
251         :param callback: The callback which will be executed, when data is available.
252         :type callback:
253         """
254         self.__data_available_callback__ = callback
255
256     def register_connect_callback(self, callback):
257         """
258         This method stores the callback which is executed, if a connection is established.
259
260         :param callback: The callback which will be executed, when a connection is established.
261         :type callback:
262         """
263         self.__connect_callback__ = callback
264
265     def register_disconnect_callback(self, callback):
266         """
267         This method stores the callback which is executed, after the connection is lost.
268
269         :param callback: The callback which will be executed, after the connection is lost.
270         :type callback:
271         """
272         self.__disconnect_callback__ = callback
273
274     def send(self, data, timeout=1):
275         """
276         This method sends data via the initiated communication channel.
277
278         :param data: The data to be send over the communication channel.
279         :type data: bytes
280         :param timeout: The timeout for sending data (e.g. time to establish new connection).
281         :type timeout: float
282         :return: True if data had been sent, otherwise False.
283         :rtype: bool
284         """
```

```
285         tm = time.time()
286         while time.time() - tm < timeout:
287             if self.__connection__ is not None:
288                 try:
289                     self.__connection__.sendall(data)
290                 except BlockingIOError:
291                     time.sleep(0.1)  # try again till timeout exceeds
292                 except (BrokenPipeError, ConnectionResetError) as e:
293                     self.logger.exception("%s Exception while sending data", self.__log_prefix__
    ())
294                     self.__connection_lost__()
295                     return False
296                 else:
297                     self.logger.log(self.__rx_tx_log_lvl__, '%s TX -> "%s"', self.__log_prefix__
    (), stringtools.hexlify(data))
298                     return True
299             else:
300                 time.sleep(0.1)  # give some time to establish the connection
301         self.logger.log(self.__rx_tx_log_lvl__, '%s Cound NOT send -> "%s"', self.__log_prefix__
    (), stringtools.hexlify(data))
302         return False
303 
304 
305 class tcp_server(tcp_base):
306     """
307     This class creates a tcp-server for transfering a serial stream of bytes (characters). See
    also parent :class:`tcp_base`.
308 
309     :param host: The host IP for the TCP socket functionality
310     :type host: str
311     :param port: The port for the TCP socket functionality
312     :type port: int
313     :param channel_name: The name for the logging channel
314     :type channel_name: str
315 
316     .. note:: You need a :class:`tcp_client` to communicate with the server.
317 
318     **Example:**
319 
320     .. literalinclude:: tcp_socket/_examples_/tcp_socket__tcp.py
321 
322     .. literalinclude:: tcp_socket/_examples_/tcp_socket__tcp.log
323     """
324 
325     def _connect(self):
326         if self._socket is None:
327             # Create a TCP/IP socket
328             self._socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
329             # Bind the socket to the port
330             server_address = (self.host, self.port)
331             self._socket.bind(server_address)
332             # Listen for incoming connections
333             self.__socket__.listen()
334             self.__socket__.settimeout(self.COM_TIMEOUT)
335             self.__socket__.setblocking(False)
336         if not self.__listening_message_displayed__:
337             self.logger.info("%s Server listening to %s:%d", self.__log_prefix__(), self.host,
    self.port)
338             self.__listening_message_displayed__ = True
339         try:
340             self.__connection__, self.__client_address__ = self.__socket__.accept()
341         except socket.error as e:
342             if e.errno != 11:
343                 raise
```

```
344             else:
345                 time.sleep(0.05)
346         else:
347             self.logger.debug("%s Connection established ... (from %s:%s)", self.__log_prefix__(),
    self.client_address(), self.port)
348             self.__clean_receive_buffer__()
349             self.__connection__.setblocking(False)
350             if self.__connect_callback__ is not None:
351                 self.__connect_callback__()
352
353
354 class tcp_client(tcp_base):
355     """
356     This class creates a tcp-client for transfering a serial stream of bytes (characters). See
    also parent :class:`tcp_base`.
357
358     :param host: The host IP for the TCP socket functionality
359     :type host: str
360     :param port: The port for the TCP socket functionality
361     :type port: int
362     :param channel_name: The name for the logging channel
363     :type channel_name: str
364
365     .. note:: You need a running :class:`tcp_server` listening at the given IP and Port to be
    able to communicate.
366
367     **Example:**
368
369     .. literalinclude:: tcp_socket/_examples_/tcp_socket__tcp.py
370
371     .. literalinclude:: tcp_socket/_examples_/tcp_socket__tcp.log
372     """
373
374     IS_CLIENT = True
375
376     def __connect__(self):
377         if self.__socket__ is None:
378             # Create a TCP/IP socket
379             self.__socket__ = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
380             self.__socket__.setblocking(False)
381         # Connect the socket to the port where the server is listening
382         try:
383             self.__socket__.connect((self.host, self.port))
384         except socket.error as e:
385             if e.errno == 9:
386                 self.__socket__.close()
387             elif e.errno != 115 and e.errno != 111 and e.errno != 114:
388                 raise
389             else:
390                 self.__connection__ = None
391                 time.sleep(0.05)
392         else:
393             self.logger.debug("%s Connection established ... (to %s:%s)", self.__log_prefix__(),
    self.host, self.port)
394             self.__clean_receive_buffer__()
395             self.__connection__ = self.__socket__
396             if self.__connect_callback__ is not None:
397                 self.__connect_callback__()
398
399     def __connection_lost__(self):
400         self.__socket__ = None
401         tcp_base.__connection_lost__(self)
```

