Paramiko

Release

April 29, 2015

Contents

This site covers Paramiko's usage & API documentation. For basic info on what Paramiko is, including its public changelog & how the project is maintained, please see the main project website.

CHAPTER 1

API documentation

The high-level client API starts with creation of an `SSHClient` object. For more direct control, pass a socket (or socket-like object) to a `Transport`, and use `start_server` or `start_client` to negotiate with the remote host as either a server or client.

As a client, you are responsible for authenticating using a password or private key, and checking the server's host key. (Key signature and verification is done by paramiko, but you will need to provide private keys and check that the content of a public key matches what you expected to see.)

As a server, you are responsible for deciding which users, passwords, and keys to allow, and what kind of channels to allow.

Once you have finished, either side may request flow-controlled `channels` to the other side, which are Python objects that act like sockets, but send and receive data over the encrypted session.

For details, please see the following tables of contents (which are organized by area of interest.)

1.1 Core SSH protocol classes

1.1.1 Channel

Abstraction for an SSH2 channel.

class paramiko.channel.**Channel**(*chanid*)

A secure tunnel across an SSH `Transport`. A Channel is meant to behave like a socket, and has an API that should be indistinguishable from the Python socket API.

Because SSH2 has a windowing kind of flow control, if you stop reading data from a Channel and its buffer fills up, the server will be unable to send you any more data until you read some of it. (This won't affect other channels on the same transport – all channels on a single transport are flow-controlled independently.) Similarly, if the server isn't reading data you send, calls to `send` may block, unless you set a timeout. This is exactly like a normal network socket, so it shouldn't be too surprising.

__init__(*chanid*)

Create a new channel. The channel is not associated with any particular session or `Transport` until the Transport attaches it. Normally you would only call this method from the constructor of a subclass of `Channel`.

Parameters **chanid** (*int*) – the ID of this channel, as passed by an existing `Transport`.

__repr__()

Return a string representation of this object, for debugging.

__weakref__
list of weak references to the object (if defined)

close()
Close the channel. All future read/write operations on the channel will fail. The remote end will receive no more data (after queued data is flushed). Channels are automatically closed when their `Transport` is closed or when they are garbage collected.

exec_command(*command*)
Execute a command on the server. If the server allows it, the channel will then be directly connected to the stdin, stdout, and stderr of the command being executed.

When the command finishes executing, the channel will be closed and can't be reused. You must open a new channel if you wish to execute another command.

Parameters **command** (*str*) – a shell command to execute.

Raises SSHException if the request was rejected or the channel was closed

exit_status_ready()
Return true if the remote process has exited and returned an exit status. You may use this to poll the process status if you don't want to block in `recv_exit_status`. Note that the server may not return an exit status in some cases (like bad servers).

Returns `True` if `recv_exit_status` will return immediately, else `False`.

New in version 1.7.3.

fileno()
Returns an OS-level file descriptor which can be used for polling, but but not for reading or writing. This is primarily to allow Python's `select` module to work.

The first time `fileno` is called on a channel, a pipe is created to simulate real OS-level file descriptor (FD) behavior. Because of this, two OS-level FDs are created, which will use up FDs faster than normal. (You won't notice this effect unless you have hundreds of channels open at the same time.)

Returns an OS-level file descriptor (`int`)

Warning: This method causes channel reads to be slightly less efficient.

get_id()
Return the `int` ID # for this channel.

The channel ID is unique across a `Transport` and usually a small number. It's also the number passed to `ServerInterface.check_channel_request` when determining whether to accept a channel request in server mode.

get_name()
Get the name of this channel that was previously set by `set_name`.

get_pty(*term='vt100'*, *width=80*, *height=24*, *width_pixels=0*, *height_pixels=0*)
Request a pseudo-terminal from the server. This is usually used right after creating a client channel, to ask the server to provide some basic terminal semantics for a shell invoked with `invoke_shell`. It isn't necessary (or desirable) to call this method if you're going to exectue a single command with `exec_command`.

Parameters

- **term** (*str*) – the terminal type to emulate (for example, `'vt100'`)
- **width** (*int*) – width (in characters) of the terminal screen

- **height** (*int*) – height (in characters) of the terminal screen
- **width_pixels** (*int*) – width (in pixels) of the terminal screen
- **height_pixels** (*int*) – height (in pixels) of the terminal screen

Raises SSHException if the request was rejected or the channel was closed

get_transport()
Return the `Transport` associated with this channel.

getpeername()
Return the address of the remote side of this Channel, if possible.

This simply wraps `Transport.getpeername`, used to provide enough of a socket-like interface to allow asyncore to work. (asyncore likes to call `'getpeername'`.)

gettimeout()
Returns the timeout in seconds (as a float) associated with socket operations, or `None` if no timeout is set. This reflects the last call to `setblocking` or `settimeout`.

invoke_shell()
Request an interactive shell session on this channel. If the server allows it, the channel will then be directly connected to the stdin, stdout, and stderr of the shell.

Normally you would call `get_pty` before this, in which case the shell will operate through the pty, and the channel will be connected to the stdin and stdout of the pty.

When the shell exits, the channel will be closed and can't be reused. You must open a new channel if you wish to open another shell.

Raises SSHException if the request was rejected or the channel was closed

invoke_subsystem(*subsystem*)
Request a subsystem on the server (for example, `sftp`). If the server allows it, the channel will then be directly connected to the requested subsystem.

When the subsystem finishes, the channel will be closed and can't be reused.

Parameters **subsystem** (*str*) – name of the subsystem being requested.

Raises SSHException if the request was rejected or the channel was closed

makefile(**params*)
Return a file-like object associated with this channel. The optional `mode` and `bufsize` arguments are interpreted the same way as by the built-in `file()` function in Python.

Returns `ChannelFile` object which can be used for Python file I/O.

makefile_stderr(**params*)
Return a file-like object associated with this channel's stderr stream. Only channels using `exec_command` or `invoke_shell` without a pty will ever have data on the stderr stream.

The optional `mode` and `bufsize` arguments are interpreted the same way as by the built-in `file()` function in Python. For a client, it only makes sense to open this file for reading. For a server, it only makes sense to open this file for writing.

Returns `ChannelFile` object which can be used for Python file I/O.

New in version 1.1.

recv(*nbytes*)
Receive data from the channel. The return value is a string representing the data received. The maximum amount of data to be received at once is specified by `nbytes`. If a string of length zero is returned, the channel stream has closed.

Parameters **nbytes** (*int*) – maximum number of bytes to read.

Returns received data, as a `str`

Raises **socket.timeout** if no data is ready before the timeout set by `settimeout`.

recv_exit_status()
Return the exit status from the process on the server. This is mostly useful for retrieving the results of an `exec_command`. If the command hasn't finished yet, this method will wait until it does, or until the channel is closed. If no exit status is provided by the server, -1 is returned.

Returns the exit code (as an `int`) of the process on the server.

New in version 1.2.

recv_ready()
Returns true if data is buffered and ready to be read from this channel. A `False` result does not mean that the channel has closed; it means you may need to wait before more data arrives.

Returns `True` if a `recv` call on this channel would immediately return at least one byte; `False` otherwise.

recv_stderr(*nbytes*)
Receive data from the channel's stderr stream. Only channels using `exec_command` or `invoke_shell` without a pty will ever have data on the stderr stream. The return value is a string representing the data received. The maximum amount of data to be received at once is specified by `nbytes`. If a string of length zero is returned, the channel stream has closed.

Parameters **nbytes** (*int*) – maximum number of bytes to read.

Returns received data as a `str`

Raises **socket.timeout** if no data is ready before the timeout set by `settimeout`.

New in version 1.1.

recv_stderr_ready()
Returns true if data is buffered and ready to be read from this channel's stderr stream. Only channels using `exec_command` or `invoke_shell` without a pty will ever have data on the stderr stream.

Returns `True` if a `recv_stderr` call on this channel would immediately return at least one byte; `False` otherwise.

New in version 1.1.

request_forward_agent(*handler*)
Request for a forward SSH Agent on this channel. This is only valid for an ssh-agent from OpenSSH !!!

Parameters **handler** (*function*) – a required handler to use for incoming SSH Agent connections

Returns True if we are ok, else False (at that time we always return ok)

Raises SSHException in case of channel problem.

request_x11(*screen_number=0*, *auth_protocol=None*, *auth_cookie=None*, *single_connection=False*, *handler=None*)
Request an x11 session on this channel. If the server allows it, further x11 requests can be made from the server to the client, when an x11 application is run in a shell session.

From RFC4254:

```
It is RECOMMENDED that the 'x11 authentication cookie' that is
sent be a fake, random cookie, and that the cookie be checked and
replaced by the real cookie when a connection request is received.
```

If you omit the auth_cookie, a new secure random 128-bit value will be generated, used, and returned. You will need to use this value to verify incoming x11 requests and replace them with the actual local x11 cookie (which requires some knowledge of the x11 protocol).

If a handler is passed in, the handler is called from another thread whenever a new x11 connection arrives. The default handler queues up incoming x11 connections, which may be retrieved using `Transport.accept`. The handler's calling signature is:

```
handler(channel: Channel, (address: str, port: int))
```

Parameters

- **screen_number** (*int*) – the x11 screen number (0, 10, etc.)
- **auth_protocol** (*str*) – the name of the X11 authentication method used; if none is given, `"MIT-MAGIC-COOKIE-1"` is used
- **auth_cookie** (*str*) – hexadecimal string containing the x11 auth cookie; if none is given, a secure random 128-bit value is generated
- **single_connection** (*bool*) – if True, only a single x11 connection will be forwarded (by default, any number of x11 connections can arrive over this session)
- **handler** (*function*) – an optional handler to use for incoming X11 connections

Returns the auth_cookie used

resize_pty(*width=80*, *height=24*, *width_pixels=0*, *height_pixels=0*)

Resize the pseudo-terminal. This can be used to change the width and height of the terminal emulation created in a previous `get_pty` call.

Parameters

- **width** (*int*) – new width (in characters) of the terminal screen
- **height** (*int*) – new height (in characters) of the terminal screen
- **width_pixels** (*int*) – new width (in pixels) of the terminal screen
- **height_pixels** (*int*) – new height (in pixels) of the terminal screen

Raises SSHException if the request was rejected or the channel was closed

send(*s*)

Send data to the channel. Returns the number of bytes sent, or 0 if the channel stream is closed. Applications are responsible for checking that all data has been sent: if only some of the data was transmitted, the application needs to attempt delivery of the remaining data.

Parameters **s** (*str*) – data to send

Returns number of bytes actually sent, as an `int`

Raises **socket.timeout** if no data could be sent before the timeout set by `settimeout`.

send_exit_status(*status*)

Send the exit status of an executed command to the client. (This really only makes sense in server mode.) Many clients expect to get some sort of status code back from an executed command after it completes.

Parameters **status** (*int*) – the exit code of the process

New in version 1.2.

send_ready()

Returns true if data can be written to this channel without blocking. This means the channel is either closed

(so any write attempt would return immediately) or there is at least one byte of space in the outbound buffer. If there is at least one byte of space in the outbound buffer, a `send` call will succeed immediately and return the number of bytes actually written.

Returns `True` if a `send` call on this channel would immediately succeed or fail

send_stderr(*s*)

Send data to the channel on the "stderr" stream. This is normally only used by servers to send output from shell commands – clients won't use this. Returns the number of bytes sent, or 0 if the channel stream is closed. Applications are responsible for checking that all data has been sent: if only some of the data was transmitted, the application needs to attempt delivery of the remaining data.

Parameters **s** (*str*) – data to send.

Returns number of bytes actually sent, as an `int`.

Raises **socket.timeout** if no data could be sent before the timeout set by `settimeout`.

New in version 1.1.

sendall(*s*)

Send data to the channel, without allowing partial results. Unlike `send`, this method continues to send data from the given string until either all data has been sent or an error occurs. Nothing is returned.

Parameters **s** (*str*) – data to send.

Raises

- **socket.timeout** – if sending stalled for longer than the timeout set by `settimeout`.
- **socket.error** – if an error occurred before the entire string was sent.

Note: If the channel is closed while only part of the data has been sent, there is no way to determine how much data (if any) was sent. This is irritating, but identically follows Python's API.

---

sendall_stderr(*s*)

Send data to the channel's "stderr" stream, without allowing partial results. Unlike `send_stderr`, this method continues to send data from the given string until all data has been sent or an error occurs. Nothing is returned.

Parameters **s** (*str*) – data to send to the client as "stderr" output.

Raises

- **socket.timeout** – if sending stalled for longer than the timeout set by `settimeout`.
- **socket.error** – if an error occurred before the entire string was sent.

New in version 1.1.

set_combine_stderr(*combine*)

Set whether stderr should be combined into stdout on this channel. The default is `False`, but in some cases it may be convenient to have both streams combined.