```
402
403     def reconnect(self):
404         self.__connect__()
405
406
407 class tcp_base_stp(tcp_base):
408     """
409     This is the base class for other classes in this module.  See also parent :class:`tcp_base`.
410
411     :param host: The host IP for the TCP socket functionality
412     :type host: str
413     :param port: The port for the TCP socket functionality
414     :type port: int
415     :param channel_name: The name for the logging channel
416     :type channel_name: str
417
418     .. note:: This class is not designed for direct usage.
419     """
420
421     def __init__(self, host, port, channel_name=None):
422         tcp_base.__init__(self, host, port, channel_name=channel_name, rx_tx_log_lvl=logging.
    DEBUG)
423         self.__stp__ = stringtools.stp.stp()
424
425     def __clean_receive_buffer__(self):
426         self.logger.debug("%s Cleaning up receive-buffer", self.__log_prefix__())
427         self.__receive_buffer__ = []
428
429     def __receive_task__(self, queue_inst):
430         if self.__connection__ is not None:
431             try:
432                 data = self.__connection__.recv(self.RX_LENGTH)
433             except socket.error as e:
434                 if e.errno == 104:
435                     self.__connection_lost__()
436                 elif e.errno != 11:
437                     raise
438                 else:
439                     time.sleep(0.05)
440             else:
441                 if len(data) > 0:
442                     self.logger.log(self.__rx_tx_log_lvl__, '%s RX <- "%s"', self.__log_prefix__
    (), stringtools.hexlify(data))
443                     content = self.__stp__.process(data)
444                     for msg in content:
445                         self.logger.log(self.__rx_tx_log_lvl__, '%s RX  <- "%s"', self.
    __log_prefix__(), stringtools.hexlify(msg))
446                         self.__receive_buffer__.append(msg)
447                 else:
448                     self.__connection_lost__()
449             self.__call_data_available_callback__()
450         else:
451             self.__connect__()
452         queue_inst.enqueue(5, self.__receive_task__)
453
454     def receive(self, timeout=1):
455         """
456         This method returns one received messages via the initiated communication channel.
457
458         :param timeout: The timeout for receiving data (at least after the timeout the method
    returns data or None).
459         :type timeout: float
460         :return: The received data.
461         :rtype: bytes
462         """
```

```
463         try:
464             return tcp_base.receive(self, timeout=timeout, num=1)[0]
465         except TypeError:
466             return None
467 
468     def send(self, data, timeout=1):
469         """
470         This method sends one stp message via the initiated communication channel.
471 
472         :param data: The message to be send over the communication channel.
473         :type data: bytes
474         :param timeout: The timeout for sending data (e.g. time to establish new connection).
475         :type timeout: float
476         :return: True if data had been sent, otherwise False.
477         :rtype: bool
478         """
479         if tcp_base.send(self, stringtools.stp.build_frame(data), timeout=timeout):
480             self.logger.log(self.__rx_tx_log_lvl__, '%s TX -> "%s"', self.__log_prefix__(),
    stringtools.hexlify(data))
481             return True
482         else:
483             return False
484 
485 
486 class tcp_server_stp(tcp_server, tcp_base_stp):
487     """
488     This class creates a tcp-server for transfering a message. The bytes will be packed on send
    and unpacked on receive. See also parents :class:`tcp_server` and :class:`tcp_base_stp`.
489     See :mod:`stringtools.stp` for more information on packing and unpacking.
490 
491     :param host: The host IP for the TCP socket functionality
492     :type host: str
493     :param port: The port for the TCP socket functionality
494     :type port: int
495     :param channel_name: The name for the logging channel
496     :type channel_name: str
497 
498     .. note:: You need a :class:`tcp_client_stp` to communicate with the server.
499 
500     **Example:**
501 
502     .. literalinclude:: tcp_socket/_examples_/tcp_socket__stp.py
503 
504     .. literalinclude:: tcp_socket/_examples_/tcp_socket__stp.log
505     """
506 
507     pass
508 
509 
510 class tcp_client_stp(tcp_client, tcp_base_stp):
511     """
512     This class creates a tcp-client for transfering a message. The bytes will be packed on send
    and unpacked on receive. See also parents :class:`tcp_client` and :class:`tcp_base_stp`.
513     See :mod:`stringtools.stp` for more information on packing and unpacking.
514 
515     :param host: The host IP for the TCP socket functionality
516     :type host: str
517     :param port: The port for the TCP socket functionality
518     :type port: int
519     :param channel_name: The name for the logging channel
520     :type channel_name: str
521 
```

```
522     .. note:: You need a running :class:`tcp_server_stp` listening at the given IP and Port to be
        able to communicate.
523
524     **Example:**
525
526     .. literalinclude:: tcp_socket/_examples_/tcp_socket__stp.py
527
528     .. literalinclude:: tcp_socket/_examples_/tcp_socket__stp.log
529     """
530
531     pass
```