If this is `False`, and `exec_command` is called (or `invoke_shell` with no pty), output to stderr will not show up through the `recv` and `recv_ready` calls. You will have to use `recv_stderr` and `recv_stderr_ready` to get stderr output.

If this is `True`, data will never show up via `recv_stderr` or `recv_stderr_ready`.

Parameters **combine** (*bool*) – `True` if stderr output should be combined into stdout on this channel.

Returns the previous setting (a `bool`).

New in version 1.1.

set_name(*name*)
Set a name for this channel. Currently it's only used to set the name of the channel in logfile entries. The name can be fetched with the `get_name` method.

Parameters **name** (*str*) – new channel name

setblocking(*blocking*)
Set blocking or non-blocking mode of the channel: if `blocking` is 0, the channel is set to non-blocking mode; otherwise it's set to blocking mode. Initially all channels are in blocking mode.

In non-blocking mode, if a `recv` call doesn't find any data, or if a `send` call can't immediately dispose of the data, an error exception is raised. In blocking mode, the calls block until they can proceed. An EOF condition is considered "immediate data" for `recv`, so if the channel is closed in the read direction, it will never block.

`chan.setblocking(0)` is equivalent to `chan.settimeout(0)`; `chan.setblocking(1)` is equivalent to `chan.settimeout(None)`.

Parameters **blocking** (*int*) – 0 to set non-blocking mode; non-0 to set blocking mode.

settimeout(*timeout*)
Set a timeout on blocking read/write operations. The `timeout` argument can be a nonnegative float expressing seconds, or `None`. If a float is given, subsequent channel read/write operations will raise a timeout exception if the timeout period value has elapsed before the operation has completed. Setting a timeout of `None` disables timeouts on socket operations.

`chan.settimeout(0.0)` is equivalent to `chan.setblocking(0)`; `chan.settimeout(None)` is equivalent to `chan.setblocking(1)`.

Parameters **timeout** (*float*) – seconds to wait for a pending read/write operation before raising `socket.timeout`, or `None` for no timeout.

shutdown(*how*)
Shut down one or both halves of the connection. If `how` is 0, further receives are disallowed. If `how` is 1, further sends are disallowed. If `how` is 2, further sends and receives are disallowed. This closes the stream in one or both directions.

Parameters **how** (*int*) –

0 (stop receiving), 1 (stop sending), or 2 (stop receiving and sending).

shutdown_read()
Shutdown the receiving side of this socket, closing the stream in the incoming direction. After this call, future reads on this channel will fail instantly. This is a convenience method, equivalent to `shutdown(0)`, for people who don't make it a habit to memorize unix constants from the 1970s.

New in version 1.2.

shutdown_write()
Shutdown the sending side of this socket, closing the stream in the outgoing direction. After this call, future writes on this channel will fail instantly. This is a convenience method, equivalent to `shutdown(1)`, for people who don't make it a habit to memorize unix constants from the 1970s.

New in version 1.2.

class `paramiko.channel.`**ChannelFile**(*channel*, *mode='r'*, *bufsize=-1*)
A file-like wrapper around `Channel`. A ChannelFile is created by calling `Channel.makefile`.

> **Warning:** To correctly emulate the file object created from a socket's `makefile` method, a `Channel` and its `ChannelFile` should be able to be closed or garbage-collected independently. Currently, closing the `ChannelFile` does nothing but flush the buffer.

__repr__()
Returns a string representation of this object, for debugging.

1.1.2 Client

SSH client & key policies

class paramiko.client.**AutoAddPolicy**
Policy for automatically adding the hostname and new host key to the local `HostKeys` object, and saving it. This is used by `SSHClient`.

class paramiko.client.**MissingHostKeyPolicy**
Interface for defining the policy that `SSHClient` should use when the SSH server's hostname is not in either the system host keys or the application's keys. Pre-made classes implement policies for automatically adding the key to the application's `HostKeys` object (`AutoAddPolicy`), and for automatically rejecting the key (`RejectPolicy`).

This function may be used to ask the user to verify the key, for example.

__weakref__
list of weak references to the object (if defined)

missing_host_key(*client*, *hostname*, *key*)
Called when an `SSHClient` receives a server key for a server that isn't in either the system or local `HostKeys` object. To accept the key, simply return. To reject, raised an exception (which will be passed to the calling application).

class paramiko.client.**RejectPolicy**
Policy for automatically rejecting the unknown hostname & key. This is used by `SSHClient`.

class paramiko.client.**SSHClient**
A high-level representation of a session with an SSH server. This class wraps `Transport`, `Channel`, and `SFTPClient` to take care of most aspects of authenticating and opening channels. A typical use case is:

```
client = SSHClient()
client.load_system_host_keys()
client.connect('ssh.example.com')
stdin, stdout, stderr = client.exec_command('ls -l')
```

You may pass in explicit overrides for authentication and server host key checking. The default mechanism is to try to use local key files or an SSH agent (if one is running).

New in version 1.6.

__init__()
Create a new SSHClient.

__weakref__
list of weak references to the object (if defined)

close()
Close this SSHClient and its underlying `Transport`.

connect(*hostname*, *port=22*, *username=None*, *password=None*, *pkey=None*, *key_filename=None*, *timeout=None*, *allow_agent=True*, *look_for_keys=True*, *compress=False*, *sock=None*)

Connect to an SSH server and authenticate to it. The server's host key is checked against the system host keys (see `load_system_host_keys`) and any local host keys (`load_host_keys`). If the server's hostname is not found in either set of host keys, the missing host key policy is used (see `set_missing_host_key_policy`). The default policy is to reject the key and raise an `SSHException`.

Authentication is attempted in the following order of priority:

- The `pkey` or `key_filename` passed in (if any)
- Any key we can find through an SSH agent
- Any "id_rsa" or "id_dsa" key discoverable in `~/.ssh/`
- Plain username/password auth, if a password was given

If a private key requires a password to unlock it, and a password is passed in, that password will be used to attempt to unlock the key.

Parameters

- **hostname** (*str*) – the server to connect to
- **port** (*int*) – the server port to connect to
- **username** (*str*) – the username to authenticate as (defaults to the current local username)
- **password** (*str*) – a password to use for authentication or for unlocking a private key
- **pkey** (*.PKey*) – an optional private key to use for authentication
- **key_filename** (*str*) – the filename, or list of filenames, of optional private key(s) to try for authentication
- **timeout** (*float*) – an optional timeout (in seconds) for the TCP connect
- **allow_agent** (*bool*) – set to False to disable connecting to the SSH agent
- **look_for_keys** (*bool*) – set to False to disable searching for discoverable private key files in `~/.ssh/`
- **compress** (*bool*) – set to True to turn on compression
- **sock** (*socket*) – an open socket or socket-like object (such as a `Channel`) to use for communication to the target host

Raises

- **BadHostKeyException** – if the server's host key could not be verified
- **AuthenticationException** – if authentication failed
- **SSHException** – if there was any other error connecting or establishing an SSH session
- **socket.error** – if a socket error occurred while connecting

exec_command(*command*, *bufsize=-1*, *timeout=None*, *get_pty=False*)

Execute a command on the SSH server. A new `Channel` is opened and the requested command is executed. The command's input and output streams are returned as Python `file`-like objects representing stdin, stdout, and stderr.

Parameters

- **command** (*str*) – the command to execute
- **bufsize** (*int*) – interpreted the same way as by the built-in `file()` function in Python

- **timeout** (*int*) – set command's channel timeout. See `Channel.settimeout`.settimeout

Returns the stdin, stdout, and stderr of the executing command, as a 3-tuple

Raises SSHException if the server fails to execute the command

get_host_keys()
Get the local `HostKeys` object. This can be used to examine the local host keys or change them.

Returns the local host keys as a `HostKeys` object.

get_transport()
Return the underlying `Transport` object for this SSH connection. This can be used to perform lower-level tasks, like opening specific kinds of channels.

Returns the `Transport` for this connection

invoke_shell(*term='vt100'*, *width=80*, *height=24*, *width_pixels=0*, *height_pixels=0*)
Start an interactive shell session on the SSH server. A new `Channel` is opened and connected to a pseudo-terminal using the requested terminal type and size.

Parameters

- **term** (*str*) – the terminal type to emulate (for example, `"vt100"`)
- **width** (*int*) – the width (in characters) of the terminal window
- **height** (*int*) – the height (in characters) of the terminal window
- **width_pixels** (*int*) – the width (in pixels) of the terminal window
- **height_pixels** (*int*) – the height (in pixels) of the terminal window

Returns a new `Channel` connected to the remote shell

Raises SSHException if the server fails to invoke a shell

load_host_keys(*filename*)
Load host keys from a local host-key file. Host keys read with this method will be checked after keys loaded via `load_system_host_keys`, but will be saved back by `save_host_keys` (so they can be modified). The missing host key policy `AutoAddPolicy` adds keys to this set and saves them, when connecting to a previously-unknown server.

This method can be called multiple times. Each new set of host keys will be merged with the existing set (new replacing old if there are conflicts). When automatically saving, the last hostname is used.

Parameters **filename** (*str*) – the filename to read

Raises IOError if the filename could not be read

load_system_host_keys(*filename=None*)
Load host keys from a system (read-only) file. Host keys read with this method will not be saved back by `save_host_keys`.

This method can be called multiple times. Each new set of host keys will be merged with the existing set (new replacing old if there are conflicts).

If `filename` is left as `None`, an attempt will be made to read keys from the user's local "known hosts" file, as used by OpenSSH, and no exception will be raised if the file can't be read. This is probably only useful on posix.

Parameters **filename** (*str*) – the filename to read, or `None`

Raises IOError if a filename was provided and the file could not be read

open_sftp()
Open an SFTP session on the SSH server.

Returns a new SFTPClient session object

save_host_keys(*filename*)
Save the host keys back to a file. Only the host keys loaded with load_host_keys (plus any added directly) will be saved – not any host keys loaded with load_system_host_keys.

Parameters **filename** (*str*) – the filename to save to

Raises IOError if the file could not be written

set_log_channel(*name*)
Set the channel for logging. The default is "paramiko.transport" but it can be set to anything you want.

Parameters **name** (*str*) – new channel name for logging

set_missing_host_key_policy(*policy*)
Set the policy to use when connecting to a server that doesn't have a host key in either the system or local HostKeys objects. The default policy is to reject all unknown servers (using RejectPolicy). You may substitute AutoAddPolicy or write your own policy class.

Parameters **policy** (*.MissingHostKeyPolicy*) – the policy to use when receiving a host key from a previously-unknown server

class paramiko.client.**WarningPolicy**
Policy for logging a Python-style warning for an unknown host key, but accepting it. This is used by SSHClient.

1.1.3 Message

Implementation of an SSH2 "message".

class paramiko.message.**Message**(*content=None*)
An SSH2 message is a stream of bytes that encodes some combination of strings, integers, bools, and infinite-precision integers (known in Python as longs). This class builds or breaks down such a byte stream.

Normally you don't need to deal with anything this low-level, but it's exposed for people implementing custom extensions, or features that paramiko doesn't support yet.

__init__(*content=None*)
Create a new SSH2 message.

Parameters **content** (*str*) – the byte stream to use as the message content (passed in only when decomposing a message).

__repr__()
Returns a string representation of this object, for debugging.

__str__()
Return the byte stream content of this message, as a string/bytes obj.

__weakref__
list of weak references to the object (if defined)

add(**seq*)
Add a sequence of items to the stream. The values are encoded based on their type: str, int, bool, list, or long.

> **Warning:** Longs are encoded non-deterministically. Don't use this method.

Parameters **seq** – the sequence of items

add_boolean(*b*)
Add a boolean value to the stream.

Parameters **b** (*bool*) – boolean value to add

add_byte(*b*)
Write a single byte to the stream, without any formatting.

Parameters **b** (*str*) – byte to add

add_bytes(*b*)
Write bytes to the stream, without any formatting.

Parameters **b** (*str*) – bytes to add

add_int(*n*)
Add an integer to the stream.

Parameters **n** (*int*) – integer to add

add_int64(*n*)
Add a 64-bit int to the stream.

Parameters **n** (*long*) – long int to add

add_list(*l*)
Add a list of strings to the stream. They are encoded identically to a single string of values separated by commas. (Yes, really, that's how SSH2 does it.)

Parameters **l** (*list*) – list of strings to add

add_mpint(*z*)
Add a long int to the stream, encoded as an infinite-precision integer. This method only works on positive numbers.

Parameters **z** (*long*) – long int to add

add_size(*n*)
Add an integer to the stream.

Parameters **n** (*int*) – integer to add

add_string(*s*)
Add a string to the stream.

Parameters **s** (*str*) – string to add

asbytes()
Return the byte stream content of this Message, as bytes.

get_binary()
Fetch a string from the stream. This could be a byte string and may contain unprintable characters. (It's not unheard of for a string to contain another byte-stream Message.)

@return: a string. @rtype: string

get_boolean()
Fetch a boolean from the stream.

get_byte()
Return the next byte of the message, without decomposing it. This is equivalent to `get_bytes(1)`.

Returns the next (`str`) byte of the message, or `'\'` if there aren't any bytes remaining.

get_bytes(*n*)
Return the next `n` bytes of the message (as a `str`), without decomposing into an int, decoded string, etc. Just the raw bytes are returned. Returns a string of `n` zero bytes if there weren't `n` bytes remaining in the message.

get_int()
Fetch an int from the stream.

Returns a 32-bit unsigned `int`.

get_int64()
Fetch a 64-bit int from the stream.

Returns a 64-bit unsigned integer (`long`).

get_list()
Fetch a `list` of `strings` from the stream.

These are trivially encoded as comma-separated values in a string.

get_mpint()
Fetch a long int (mpint) from the stream.

Returns an arbitrary-length integer (`long`).

get_remainder()
Return the bytes (as a `str`) of this message that haven't already been parsed and returned.

get_size()
Fetch an int from the stream.

@return: a 32-bit unsigned integer. @rtype: int

get_so_far()
Returns the `str` bytes of this message that have been parsed and returned. The string passed into a message's constructor can be regenerated by concatenating `get_so_far` and `get_remainder`.

get_string()
Fetch a `str` from the stream. This could be a byte string and may contain unprintable characters. (It's not unheard of for a string to contain another byte-stream message.)

get_text()
Fetch a string from the stream. This could be a byte string and may contain unprintable characters. (It's not unheard of for a string to contain another byte-stream Message.)

@return: a string. @rtype: string

rewind()
Rewind the message to the beginning as if no items had been parsed out of it yet.

1.1.4 Packetizer

Packet handling

class `paramiko.packet.`**Packetizer**(*socket*)
Implementation of the base SSH packet protocol.

__weakref__
list of weak references to the object (if defined)

need_rekey()
Returns `True` if a new set of keys needs to be negotiated. This will be triggered during a packet read or write, so it should be checked after every read or write, or at least after every few.

read_all(*n*, *check_rekey=False*)
Read as close to N bytes as possible, blocking as long as necessary.

Parameters **n** (*int*) – number of bytes to read

Returns the data read, as a `str`

Raises EOFError if the socket was closed before all the bytes could be read

read_message()
Only one thread should ever be in this function (no other locking is done).

Raises

- **SSHException** – if the packet is mangled
- **NeedRekeyException** – if the transport should rekey

readline(*timeout*)
Read a line from the socket. We assume no data is pending after the line, so it's okay to attempt large reads.

send_message(*data*)
Write a block of data using the current cipher, as an SSH block.

set_inbound_cipher(*block_engine*, *block_size*, *mac_engine*, *mac_size*, *mac_key*)
Switch inbound data cipher.

set_keepalive(*interval*, *callback*)
Turn on/off the callback keepalive. If `interval` seconds pass with no data read from or written to the socket, the callback will be executed and the timer will be reset.

set_log(*log*)
Set the Python log object to use for logging.

set_outbound_cipher(*block_engine*, *block_size*, *mac_engine*, *mac_size*, *mac_key*, *sdctr=False*)
Switch outbound data cipher.

1.1.5 Transport

Core protocol implementation

class paramiko.transport.**SecurityOptions**(*transport*)
Simple object containing the security preferences of an ssh transport. These are tuples of acceptable ciphers, digests, key types, and key exchange algorithms, listed in order of preference.

Changing the contents and/or order of these fields affects the underlying `Transport` (but only if you change them before starting the session). If you try to add an algorithm that paramiko doesn't recognize, `ValueError` will be raised. If you try to assign something besides a tuple to one of the fields, `TypeError` will be raised.

__repr__()
Returns a string representation of this object, for debugging.

ciphers
Symmetric encryption ciphers

compression
Compression algorithms

digests
Digest (one-way hash) algorithms

kex
Key exchange algorithms

key_types
Public-key algorithms

class paramiko.transport.**Transport**(*sock*)
An SSH Transport attaches to a stream (usually a socket), negotiates an encrypted session, authenticates, and then creates stream tunnels, called channels, across the session. Multiple channels can be multiplexed across a single session (and often are, in the case of port forwardings).

__init__(*sock*)
Create a new SSH session over an existing socket, or socket-like object. This only creates the Transport object; it doesn't begin the SSH session yet. Use connect or start_client to begin a client session, or start_server to begin a server session.

If the object is not actually a socket, it must have the following methods:

- send(str): Writes from 1 to len(str) bytes, and returns an int representing the number of bytes written. Returns 0 or raises EOFError if the stream has been closed.
- recv(int): Reads from 1 to int bytes and returns them as a string. Returns 0 or raises EOFError if the stream has been closed.
- close(): Closes the socket.
- settimeout(n): Sets a (float) timeout on I/O operations.

For ease of use, you may also pass in an address (as a tuple) or a host string as the sock argument. (A host string is a hostname with an optional port (separated by ":") which will be converted into a tuple of (hostname, port).) A socket will be connected to this address and used for communication. Exceptions from the socket call may be thrown in this case.

Parameters **sock** (*socket*) – a socket or socket-like object to create the session over.

__repr__()
Returns a string representation of this object, for debugging.

accept(*timeout=None*)
Return the next channel opened by the client over this transport, in server mode. If no channel is opened before the given timeout, None is returned.

Parameters **timeout** (*int*) – seconds to wait for a channel, or None to wait forever

Returns a new Channel opened by the client

add_server_key(*key*)
Add a host key to the list of keys used for server mode. When behaving as a server, the host key is used to sign certain packets during the SSH2 negotiation, so that the client can trust that we are who we say we are. Because this is used for signing, the key must contain private key info, not just the public half. Only one key of each type (RSA or DSS) is kept.

Parameters **key** (*.PKey*) – the host key to add, usually an RSAKey or DSSKey.

atfork()
Terminate this Transport without closing the session. On posix systems, if a Transport is open during process forking, both parent and child will share the underlying socket, but only one process can use the connection (without corrupting the session). Use this method to clean up a Transport object without disrupting the other process.

New in version 1.5.3.

auth_interactive(*username*, *handler*, *submethods=''*)

Authenticate to the server interactively. A handler is used to answer arbitrary questions from the server. On many servers, this is just a dumb wrapper around PAM.

This method will block until the authentication succeeds or fails, peroidically calling the handler asynchronously to get answers to authentication questions. The handler may be called more than once if the server continues to ask questions.

The handler is expected to be a callable that will handle calls of the form: `handler(title, instructions, prompt_list)`. The `title` is meant to be a dialog-window title, and the `instructions` are user instructions (both are strings). `prompt_list` will be a list of prompts, each prompt being a tuple of `(str, bool)`. The string is the prompt and the boolean indicates whether the user text should be echoed.

A sample call would thus be: `handler('title', 'instructions', [('Password:', False)])`.

The handler should return a list or tuple of answers to the server's questions.

If the server requires multi-step authentication (which is very rare), this method will return a list of auth types permissible for the next step. Otherwise, in the normal case, an empty list is returned.

Parameters

- **username** (*str*) – the username to authenticate as
- **handler** (*callable*) – a handler for responding to server questions
- **submethods** (*str*) – a string list of desired submethods (optional)

Returns `list` of auth types permissible for the next stage of authentication (normally empty).

Raises

- **BadAuthenticationType** – if public-key authentication isn't allowed by the server for this user
- **AuthenticationException** – if the authentication failed
- **SSHException** – if there was a network error

New in version 1.5.

auth_none(*username*)

Try to authenticate to the server using no authentication at all. This will almost always fail. It may be useful for determining the list of authentication types supported by the server, by catching the `BadAuthenticationType` exception raised.

Parameters **username** (*str*) – the username to authenticate as

Returns `list` of auth types permissible for the next stage of authentication (normally empty)

Raises

- **BadAuthenticationType** – if "none" authentication isn't allowed by the server for this user
- **SSHException** – if the authentication failed due to a network error

New in version 1.5.

auth_password(*username*, *password*, *event=None*, *fallback=True*)

Authenticate to the server using a password. The username and password are sent over an encrypted link.

If an `event` is passed in, this method will return immediately, and the event will be triggered once authentication succeeds or fails. On success, `is_authenticated` will return `True`. On failure, you may use `get_exception` to get more detailed error information.

Since 1.1, if no event is passed, this method will block until the authentication succeeds or fails. On failure, an exception is raised. Otherwise, the method simply returns.

Since 1.5, if no event is passed and `fallback` is `True` (the default), if the server doesn't support plain password authentication but does support so-called "keyboard-interactive" mode, an attempt will be made to authenticate using this interactive mode. If it fails, the normal exception will be thrown as if the attempt had never been made. This is useful for some recent Gentoo and Debian distributions, which turn off plain password authentication in a misguided belief that interactive authentication is "more secure". (It's not.)

If the server requires multi-step authentication (which is very rare), this method will return a list of auth types permissible for the next step. Otherwise, in the normal case, an empty list is returned.

Parameters

- **username** (*str*) – the username to authenticate as
- **password** (*basestring*) – the password to authenticate with
- **event** (*.threading.Event*) – an event to trigger when the authentication attempt is complete (whether it was successful or not)
- **fallback** (*bool*) – `True` if an attempt at an automated "interactive" password auth should be made if the server doesn't support normal password auth

Returns `list` of auth types permissible for the next stage of authentication (normally empty)

Raises

- **BadAuthenticationType** – if password authentication isn't allowed by the server for this user (and no event was passed in)
- **AuthenticationException** – if the authentication failed (and no event was passed in)
- **SSHException** – if there was a network error

auth_publickey (*username*, *key*, *event=None*)

Authenticate to the server using a private key. The key is used to sign data from the server, so it must include the private part.

If an `event` is passed in, this method will return immediately, and the event will be triggered once authentication succeeds or fails. On success, `is_authenticated` will return `True`. On failure, you may use `get_exception` to get more detailed error information.

Since 1.1, if no event is passed, this method will block until the authentication succeeds or fails. On failure, an exception is raised. Otherwise, the method simply returns.

If the server requires multi-step authentication (which is very rare), this method will return a list of auth types permissible for the next step. Otherwise, in the normal case, an empty list is returned.

Parameters

- **username** (*str*) – the username to authenticate as
- **key** (*.PKey*) – the private key to authenticate with
- **event** (*.threading.Event*) – an event to trigger when the authentication attempt is complete (whether it was successful or not)

Returns `list` of auth types permissible for the next stage of authentication (normally empty)

Raises

- **BadAuthenticationType** – if public-key authentication isn't allowed by the server for this user (and no event was passed in)
- **AuthenticationException** – if the authentication failed (and no event was passed in)
- **SSHException** – if there was a network error

cancel_port_forward(*address*, *port*)
Ask the server to cancel a previous port-forwarding request. No more connections to the given address & port will be forwarded across this ssh connection.

Parameters

- **address** (*str*) – the address to stop forwarding
- **port** (*int*) – the port to stop forwarding

close()
Close this session, and any open channels that are tied to it.

connect(*hostkey=None*, *username=''*, *password=None*, *pkey=None*)
Negotiate an SSH2 session, and optionally verify the server's host key and authenticate using a password or private key. This is a shortcut for `start_client`, `get_remote_server_key`, and `Transport.auth_password` or `Transport.auth_publickey`. Use those methods if you want more control.

You can use this method immediately after creating a Transport to negotiate encryption with a server. If it fails, an exception will be thrown. On success, the method will return cleanly, and an encrypted session exists. You may immediately call `open_channel` or `open_session` to get a `Channel` object, which is used for data transfer.

Note: If you fail to supply a password or private key, this method may succeed, but a subsequent `open_channel` or `open_session` call may fail because you haven't authenticated yet.

Parameters

- **hostkey** (*.PKey*) – the host key expected from the server, or `None` if you don't want to do host key verification.
- **username** (*str*) – the username to authenticate as.
- **password** (*str*) – a password to use for authentication, if you want to use password authentication; otherwise `None`.
- **pkey** (*.PKey*) – a private key to use for authentication, if you want to use private key authentication; otherwise `None`.

Raises SSHException if the SSH2 negotiation fails, the host key supplied by the server is incorrect, or authentication fails.

get_banner()
Return the banner supplied by the server upon connect. If no banner is supplied, this method returns `None`.

Returns server supplied banner (`str`), or `None`.

New in version 1.13.

get_exception()
Return any exception that happened during the last server request. This can be used to fetch more specific error information after using calls like `start_client`. The exception (if any) is cleared after this call.

Returns an exception, or `None` if there is no stored exception.

New in version 1.1.

get_hexdump()
Return `True` if the transport is currently logging hex dumps of protocol traffic.

Returns `True` if hex dumps are being logged, else `False`.

New in version 1.4.

get_log_channel()
Return the channel name used for this transport's logging.

Returns channel name as a `str`

New in version 1.2.

get_remote_server_key()
Return the host key of the server (in client mode).

Note: Previously this call returned a tuple of `(key type, key string)`. You can get the same effect by calling `PKey.get_name` for the key type, and `str(key)` for the key string.

Raises SSHException if no session is currently active.

Returns public key (`PKey`) of the remote server

get_security_options()
Return a `SecurityOptions` object which can be used to tweak the encryption algorithms this transport will permit (for encryption, digest/hash operations, public keys, and key exchanges) and the order of preference for them.

get_server_key()
Return the active host key, in server mode. After negotiating with the client, this method will return the negotiated host key. If only one type of host key was set with `add_server_key`, that's the only key that will ever be returned. But in cases where you have set more than one type of host key (for example, an RSA key and a DSS key), the key type will be negotiated by the client, and this method will return the key of the type agreed on. If the host key has not been negotiated yet, `None` is returned. In client mode, the behavior is undefined.

Returns host key (`PKey`) of the type negotiated by the client, or `None`.

get_username()
Return the username this connection is authenticated for. If the session is not authenticated (or authentication failed), this method returns `None`.

Returns username that was authenticated (a `str`), or `None`.

getpeername()
Return the address of the remote side of this Transport, if possible. This is effectively a wrapper around `'getpeername'` on the underlying socket. If the socket-like object has no `'getpeername'` method, then `("unknown", 0)` is returned.

Returns the address of the remote host, if known, as a `(str, int)` tuple.

global_request(*kind*, *data=None*, *wait=True*)
Make a global request to the remote host. These are normally extensions to the SSH2 protocol.

Parameters

- **kind** (*str*) – name of the request.
- **data** (*tuple*) – an optional tuple containing additional data to attach to the request.

- **wait** (*bool*) – `True` if this method should not return until a response is received; `False` otherwise.

Returns a `Message` containing possible additional data if the request was successful (or an empty `Message` if `wait` was `False`); `None` if the request was denied.

is_active()
Return true if this session is active (open).

Returns True if the session is still active (open); False if the session is closed

is_authenticated()
Return true if this session is active and authenticated.

Returns True if the session is still open and has been authenticated successfully; False if authentication failed and/or the session is closed.

static **load_server_moduli**(*filename=None*)
(optional) Load a file of prime moduli for use in doing group-exchange key negotiation in server mode. It's a rather obscure option and can be safely ignored.

In server mode, the remote client may request "group-exchange" key negotiation, which asks the server to send a random prime number that fits certain criteria. These primes are pretty difficult to compute, so they can't be generated on demand. But many systems contain a file of suitable primes (usually named something like `/etc/ssh/moduli`). If you call `load_server_moduli` and it returns `True`, then this file of primes has been loaded and we will support "group-exchange" in server mode. Otherwise server mode will just claim that it doesn't support that method of key negotiation.

Parameters **filename** (*str*) – optional path to the moduli file, if you happen to know that it's not in a standard location.

Returns True if a moduli file was successfully loaded; False otherwise.

Note: This has no effect when used in client mode.

---

open_channel(*kind*, *dest_addr=None*, *src_addr=None*)
Request a new channel to the server. `Channels` are socket-like objects used for the actual transfer of data across the session. You may only request a channel after negotiating encryption (using `connect` or `start_client`) and authenticating.

Parameters

- **kind** (*str*) – the kind of channel requested (usually `"session"`, `"forwarded-tcpip"`, `"direct-tcpip"`, or `"x11"`)
- **dest_addr** (*tuple*) – the destination address (address + port tuple) of this port forwarding, if `kind` is `"forwarded-tcpip"` or `"direct-tcpip"` (ignored for other channel types)
- **src_addr** – the source address of this port forwarding, if `kind` is `"forwarded-tcpip"`, `"direct-tcpip"`, or `"x11"`

Returns a new `Channel` on success

Raises SSHException if the request is rejected or the session ends prematurely

open_forward_agent_channel()
Request a new channel to the client, of type `"auth-agent@openssh.com"`.

This is just an alias for `open_channel('auth-agent@openssh.com')`.

Returns a new `Channel`

Raises SSHException if the request is rejected or the session ends prematurely

open_forwarded_tcpip_channel(*src_addr*, *dest_addr*)
Request a new channel back to the client, of type `"forwarded-tcpip"`. This is used after a client has requested port forwarding, for sending incoming connections back to the client.

Parameters

- **src_addr** – originator's address
- **dest_addr** – local (server) connected address

open_session()
Request a new channel to the server, of type `"session"`. This is just an alias for calling `open_channel` with an argument of `"session"`.

Returns a new `Channel`

Raises SSHException if the request is rejected or the session ends prematurely

open_sftp_client()
Create an SFTP client channel from an open transport. On success, an SFTP session will be opened with the remote host, and a new `SFTPClient` object will be returned.

Returns a new `SFTPClient` referring to an sftp session (channel) across this transport

open_x11_channel(*src_addr=None*)
Request a new channel to the client, of type `"x11"`. This is just an alias for `open_channel('x11', src_addr=src_addr)`.

Parameters **src_addr** (*tuple*) – the source address (`(str, int)`) of the x11 server (port is the x11 port, ie. 6010)

Returns a new `Channel`

Raises SSHException if the request is rejected or the session ends prematurely

renegotiate_keys()
Force this session to switch to new keys. Normally this is done automatically after the session hits a certain number of packets or bytes sent or received, but this method gives you the option of forcing new keys whenever you want. Negotiating new keys causes a pause in traffic both ways as the two sides swap keys and do computations. This method returns when the session has switched to new keys.

Raises SSHException if the key renegotiation failed (which causes the session to end)

request_port_forward(*address*, *port*, *handler=None*)
Ask the server to forward TCP connections from a listening port on the server, across this SSH session.

If a handler is given, that handler is called from a different thread whenever a forwarded connection arrives. The handler parameters are:

```
handler(channel, (origin_addr, origin_port), (server_addr, server_port))
```

where `server_addr` and `server_port` are the address and port that the server was listening on.

If no handler is set, the default behavior is to send new incoming forwarded connections into the accept queue, to be picked up via `accept`.

Parameters

- **address** (*str*) – the address to bind when forwarding
- **port** (*int*) – the port to forward, or 0 to ask the server to allocate any port
- **handler** (*callable*) – optional handler for incoming forwarded connections, of the form `func(Channel, (str, int), (str, int))`.

Returns the port number (`int`) allocated by the server

Raises SSHException if the server refused the TCP forward request

send_ignore (*byte_count=None*)
Send a junk packet across the encrypted link. This is sometimes used to add "noise" to a connection to confuse would-be attackers. It can also be used as a keep-alive for long lived connections traversing firewalls.

Parameters **byte_count** (*int*) – the number of random bytes to send in the payload of the ignored packet – defaults to a random number from 10 to 41.

set_hexdump (*hexdump*)
Turn on/off logging a hex dump of protocol traffic at DEBUG level in the logs. Normally you would want this off (which is the default), but if you are debugging something, it may be useful.

Parameters **hexdump** (*bool*) – `True` to log protocol traffix (in hex) to the log; `False` otherwise.

set_keepalive (*interval*)
Turn on/off keepalive packets (default is off). If this is set, after `interval` seconds without sending any data over the connection, a "keepalive" packet will be sent (and ignored by the remote host). This can be useful to keep connections alive over a NAT, for example.

Parameters **interval** (*int*) – seconds to wait before sending a keepalive packet (or 0 to disable keepalives).

set_log_channel (*name*)
Set the channel for this transport's logging. The default is `"paramiko.transport"` but it can be set to anything you want. (See the `logging` module for more info.) SSH Channels will log to a sub-channel of the one specified.

Parameters **name** (*str*) – new channel name for logging

New in version 1.1.

set_subsystem_handler (*name*, *handler*, **larg*, ***kwarg*)
Set the handler class for a subsystem in server mode. If a request for this subsystem is made on an open ssh channel later, this handler will be constructed and called – see `SubsystemHandler` for more detailed documentation.

Any extra parameters (including keyword arguments) are saved and passed to the `SubsystemHandler` constructor later.

Parameters

- **name** (*str*) – name of the subsystem.
- **handler** (*class*) – subclass of `SubsystemHandler` that handles this subsystem.

start_client (*event=None*)
Negotiate a new SSH2 session as a client. This is the first step after creating a new `Transport`. A separate thread is created for protocol negotiation.

If an event is passed in, this method returns immediately. When negotiation is done (successful or not), the given `Event` will be triggered. On failure, `is_active` will return `False`.

(Since 1.4) If `event` is `None`, this method will not return until negotation is done. On success, the method returns normally. Otherwise an SSHException is raised.

After a successful negotiation, you will usually want to authenticate, calling `auth_password` or `auth_publickey`.

Note: `connect` is a simpler method for connecting as a client.

Note: After calling this method (or `start_server` or `connect`), you should no longer directly read from or write to the original socket object.

Parameters **event** (*.threading.Event*) – an event to trigger when negotiation is complete (optional)

Raises SSHException if negotiation fails (and no `event` was passed in)

`start_server`(*event=None*, *server=None*)

Negotiate a new SSH2 session as a server. This is the first step after creating a new `Transport` and setting up your server host key(s). A separate thread is created for protocol negotiation.

If an event is passed in, this method returns immediately. When negotiation is done (successful or not), the given `Event` will be triggered. On failure, `is_active` will return `False`.

(Since 1.4) If `event` is `None`, this method will not return until negotation is done. On success, the method returns normally. Otherwise an SSHException is raised.

After a successful negotiation, the client will need to authenticate. Override the methods `get_allowed_auths`, `check_auth_none`, `check_auth_password`, and `check_auth_publickey` in the given `server` object to control the authentication process.

After a successful authentication, the client should request to open a channel. Override `check_channel_request` in the given `server` object to allow channels to be opened.

Note: After calling this method (or `start_client` or `connect`), you should no longer directly read from or write to the original socket object.

Parameters

- **event** (*.threading.Event*) – an event to trigger when negotiation is complete.
- **server** (*.ServerInterface*) – an object used to perform authentication and create `channels`

Raises SSHException if negotiation fails (and no `event` was passed in)

`use_compression`(*compress=True*)

Turn on/off compression. This will only have an affect before starting the transport (ie before calling `connect`, etc). By default, compression is off since it negatively affects interactive sessions.

Parameters **compress** (*bool*) – `True` to ask the remote client/server to compress traffic; `False` to refuse compression

New in version 1.5.2.

1.2 Authentication & keys

1.2.1 SSH Agents

SSH Agent interface

class paramiko.agent.**Agent**

Client interface for using private keys from an SSH agent running on the local machine. If an SSH agent is running, this class can be used to connect to it and retreive PKey objects which can be used when attempting to authenticate to remote SSH servers.

Upon initialization, a session with the local machine's SSH agent is opened, if one is running. If no agent is running, initialization will succeed, but get_keys will return an empty tuple.

Raises SSHException if an SSH agent is found, but speaks an incompatible protocol

close()

Close the SSH agent connection.

get_keys()

Return the list of keys available through the SSH agent, if any. If no SSH agent was running (or it couldn't be contacted), an empty list will be returned.

Returns a tuple of AgentKey objects representing keys available on the SSH agent

class paramiko.agent.**AgentClientProxy**(*chanRemote*)

Class proxying request as a client:

1.client ask for a request_forward_agent()

2.server creates a proxy and a fake SSH Agent

3.server ask for establishing a connection when needed, calling the forward_agent_handler at client side.

4.the forward_agent_handler launch a thread for connecting the remote fake agent and the local agent

5.Communication occurs ...

close()

Close the current connection and terminate the agent Should be called manually

connect()

Method automatically called by AgentProxyThread.run.

class paramiko.agent.**AgentKey**(*agent*, *blob*)

Private key held in a local SSH agent. This type of key can be used for authenticating to a remote server (signing). Most other key operations work as expected.

can_sign()

Return True if this key has the private part necessary for signing data.

classmethod from_private_key(*file_obj*, *password=None*)

Create a key object by reading a private key from a file (or file-like) object. If the private key is encrypted and password is not None, the given password will be used to decrypt the key (otherwise PasswordRequiredException is thrown).

Parameters

- **file_obj** (*file*) – the file to read from
- **password** (*str*) – an optional password to use to decrypt the key, if it's encrypted

Returns a new PKey based on the given private key

Raises

- **IOError** – if there was an error reading the key
- **PasswordRequiredException** – if the private key file is encrypted, and password is None
- **SSHException** – if the key file is invalid

classmethod from_private_key_file (*filename*, *password=None*)
Create a key object by reading a private key file. If the private key is encrypted and `password` is not `None`, the given password will be used to decrypt the key (otherwise `PasswordRequiredException` is thrown). Through the magic of Python, this factory method will exist in all subclasses of PKey (such as `RSAKey` or `DSSKey`), but is useless on the abstract PKey class.

Parameters

- **filename** (*str*) – name of the file to read
- **password** (*str*) – an optional password to use to decrypt the key file, if it's encrypted

Returns a new `PKey` based on the given private key

Raises

- **IOError** – if there was an error reading the file
- **PasswordRequiredException** – if the private key file is encrypted, and `password` is `None`
- **SSHException** – if the key file is invalid

get_base64 ()
Return a base64 string containing the public part of this key. Nothing secret is revealed. This format is compatible with that used to store public key files or recognized host keys.

Returns a base64 `string` containing the public part of the key.

get_bits ()
Return the number of significant bits in this key. This is useful for judging the relative security of a key.

Returns bits in the key (as an `int`)

get_fingerprint ()
Return an MD5 fingerprint of the public part of this key. Nothing secret is revealed.

Returns a 16-byte `string` (binary) of the MD5 fingerprint, in SSH format.

verify_ssh_sig (*data*, *msg*)
Given a blob of data, and an SSH message representing a signature of that data, verify that it was signed with this key.

Parameters

- **data** (*str*) – the data that was signed.
- **msg** (*.Message*) – an SSH signature message

Returns `True` if the signature verifies correctly; `False` otherwise.

write_private_key (*file_obj*, *password=None*)
Write private key contents into a file (or file-like) object. If the password is not `None`, the key is encrypted before writing.

Parameters

- **file_obj** (*file*) – the file object to write into
- **password** (*str*) – an optional password to use to encrypt the key

Raises

- **IOError** – if there was an error writing to the file
- **SSHException** – if the key is invalid

write_private_key_file(*filename*, *password=None*)
Write private key contents into a file. If the password is not `None`, the key is encrypted before writing.

Parameters

- **filename** (*str*) – name of the file to write
- **password** (*str*) – an optional password to use to encrypt the key file

Raises

- **IOError** – if there was an error writing the file
- **SSHException** – if the key is invalid

class paramiko.agent.**AgentLocalProxy**(*agent*)
Class to be used when wanting to ask a local SSH Agent being asked from a remote fake agent (so use a unix socket for ex.)

daemon
A boolean value indicating whether this thread is a daemon thread (True) or not (False).

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when no alive non-daemon threads are left.

get_connection()
Return a pair of socket object and string address.

May block!

ident
Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the thread.get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isAlive()
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

is_alive()
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

join(*timeout=None*)
Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call isAlive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

name
A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

start()
Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

class paramiko.agent.**AgentProxyThread**(*agent*)
Class in charge of communication between two channels.

daemon
A boolean value indicating whether this thread is a daemon thread (True) or not (False).

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when no alive non-daemon threads are left.

ident
Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the thread.get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isAlive()
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

is_alive()
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

join(*timeout=None*)
Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call isAlive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

name
A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

start()
Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

class paramiko.agent.**AgentRemoteProxy**(*agent*, *chan*)
Class to be used when wanting to ask a remote SSH Agent

daemon
A boolean value indicating whether this thread is a daemon thread (True) or not (False).

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when no alive non-daemon threads are left.

ident
Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the thread.get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isAlive()
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

is_alive()
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

join(*timeout=None*)
Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call isAlive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

name
A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

start()
Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

class paramiko.agent.**AgentServerProxy**(*t*)

Parameters **t** (*.Transport*) – Transport used for SSH Agent communication forwarding

Raises SSHException mostly if we lost the agent

close()
Terminate the agent, clean the files, close connections Should be called manually

get_env()
Helper for the environnement under unix

Returns a dict containing the SSH_AUTH_SOCK environnement variables

get_keys()
Return the list of keys available through the SSH agent, if any. If no SSH agent was running (or it couldn't be contacted), an empty list will be returned.

Returns a tuple of AgentKey objects representing keys available on the SSH agent

1.2.2 Host keys / known_hosts files

class paramiko.hostkeys.**HostKeyEntry**(*hostnames=None*, *key=None*)
Representation of a line in an OpenSSH-style "known hosts" file.

classmethod from_line(*line*, *lineno=None*)
Parses the given line of text to find the names for the host, the type of key, and the key data. The line is expected to be in the format used by the OpenSSH known_hosts file.

Lines are expected to not have leading or trailing whitespace. We don't bother to check for comments or empty lines. All of that should be taken care of before sending the line to us.

Parameters **line** (*str*) – a line from an OpenSSH known_hosts file

to_line()
Returns a string in OpenSSH known_hosts file format, or None if the object is not in a valid state. A trailing newline is included.

class paramiko.hostkeys.**HostKeys**(*filename=None*)
Representation of an OpenSSH-style "known hosts" file. Host keys can be read from one or more files, and then individual hosts can be looked up to verify server keys during SSH negotiation.

A HostKeys object can be treated like a dict; any dict lookup is equivalent to calling lookup.

New in version 1.5.3.

__init__(*filename=None*)
Create a new HostKeys object, optionally loading keys from an OpenSSH style host-key file.

Parameters **filename** (*str*) – filename to load host keys from, or None

add(*hostname*, *keytype*, *key*)
Add a host key entry to the table. Any existing entry for a `(hostname, keytype)` pair will be replaced.

Parameters

- **hostname** (*str*) – the hostname (or IP) to add
- **keytype** (*str*) – key type (`"ssh-rsa"` or `"ssh-dss"`)
- **key** (*.PKey*) – the key to add

check(*hostname*, *key*)
Return True if the given key is associated with the given hostname in this dictionary.

Parameters

- **hostname** (*str*) – hostname (or IP) of the SSH server
- **key** (*.PKey*) – the key to check

Returns `True` if the key is associated with the hostname; else `False`

clear()
Remove all host keys from the dictionary.

static **hash_host**(*hostname*, *salt=None*)
Return a "hashed" form of the hostname, as used by OpenSSH when storing hashed hostnames in the known_hosts file.

Parameters

- **hostname** (*str*) – the hostname to hash
- **salt** (*str*) – optional salt to use when hashing (must be 20 bytes long)

Returns the hashed hostname as a `str`

load(*filename*)
Read a file of known SSH host keys, in the format used by OpenSSH. This type of file unfortunately doesn't exist on Windows, but on posix, it will usually be stored in `os.path.expanduser("~/.ssh/known_hosts")`.

If this method is called multiple times, the host keys are merged, not cleared. So multiple calls to `load` will just call `add`, replacing any existing entries and adding new ones.

Parameters **filename** (*str*) – name of the file to read host keys from

Raises IOError if there was an error reading the file

lookup(*hostname*)
Find a hostkey entry for a given hostname or IP. If no entry is found, `None` is returned. Otherwise a dictionary of keytype to key is returned. The keytype will be either `"ssh-rsa"` or `"ssh-dss"`.

Parameters **hostname** (*str*) – the hostname (or IP) to lookup

Returns dict of `str` -> `PKey` keys associated with this host (or `None`)

save(*filename*)
Save host keys into a file, in the format used by OpenSSH. The order of keys in the file will be preserved when possible (if these keys were loaded from a file originally). The single exception is that combined lines will be split into individual key lines, which is arguably a bug.

Parameters **filename** (*str*) – name of the file to write

Raises IOError if there was an error writing the file

New in version 1.6.1.

1.2.3 Key handling

Parent key class

Common API for all public keys.

class paramiko.pkey.**PKey**(*msg=None*, *data=None*)
Base class for public keys.

__cmp__(*other*)
Compare this key to another. Returns 0 if this key is equivalent to the given key, or non-0 if they are different. Only the public parts of the key are compared, so a public key will compare equal to its corresponding private key.

Parameters **other** (*.Pkey*) – key to compare to.

__init__(*msg=None*, *data=None*)
Create a new instance of this public key type. If `msg` is given, the key's public part(s) will be filled in from the message. If `data` is given, the key's public part(s) will be filled in from the string.

Parameters

- **msg** (*.Message*) – an optional SSH `Message` containing a public key of this type.
- **data** (*str*) – an optional string containing a public key of this type

Raises SSHException if a key cannot be created from the `data` or `msg` given, or no key was passed in.

__weakref__
list of weak references to the object (if defined)

asbytes()
Return a string of an SSH `Message` made up of the public part(s) of this key. This string is suitable for passing to `__init__` to re-create the key object later.

can_sign()
Return `True` if this key has the private part necessary for signing data.

classmethod from_private_key(*file_obj*, *password=None*)
Create a key object by reading a private key from a file (or file-like) object. If the private key is encrypted and `password` is not `None`, the given password will be used to decrypt the key (otherwise `PasswordRequiredException` is thrown).

Parameters

- **file_obj** (*file*) – the file to read from
- **password** (*str*) – an optional password to use to decrypt the key, if it's encrypted

Returns a new `PKey` based on the given private key

Raises

- **IOError** – if there was an error reading the key
- **PasswordRequiredException** – if the private key file is encrypted, and `password` is `None`
- **SSHException** – if the key file is invalid

classmethod from_private_key_file (*filename*, *password=None*)
Create a key object by reading a private key file. If the private key is encrypted and `password` is not `None`, the given password will be used to decrypt the key (otherwise `PasswordRequiredException` is thrown). Through the magic of Python, this factory method will exist in all subclasses of PKey (such as `RSAKey` or `DSSKey`), but is useless on the abstract PKey class.

Parameters

- **filename** (*str*) – name of the file to read
- **password** (*str*) – an optional password to use to decrypt the key file, if it's encrypted

Returns a new `PKey` based on the given private key

Raises

- **IOError** – if there was an error reading the file
- **PasswordRequiredException** – if the private key file is encrypted, and `password` is `None`
- **SSHException** – if the key file is invalid

get_base64 ()
Return a base64 string containing the public part of this key. Nothing secret is revealed. This format is compatible with that used to store public key files or recognized host keys.

Returns a base64 `string` containing the public part of the key.

get_bits ()
Return the number of significant bits in this key. This is useful for judging the relative security of a key.

Returns bits in the key (as an `int`)

get_fingerprint ()
Return an MD5 fingerprint of the public part of this key. Nothing secret is revealed.

Returns a 16-byte `string` (binary) of the MD5 fingerprint, in SSH format.

get_name ()
Return the name of this private key implementation.

Returns name of this private key type, in SSH terminology, as a `str` (for example, `"ssh-rsa"`).

sign_ssh_data (*rng*, *data*)
Sign a blob of data with this private key, and return a `Message` representing an SSH signature message.

Parameters

- **rng** (*.Crypto.Util.rng.RandomPool*) – a secure random number generator.
- **data** (*str*) – the data to sign.

Returns an SSH signature `message`.

verify_ssh_sig (*data*, *msg*)
Given a blob of data, and an SSH message representing a signature of that data, verify that it was signed with this key.

Parameters

- **data** (*str*) – the data that was signed.
- **msg** (*.Message*) – an SSH signature message

Returns `True` if the signature verifies correctly; `False` otherwise.

write_private_key (*file_obj*, *password=None*)
Write private key contents into a file (or file-like) object. If the password is not `None`, the key is encrypted before writing.

Parameters

- **file_obj** (*file*) – the file object to write into
- **password** (*str*) – an optional password to use to encrypt the key

Raises

- **IOError** – if there was an error writing to the file
- **SSHException** – if the key is invalid

write_private_key_file (*filename*, *password=None*)
Write private key contents into a file. If the password is not `None`, the key is encrypted before writing.

Parameters

- **filename** (*str*) – name of the file to write
- **password** (*str*) – an optional password to use to encrypt the key file

Raises

- **IOError** – if there was an error writing the file
- **SSHException** – if the key is invalid

DSA (DSS)

DSS keys.

class paramiko.dsskey.**DSSKey** (*msg=None*, *data=None*, *filename=None*, *password=None*, *vals=None*, *file_obj=None*)
Representation of a DSS key which can be used to sign an verify SSH2 data.

static generate (*bits=1024*, *progress_func=None*)
Generate a new private DSS key. This factory function can be used to generate a new host key or authentication key.

Parameters

- **bits** (*int*) – number of bits the generated key should be.
- **progress_func** (*function*) – an optional function to call at key points in key generation (used by `pyCrypto.PublicKey`).

Returns new `DSSKey` private key

RSA

RSA keys.

class paramiko.rsakey.**RSAKey** (*msg=None*, *data=None*, *filename=None*, *password=None*, *vals=None*, *file_obj=None*)
Representation of an RSA key which can be used to sign and verify SSH2 data.

static generate (*bits*, *progress_func=None*)
Generate a new private RSA key. This factory function can be used to generate a new host key or authentication key.

Parameters

- **bits** (*int*) – number of bits the generated key should be.
- **progress_func** (*function*) – an optional function to call at key points in key generation (used by `pyCrypto.PublicKey`).

Returns new `RSAKey` private key

ECDSA

ECDSA keys

class `paramiko.ecdsakey.`**`ECDSAKey`**(*msg=None*, *data=None*, *filename=None*, *password=None*, *vals=None*, *file_obj=None*)

Representation of an ECDSA key which can be used to sign and verify SSH2 data.

static `generate`(*bits*, *progress_func=None*)

Generate a new private RSA key. This factory function can be used to generate a new host key or authentication key.

Parameters **progress_func** (*function*) – an optional function to call at key points in key generation (used by `pyCrypto.PublicKey`).

Returns A new private key (`RSAKey`) object

1.3 Other primary functions

1.3.1 Configuration

Configuration file (aka `ssh_config`) support.

class `paramiko.config.`**`LazyFqdn`**(*config*, *host=None*)

Returns the host's fqdn on request as string.

`__weakref__`

list of weak references to the object (if defined)

class `paramiko.config.`**`SSHConfig`**

Representation of config information as stored in the format used by OpenSSH. Queries can be made via `lookup`. The format is described in OpenSSH's `ssh_config` man page. This class is provided primarily as a convenience to posix users (since the OpenSSH format is a de-facto standard on posix) but should work fine on Windows too.

New in version 1.6.

`__init__`()

Create a new OpenSSH config object.

`__weakref__`

list of weak references to the object (if defined)

`lookup`(*hostname*)

Return a dict of config options for a given hostname.

The host-matching rules of OpenSSH's `ssh_config` man page are used: For each parameter, the first obtained value will be used. The configuration files contain sections separated by "Host" specifications, and that section is only applied for hosts that match one of the patterns given in the specification.

Since the first obtained value for each parameter is used, more host- specific declarations should be given near the beginning of the file, and general defaults at the end.

The keys in the returned dict are all normalized to lowercase (look for `"port"`, not `"Port"`. The values are processed according to the rules for substitution variable expansion in `ssh_config`.

Parameters **hostname** (*str*) – the hostname to lookup

parse(*file_obj*)
Read an OpenSSH config from the given file object.

Parameters **file_obj** (*file*) – a file-like object to read the config file from

1.3.2 `ProxyCommand` support

class paramiko.proxy.**ProxyCommand**(*command_line*)
Wraps a subprocess running ProxyCommand-driven programs.

This class implements a the socket-like interface needed by the `Transport` and `Packetizer` classes. Using this class instead of a regular socket makes it possible to talk with a Popen'd command that will proxy traffic between the client and a server hosted in another machine.

__init__(*command_line*)
Create a new CommandProxy instance. The instance created by this class can be passed as an argument to the `Transport` class.

Parameters **command_line** (*str*) – the command that should be executed and used as the proxy.

__weakref__
list of weak references to the object (if defined)

recv(*size*)
Read from the standard output of the forked program.

Parameters **size** (*int*) – how many chars should be read

Returns the length of the read content, as an `int`

send(*content*)
Write the content received from the SSH client to the standard input of the forked command.

Parameters **content** (*str*) – string to be sent to the forked command

1.3.3 Server implementation

`ServerInterface` is an interface to override for server support.

class paramiko.server.**InteractiveQuery**(*name=''*, *instructions=''*, **prompts*)
A query (set of prompts) for a user during interactive authentication.

__init__(*name=''*, *instructions=''*, **prompts*)
Create a new interactive query to send to the client. The name and instructions are optional, but are generally displayed to the end user. A list of prompts may be included, or they may be added via the `add_prompt` method.

Parameters

- **name** (*str*) – name of this query
- **instructions** (*str*) – user instructions (usually short) about this query
- **prompts** (*str*) – one or more authentication prompts

__weakref__
list of weak references to the object (if defined)

add_prompt(*prompt*, *echo=True*)
Add a prompt to this query. The prompt should be a (reasonably short) string. Multiple prompts can be added to the same query.

Parameters

- **prompt** (*str*) – the user prompt
- **echo** (*bool*) – `True` (default) if the user's response should be echoed; `False` if not (for a password or similar)

class paramiko.server.**ServerInterface**
This class defines an interface for controlling the behavior of Paramiko in server mode.

Methods on this class are called from Paramiko's primary thread, so you shouldn't do too much work in them. (Certainly nothing that blocks or sleeps.)

__weakref__
list of weak references to the object (if defined)

cancel_port_forward_request(*address*, *port*)
The client would like to cancel a previous port-forwarding request. If the given address and port is being forwarded across this ssh connection, the port should be closed.

Parameters

- **address** (*str*) – the forwarded address
- **port** (*int*) – the forwarded port

check_auth_interactive(*username*, *submethods*)
Begin an interactive authentication challenge, if supported. You should override this method in server mode if you want to support the `"keyboard-interactive"` auth type, which requires you to send a series of questions for the client to answer.

Return `AUTH_FAILED` if this auth method isn't supported. Otherwise, you should return an `InteractiveQuery` object containing the prompts and instructions for the user. The response will be sent via a call to `check_auth_interactive_response`.

The default implementation always returns `AUTH_FAILED`.

Parameters

- **username** (*str*) – the username of the authenticating client
- **submethods** (*str*) – a comma-separated list of methods preferred by the client (usually empty)

Returns `AUTH_FAILED` if this auth method isn't supported; otherwise an object containing queries for the user

Return type int or `InteractiveQuery`

check_auth_interactive_response(*responses*)
Continue or finish an interactive authentication challenge, if supported. You should override this method in server mode if you want to support the `"keyboard-interactive"` auth type.

Return `AUTH_FAILED` if the responses are not accepted, `AUTH_SUCCESSFUL` if the responses are accepted and complete the authentication, or `AUTH_PARTIALLY_SUCCESSFUL` if your authentication is stateful, and this set of responses is accepted for authentication, but more authentication is required. (In this

latter case, get_allowed_auths will be called to report to the client what options it has for continuing the authentication.)

If you wish to continue interactive authentication with more questions, you may return an InteractiveQuery object, which should cause the client to respond with more answers, calling this method again. This cycle can continue indefinitely.

The default implementation always returns AUTH_FAILED.

Parameters **responses** (*list*) – list of str responses from the client

Returns AUTH_FAILED if the authentication fails; AUTH_SUCCESSFUL if it succeeds; AUTH_PARTIALLY_SUCCESSFUL if the interactive auth is successful, but authentication must continue; otherwise an object containing queries for the user

Return type int or InteractiveQuery

check_auth_none(*username*)

Determine if a client may open channels with no (further) authentication.

Return AUTH_FAILED if the client must authenticate, or AUTH_SUCCESSFUL if it's okay for the client to not authenticate.

The default implementation always returns AUTH_FAILED.

Parameters **username** (*str*) – the username of the client.

Returns AUTH_FAILED if the authentication fails; AUTH_SUCCESSFUL if it succeeds.

Return type int

check_auth_password(*username*, *password*)

Determine if a given username and password supplied by the client is acceptable for use in authentication.

Return AUTH_FAILED if the password is not accepted, AUTH_SUCCESSFUL if the password is accepted and completes the authentication, or AUTH_PARTIALLY_SUCCESSFUL if your authentication is stateful, and this key is accepted for authentication, but more authentication is required. (In this latter case, get_allowed_auths will be called to report to the client what options it has for continuing the authentication.)

The default implementation always returns AUTH_FAILED.

Parameters

- **username** (*str*) – the username of the authenticating client.
- **password** (*str*) – the password given by the client.

Returns AUTH_FAILED if the authentication fails; AUTH_SUCCESSFUL if it succeeds; AUTH_PARTIALLY_SUCCESSFUL if the password auth is successful, but authentication must continue.

Return type int

check_auth_publickey(*username*, *key*)

Determine if a given key supplied by the client is acceptable for use in authentication. You should override this method in server mode to check the username and key and decide if you would accept a signature made using this key.

Return AUTH_FAILED if the key is not accepted, AUTH_SUCCESSFUL if the key is accepted and completes the authentication, or AUTH_PARTIALLY_SUCCESSFUL if your authentication is stateful, and this password is accepted for authentication, but more authentication is required. (In this latter case, get_allowed_auths will be called to report to the client what options it has for continuing the authentication.)

Note that you don't have to actually verify any key signtature here. If you're willing to accept the key, Paramiko will do the work of verifying the client's signature.

The default implementation always returns `AUTH_FAILED`.

Parameters

- **username** (*str*) – the username of the authenticating client
- **key** (*.PKey*) – the key object provided by the client

Returns `AUTH_FAILED` if the client can't authenticate with this key; `AUTH_SUCCESSFUL` if it can; `AUTH_PARTIALLY_SUCCESSFUL` if it can authenticate with this key but must continue with authentication

Return type int

check_channel_direct_tcpip_request(*chanid*, *origin*, *destination*)

Determine if a local port forwarding channel will be granted, and return `OPEN_SUCCEEDED` or an error code. This method is called in server mode when the client requests a channel, after authentication is complete.

The `chanid` parameter is a small number that uniquely identifies the channel within a `Transport`. A `Channel` object is not created unless this method returns `OPEN_SUCCEEDED` – once a `Channel` object is created, you can call `Channel.get_id` to retrieve the channel ID.

The origin and destination parameters are (ip_address, port) tuples that correspond to both ends of the TCP connection in the forwarding tunnel.

The return value should either be `OPEN_SUCCEEDED` (or `0`) to allow the channel request, or one of the following error codes to reject it:

- `OPEN_FAILED_ADMINISTRATIVELY_PROHIBITED`
- `OPEN_FAILED_CONNECT_FAILED`
- `OPEN_FAILED_UNKNOWN_CHANNEL_TYPE`
- `OPEN_FAILED_RESOURCE_SHORTAGE`

The default implementation always returns `OPEN_FAILED_ADMINISTRATIVELY_PROHIBITED`.

Parameters

- **chanid** (*int*) – ID of the channel
- **origin** (*tuple*) – 2-tuple containing the IP address and port of the originator (client side)
- **destination** (*tuple*) – 2-tuple containing the IP address and port of the destination (server side)

Returns an `int` success or failure code (listed above)

check_channel_env_request(*channel*, *name*, *value*)

Check whether a given environment variable can be specified for the given channel. This method should return `True` if the server is willing to set the specified environment variable. Note that some environment variables (e.g., PATH) can be exceedingly dangerous, so blindly allowing the client to set the environment is almost certainly not a good idea.

The default implementation always returns `False`.

Parameters

- **channel** – the `Channel` the env request arrived on
- **name** (*str*) – name

- **value** (*str*) – Channel value

Returns A boolean

check_channel_exec_request(*channel*, *command*)
Determine if a shell command will be executed for the client. If this method returns `True`, the channel should be connected to the stdin, stdout, and stderr of the shell command.

The default implementation always returns `False`.

Parameters

- **channel** (.*Channel*) – the `Channel` the request arrived on.
- **command** (*str*) – the command to execute.

Returns `True` if this channel is now hooked up to the stdin, stdout, and stderr of the executing command; `False` if the command will not be executed.

New in version 1.1.

check_channel_forward_agent_request(*channel*)
Determine if the client will be provided with an forward agent session. If this method returns `True`, the server will allow SSH Agent forwarding.

The default implementation always returns `False`.

Parameters **channel** (.*Channel*) – the `Channel` the request arrived on

Returns `True` if the AgentForward was loaded; `False` if not

check_channel_pty_request(*channel*, *term*, *width*, *height*, *pixelwidth*, *pixelheight*, *modes*)
Determine if a pseudo-terminal of the given dimensions (usually requested for shell access) can be provided on the given channel.

The default implementation always returns `False`.

Parameters

- **channel** (.*Channel*) – the `Channel` the pty request arrived on.
- **term** (*str*) – type of terminal requested (for example, `"vt100"`).
- **width** (*int*) – width of screen in characters.
- **height** (*int*) – height of screen in characters.
- **pixelwidth** (*int*) – width of screen in pixels, if known (may be `0` if unknown).
- **pixelheight** (*int*) – height of screen in pixels, if known (may be `0` if unknown).

Returns `True` if the psuedo-terminal has been allocated; `False` otherwise.

check_channel_request(*kind*, *chanid*)
Determine if a channel request of a given type will be granted, and return `OPEN_SUCCEEDED` or an error code. This method is called in server mode when the client requests a channel, after authentication is complete.

If you allow channel requests (and an ssh server that didn't would be useless), you should also override some of the channel request methods below, which are used to determine which services will be allowed on a given channel:

- `check_channel_pty_request`
- `check_channel_shell_request`
- `check_channel_subsystem_request`

- check_channel_window_change_request
- check_channel_x11_request
- check_channel_forward_agent_request

The chanid parameter is a small number that uniquely identifies the channel within a Transport. A Channel object is not created unless this method returns OPEN_SUCCEEDED – once a Channel object is created, you can call Channel.get_id to retrieve the channel ID.

The return value should either be OPEN_SUCCEEDED (or 0) to allow the channel request, or one of the following error codes to reject it:

- OPEN_FAILED_ADMINISTRATIVELY_PROHIBITED
- OPEN_FAILED_CONNECT_FAILED
- OPEN_FAILED_UNKNOWN_CHANNEL_TYPE
- OPEN_FAILED_RESOURCE_SHORTAGE

The default implementation always returns OPEN_FAILED_ADMINISTRATIVELY_PROHIBITED.

Parameters

- **kind** (*str*) – the kind of channel the client would like to open (usually "session").
- **chanid** (*int*) – ID of the channel

Returns an int success or failure code (listed above)

check_channel_shell_request (*channel*)

Determine if a shell will be provided to the client on the given channel. If this method returns True, the channel should be connected to the stdin/stdout of a shell (or something that acts like a shell).

The default implementation always returns False.

Parameters **channel** (*.Channel*) – the Channel the request arrived on.

Returns True if this channel is now hooked up to a shell; False if a shell can't or won't be provided.

check_channel_subsystem_request (*channel*, *name*)

Determine if a requested subsystem will be provided to the client on the given channel. If this method returns True, all future I/O through this channel will be assumed to be connected to the requested subsystem. An example of a subsystem is sftp.

The default implementation checks for a subsystem handler assigned via Transport.set_subsystem_handler. If one has been set, the handler is invoked and this method returns True. Otherwise it returns False.

Note: Because the default implementation uses the Transport to identify valid subsystems, you probably won't need to override this method.

Parameters

- **channel** (*.Channel*) – the Channel the pty request arrived on.
- **name** (*str*) – name of the requested subsystem.

Returns True if this channel is now hooked up to the requested subsystem; False if that subsystem can't or won't be provided.

check_channel_window_change_request(*channel*, *width*, *height*, *pixelwidth*, *pixelheight*)
Determine if the pseudo-terminal on the given channel can be resized. This only makes sense if a pty was previously allocated on it.

The default implementation always returns `False`.

Parameters

- **channel** (*.Channel*) – the `Channel` the pty request arrived on.
- **width** (*int*) – width of screen in characters.
- **height** (*int*) – height of screen in characters.
- **pixelwidth** (*int*) – width of screen in pixels, if known (may be `0` if unknown).
- **pixelheight** (*int*) – height of screen in pixels, if known (may be `0` if unknown).

Returns `True` if the terminal was resized; `False` if not.

check_channel_x11_request(*channel*, *single_connection*, *auth_protocol*, *auth_cookie*, *screen_number*)
Determine if the client will be provided with an X11 session. If this method returns `True`, X11 applications should be routed through new SSH channels, using `Transport.open_x11_channel`.

The default implementation always returns `False`.

Parameters

- **channel** (*.Channel*) – the `Channel` the X11 request arrived on
- **single_connection** (*bool*) – `True` if only a single X11 channel should be opened, else `False`.
- **auth_protocol** (*str*) – the protocol used for X11 authentication
- **auth_cookie** (*str*) – the cookie used to authenticate to X11
- **screen_number** (*int*) – the number of the X11 screen to connect to

Returns `True` if the X11 session was opened; `False` if not

check_global_request(*kind*, *msg*)
Handle a global request of the given `kind`. This method is called in server mode and client mode, whenever the remote host makes a global request. If there are any arguments to the request, they will be in `msg`.

There aren't any useful global requests defined, aside from port forwarding, so usually this type of request is an extension to the protocol.

If the request was successful and you would like to return contextual data to the remote host, return a tuple. Items in the tuple will be sent back with the successful result. (Note that the items in the tuple can only be strings, ints, longs, or bools.)

The default implementation always returns `False`, indicating that it does not support any global requests.

Note: Port forwarding requests are handled separately, in `check_port_forward_request`.

Parameters

- **kind** (*str*) – the kind of global request being made.
- **msg** (*.Message*) – any extra arguments to the request.

Returns `True` or a `tuple` of data if the request was granted; `False` otherwise.

check_port_forward_request(*address*, *port*)
Handle a request for port forwarding. The client is asking that connections to the given address and port be forwarded back across this ssh connection. An address of `"0.0.0.0"` indicates a global address (any address associated with this server) and a port of `0` indicates that no specific port is requested (usually the OS will pick a port).

The default implementation always returns `False`, rejecting the port forwarding request. If the request is accepted, you should return the port opened for listening.

Parameters

- **address** (*str*) – the requested address
- **port** (*int*) – the requested port

Returns the port number (`int`) that was opened for listening, or `False` to reject

get_allowed_auths(*username*)
Return a list of authentication methods supported by the server. This list is sent to clients attempting to authenticate, to inform them of authentication methods that might be successful.

The "list" is actually a string of comma-separated names of types of authentication. Possible values are `"password"`, `"publickey"`, and `"none"`.

The default implementation always returns `"password"`.

Parameters **username** (*str*) – the username requesting authentication.

Returns a comma-separated `str` of authentication types

class paramiko.server.**SubsystemHandler**(*channel*, *name*, *server*)
Handler for a subsytem in server mode. If you create a subclass of this class and pass it to `Transport.set_subsystem_handler`, an object of this class will be created for each request for this subsystem. Each new object will be executed within its own new thread by calling `start_subsystem`. When that method completes, the channel is closed.

For example, if you made a subclass `MP3Handler` and registered it as the handler for subsystem `"mp3"`, then whenever a client has successfully authenticated and requests subsytem `"mp3"`, an object of class `MP3Handler` will be created, and `start_subsystem` will be called on it from a new thread.

__init__(*channel*, *name*, *server*)
Create a new handler for a channel. This is used by `ServerInterface` to start up a new handler when a channel requests this subsystem. You don't need to override this method, but if you do, be sure to pass the `channel` and `name` parameters through to the original `__init__` method here.

Parameters

- **channel** (*.Channel*) – the channel associated with this subsystem request.
- **name** (*str*) – name of the requested subsystem.
- **server** (*.ServerInterface*) – the server object for the session that started this subsystem

finish_subsystem()
Perform any cleanup at the end of a subsystem. The default implementation just closes the channel.

New in version 1.1.

get_server()
Return the `ServerInterface` object associated with this channel and subsystem.

start_subsystem(*name*, *transport*, *channel*)
Process an ssh subsystem in server mode. This method is called on a new object (and in a new thread)

for each subsystem request. It is assumed that all subsystem logic will take place here, and when the subsystem is finished, this method will return. After this method returns, the channel is closed.

The combination of `transport` and `channel` are unique; this handler corresponds to exactly one `Channel` on one `Transport`.

Note: It is the responsibility of this method to exit if the underlying `Transport` is closed. This can be done by checking `Transport.is_active` or noticing an EOF on the `Channel`. If this method loops forever without checking for this case, your Python interpreter may refuse to exit because this thread will still be running.

Parameters

- **name** (*str*) – name of the requested subsystem.
- **transport** (*.Transport*) – the server-mode `Transport`.
- **channel** (*.Channel*) – the channel associated with this subsystem request.

1.3.4 SFTP

class `paramiko.sftp_client.`**`SFTP`**(*sock*)
An alias for `SFTPClient` for backwards compatability.

class `paramiko.sftp_client.`**`SFTPClient`**(*sock*)
SFTP client object.

Used to open an SFTP session across an open SSH `Transport` and perform remote file operations.

`__init__`(*sock*)
Create an SFTP client from an existing `Channel`. The channel should already have requested the `"sftp"` subsystem.

An alternate way to create an SFTP client context is by using `from_transport`.

Parameters **sock** (*.Channel*) – an open `Channel` using the `"sftp"` subsystem

Raises SSHException if there's an exception while negotiating sftp

`chdir`(*path=None*)
Change the "current directory" of this SFTP session. Since SFTP doesn't really have the concept of a current working directory, this is emulated by Paramiko. Once you use this method to set a working directory, all operations on this `SFTPClient` object will be relative to that path. You can pass in `None` to stop using a current working directory.

Parameters **path** (*str*) – new current working directory

Raises IOError if the requested path doesn't exist on the server

New in version 1.4.

`chmod`(*path*, *mode*)
Change the mode (permissions) of a file. The permissions are unix-style and identical to those used by Python's `os.chmod` function.

Parameters

- **path** (*str*) – path of the file to change the permissions of
- **mode** (*int*) – new permissions

chown (*path*, *uid*, *gid*)
Change the owner (uid) and group (gid) of a file. As with Python's os.chown function, you must pass both arguments, so if you only want to change one, use stat first to retrieve the current owner and group.

Parameters

- **path** (*str*) – path of the file to change the owner and group of
- **uid** (*int*) – new owner's uid
- **gid** (*int*) – new group id

close ()
Close the SFTP session and its underlying channel.

New in version 1.4.

file (*filename*, *mode='r'*, *bufsize=-1*)
Open a file on the remote server. The arguments are the same as for Python's built-in file (aka open). A file-like object is returned, which closely mimics the behavior of a normal Python file object, including the ability to be used as a context manager.

The mode indicates how the file is to be opened: 'r' for reading, 'w' for writing (truncating an existing file), 'a' for appending, 'r+' for reading/writing, 'w+' for reading/writing (truncating an existing file), 'a+' for reading/appending. The Python 'b' flag is ignored, since SSH treats all files as binary. The 'U' flag is supported in a compatible way.

Since 1.5.2, an 'x' flag indicates that the operation should only succeed if the file was created and did not previously exist. This has no direct mapping to Python's file flags, but is commonly known as the O_EXCL flag in posix.

The file will be buffered in standard Python style by default, but can be altered with the bufsize parameter. 0 turns off buffering, 1 uses line buffering, and any number greater than 1 (>1) uses that specific buffer size.

Parameters

- **filename** (*str*) – name of the file to open
- **mode** (*str*) – mode (Python-style) to open in
- **bufsize** (*int*) – desired buffering (-1 = default buffer size)

Returns an SFTPFile object representing the open file

Raises IOError if the file could not be opened.

classmethod from_transport (*t*)
Create an SFTP client channel from an open Transport.

Parameters **t** (*.Transport*) – an open Transport which is already authenticated

Returns a new SFTPClient object, referring to an sftp session (channel) across the transport

get (*remotepath*, *localpath*, *callback=None*)
Copy a remote file (remotepath) from the SFTP server to the local host as localpath. Any exception raised by operations will be passed through. This method is primarily provided as a convenience.

Parameters

- **remotepath** (*str*) – the remote file to copy
- **localpath** (*str*) – the destination path on the local host
- **callback** (*callable*) – optional callback function (form: func(int, int)) that accepts the bytes transferred so far and the total bytes to be transferred

New in version 1.4.

Changed in version 1.7.4: Added the `callback` param

get_channel()
Return the underlying `Channel` object for this SFTP session. This might be useful for doing things like setting a timeout on the channel.

New in version 1.7.1.

getcwd()
Return the "current working directory" for this SFTP session, as emulated by Paramiko. If no directory has been set with `chdir`, this method will return `None`.

New in version 1.4.

getfo(*remotepath*, *fl*, *callback=None*)
Copy a remote file (`remotepath`) from the SFTP server and write to an open file or file-like object, `fl`. Any exception raised by operations will be passed through. This method is primarily provided as a convenience.

Parameters

- **remotepath** (*object*) – opened file or file-like object to copy to
- **fl** (*str*) – the destination path on the local host or open file object
- **callback** (*callable*) – optional callback function (form: `func(int, int)`) that accepts the bytes transferred so far and the total bytes to be transferred

Returns the `number` of bytes written to the opened file object

New in version 1.10.

listdir(*path='.'*)
Return a list containing the names of the entries in the given `path`.

The list is in arbitrary order. It does not include the special entries `'.'` and `'..'` even if they are present in the folder. This method is meant to mirror `os.listdir` as closely as possible. For a list of full `SFTPAttributes` objects, see `listdir_attr`.

Parameters **path** (*str*) – path to list (defaults to `'.'`)

listdir_attr(*path='.'*)
Return a list containing `SFTPAttributes` objects corresponding to files in the given `path`. The list is in arbitrary order. It does not include the special entries `'.'` and `'..'` even if they are present in the folder.

The returned `SFTPAttributes` objects will each have an additional field: `longname`, which may contain a formatted string of the file's attributes, in unix format. The content of this string will probably depend on the SFTP server implementation.

Parameters **path** (*str*) – path to list (defaults to `'.'`)

Returns list of `SFTPAttributes` objects

New in version 1.2.

lstat(*path*)
Retrieve information about a file on the remote system, without following symbolic links (shortcuts). This otherwise behaves exactly the same as `stat`.

Parameters **path** (*str*) – the filename to stat

Returns an `SFTPAttributes` object containing attributes about the given file

mkdir (*path*, *mode=511*)
Create a folder (directory) named `path` with numeric mode `mode`. The default mode is 0777 (octal). On some systems, mode is ignored. Where it is used, the current umask value is first masked out.

Parameters

- **path** (*str*) – name of the folder to create
- **mode** (*int*) – permissions (posix-style) for the newly-created folder

normalize (*path*)
Return the normalized path (on the server) of a given path. This can be used to quickly resolve symbolic links or determine what the server is considering to be the "current folder" (by passing `'.'` as `path`).

Parameters **path** (*str*) – path to be normalized

Returns normalized form of the given path (as a `str`)

Raises IOError if the path can't be resolved on the server

open (*filename*, *mode='r'*, *bufsize=-1*)
Open a file on the remote server. The arguments are the same as for Python's built-in `file` (aka `open`). A file-like object is returned, which closely mimics the behavior of a normal Python file object, including the ability to be used as a context manager.

The mode indicates how the file is to be opened: `'r'` for reading, `'w'` for writing (truncating an existing file), `'a'` for appending, `'r+'` for reading/writing, `'w+'` for reading/writing (truncating an existing file), `'a+'` for reading/appending. The Python `'b'` flag is ignored, since SSH treats all files as binary. The `'U'` flag is supported in a compatible way.

Since 1.5.2, an `'x'` flag indicates that the operation should only succeed if the file was created and did not previously exist. This has no direct mapping to Python's file flags, but is commonly known as the `O_EXCL` flag in posix.

The file will be buffered in standard Python style by default, but can be altered with the `bufsize` parameter. `0` turns off buffering, `1` uses line buffering, and any number greater than 1 (`>1`) uses that specific buffer size.

Parameters

- **filename** (*str*) – name of the file to open
- **mode** (*str*) – mode (Python-style) to open in
- **bufsize** (*int*) – desired buffering (-1 = default buffer size)

Returns an `SFTPFile` object representing the open file

Raises IOError if the file could not be opened.

put (*localpath*, *remotepath*, *callback=None*, *confirm=True*)
Copy a local file (`localpath`) to the SFTP server as `remotepath`. Any exception raised by operations will be passed through. This method is primarily provided as a convenience.

The SFTP operations use pipelining for speed.

Parameters

- **localpath** (*str*) – the local file to copy
- **remotepath** (*str*) – the destination path on the SFTP server. Note that the filename should be included. Only specifying a directory may result in an error.
- **callback** (*callable*) – optional callback function (form: `func(int, int)`) that accepts the bytes transferred so far and the total bytes to be transferred

- **confirm** (*bool*) – whether to do a stat() on the file afterwards to confirm the file size

Returns an `SFTPAttributes` object containing attributes about the given file

New in version 1.4.

Changed in version 1.7.4: `callback` and rich attribute return value added.

Changed in version 1.7.7: `confirm` param added.

putfo (*fl*, *remotepath*, *file_size=0*, *callback=None*, *confirm=True*)

Copy the contents of an open file object (`fl`) to the SFTP server as `remotepath`. Any exception raised by operations will be passed through.

The SFTP operations use pipelining for speed.

Parameters

- **fl** (*file*) – opened file or file-like object to copy
- **remotepath** (*str*) – the destination path on the SFTP server
- **file_size** (*int*) – optional size parameter passed to callback. If none is specified, size defaults to 0
- **callback** (*callable*) – optional callback function (form: `func(int, int)`) that accepts the bytes transferred so far and the total bytes to be transferred (since 1.7.4)
- **confirm** (*bool*) – whether to do a stat() on the file afterwards to confirm the file size (since 1.7.7)

Returns an `SFTPAttributes` object containing attributes about the given file.

New in version 1.10.

readlink (*path*)

Return the target of a symbolic link (shortcut). You can use `symlink` to create these. The result may be either an absolute or relative pathname.

Parameters **path** (*str*) – path of the symbolic link file

Returns target path, as a `str`

remove (*path*)

Remove the file at the given path. This only works on files; for removing folders (directories), use `rmdir`.

Parameters **path** (*str*) – path (absolute or relative) of the file to remove

Raises IOError if the path refers to a folder (directory)

rename (*oldpath*, *newpath*)

Rename a file or folder from `oldpath` to `newpath`.

Parameters

- **oldpath** (*str*) – existing name of the file or folder
- **newpath** (*str*) – new name for the file or folder

Raises IOError if `newpath` is a folder, or something else goes wrong

rmdir (*path*)

Remove the folder named `path`.

Parameters **path** (*str*) – name of the folder to remove

stat(*path*)
Retrieve information about a file on the remote system. The return value is an object whose attributes correspond to the attributes of Python's `stat` structure as returned by `os.stat`, except that it contains fewer fields. An SFTP server may return as much or as little info as it wants, so the results may vary from server to server.

Unlike a Python `stat` object, the result may not be accessed as a tuple. This is mostly due to the author's slack factor.

The fields supported are: `st_mode`, `st_size`, `st_uid`, `st_gid`, `st_atime`, and `st_mtime`.

Parameters **path** (*str*) – the filename to stat

Returns an `SFTPAttributes` object containing attributes about the given file

symlink(*source*, *dest*)
Create a symbolic link (shortcut) of the `source` path at `destination`.

Parameters

- **source** (*str*) – path of the original file
- **dest** (*str*) – path of the newly created symlink

truncate(*path*, *size*)
Change the size of the file specified by `path`. This usually extends or shrinks the size of the file, just like the `truncate` method on Python file objects.

Parameters

- **path** (*str*) – path of the file to modify
- **size** (*int or long*) – the new size of the file

unlink(*path*)
Remove the file at the given path. This only works on files; for removing folders (directories), use `rmdir`.

Parameters **path** (*str*) – path (absolute or relative) of the file to remove

Raises IOError if the path refers to a folder (directory)

utime(*path*, *times*)
Set the access and modified times of the file specified by `path`. If `times` is `None`, then the file's access and modified times are set to the current time. Otherwise, `times` must be a 2-tuple of numbers, of the form `(atime, mtime)`, which is used to set the access and modified times, respectively. This bizarre API is mimicked from Python for the sake of consistency – I apologize.

Parameters

- **path** (*str*) – path of the file to modify
- **times** (*tuple*) – `None` or a tuple of (access time, modified time) in standard internet epoch time (seconds since 01 January 1970 GMT)

Server-mode SFTP support.

class paramiko.sftp_server.**SFTPServer**(*channel*, *name*, *server*, *sftp_si=<class 'paramiko.sftp_si.SFTPServerInterface'>*, **largs*, ***kwargs*)
Server-side SFTP subsystem support. Since this is a `SubsystemHandler`, it can be (and is meant to be) set as the handler for `"sftp"` requests. Use `Transport.set_subsystem_handler` to activate this class.

__init__(*channel*, *name*, *server*, *sftp_si=<class 'paramiko.sftp_si.SFTPServerInterface'>*, **largs*, ***kwargs*)
The constructor for SFTPServer is meant to be called from within the `Transport` as a subsystem han-

dler. `server` and any additional parameters or keyword parameters are passed from the original call to `Transport.set_subsystem_handler`.

Parameters

- **channel** (*.Channel*) – channel passed from the `Transport`.
- **name** (*str*) – name of the requested subsystem.
- **server** (*.ServerInterface*) – the server object associated with this channel and subsystem
- **sftp_si** (*class*) – a subclass of `SFTPServerInterface` to use for handling individual requests.

static convert_errno(*e*)

Convert an errno value (as from an `OSError` or `IOError`) into a standard SFTP result code. This is a convenience function for trapping exceptions in server code and returning an appropriate result.

Parameters **e** (*int*) – an errno code, as from `OSError.errno`.

Returns an `int` SFTP error code like `SFTP_NO_SUCH_FILE`.

static set_file_attr(*filename*, *attr*)

Change a file's attributes on the local filesystem. The contents of `attr` are used to change the permissions, owner, group ownership, and/or modification & access time of the file, depending on which attributes are present in `attr`.

This is meant to be a handy helper function for translating SFTP file requests into local file operations.

Parameters

- **filename** (*str*) – name of the file to alter (should usually be an absolute path).
- **attr** (*.SFTPAttributes*) – attributes to change.

class paramiko.sftp_attr.**SFTPAttributes**

Representation of the attributes of a file (or proxied file) for SFTP in client or server mode. It attemps to mirror the object returned by `os.stat` as closely as possible, so it may have the following fields, with the same meanings as those returned by an `os.stat` object:

- `st_size`
- `st_uid`
- `st_gid`
- `st_mode`
- `st_atime`
- `st_mtime`

Because SFTP allows flags to have other arbitrary named attributes, these are stored in a dict named `attr`. Occasionally, the filename is also stored, in `filename`.

__init__()

Create a new (empty) SFTPAttributes object. All fields will be empty.

__str__()

create a unix-style long description of the file (like ls -l)

__weakref__

list of weak references to the object (if defined)

classmethod from_stat(*obj*, *filename=None*)

Create an `SFTPAttributes` object from an existing `stat` object (an object returned by `os.stat`).

Parameters

- **obj** (*object*) – an object returned by `os.stat` (or equivalent).
- **filename** (*str*) – the filename associated with this file.

Returns new `SFTPAttributes` object with the same attribute fields.

SFTP file object

class `paramiko.sftp_file.`**`SFTPFile`**(*sftp*, *handle*, *mode='r'*, *bufsize=-1*)

Bases: `paramiko.file.BufferedFile`

Proxy object for a file on the remote server, in client mode SFTP.

Instances of this class may be used as context managers in the same way that built-in Python file objects are.

`check`(*hash_algorithm*, *offset=0*, *length=0*, *block_size=0*)

Ask the server for a hash of a section of this file. This can be used to verify a successful upload or download, or for various rsync-like operations.

The file is hashed from `offset`, for `length` bytes. If `length` is 0, the remainder of the file is hashed. Thus, if both `offset` and `length` are zero, the entire file is hashed.

Normally, `block_size` will be 0 (the default), and this method will return a byte string representing the requested hash (for example, a string of length 16 for MD5, or 20 for SHA-1). If a non-zero `block_size` is given, each chunk of the file (from `offset` to `offset + length`) of `block_size` bytes is computed as a separate hash. The hash results are all concatenated and returned as a single string.

For example, `check('sha1', 0, 1024, 512)` will return a string of length 40. The first 20 bytes will be the SHA-1 of the first 512 bytes of the file, and the last 20 bytes will be the SHA-1 of the next 512 bytes.

Parameters

- **hash_algorithm** (*str*) – the name of the hash algorithm to use (normally `"sha1"` or `"md5"`)
- **offset** (*int or long*) – offset into the file to begin hashing (0 means to start from the beginning)
- **length** (*int or long*) – number of bytes to hash (0 means continue to the end of the file)
- **block_size** (*int*) – number of bytes to hash per result (must not be less than 256; 0 means to compute only one hash of the entire segment)

Returns `str` of bytes representing the hash of each block, concatenated together

Raises IOError if the server doesn't support the "check-file" extension, or possibly doesn't support the hash algorithm requested

---

Note: Many (most?) servers don't support this extension yet.

New in version 1.4.

`chmod`(*mode*)

Change the mode (permissions) of this file. The permissions are unix-style and identical to those used by Python's `os.chmod` function.

Parameters **mode** (*int*) – new permissions

`chown`(*uid*, *gid*)

Change the owner (`uid`) and group (`gid`) of this file. As with Python's `os.chown` function, you must

pass both arguments, so if you only want to change one, use `stat` first to retrieve the current owner and group.

Parameters

- **uid** (*int*) – new owner's uid
- **gid** (*int*) – new group id

close()
Close the file.

flush()
Write out any data in the write buffer. This may do nothing if write buffering is not turned on.

gettimeout()
Returns the timeout in seconds (as a `float`) associated with the socket or ssh `Channel` used for this file.

See also:

`Channel.gettimeout`

next()
Returns the next line from the input, or raises `StopIteration` when EOF is hit. Unlike Python file objects, it's okay to mix calls to `next` and `readline`.

Raises StopIteration when the end of the file is reached.

Returns a line (`str`) read from the file.

prefetch()
Pre-fetch the remaining contents of this file in anticipation of future `read` calls. If reading the entire file, pre-fetching can dramatically improve the download speed by avoiding roundtrip latency. The file's contents are incrementally buffered in a background thread.

The prefetched data is stored in a buffer until read via the `read` method. Once data has been read, it's removed from the buffer. The data may be read in a random order (using `seek`); chunks of the buffer that haven't been read will continue to be buffered.

New in version 1.5.1.

read(*size=None*)
Read at most `size` bytes from the file (less if we hit the end of the file first). If the `size` argument is negative or omitted, read all the remaining data in the file.

Note: `'b'` mode flag is ignored (`self.FLAG_BINARY` in `self._flags`), because SSH treats all files as binary, since we have no idea what encoding the file is in, or even if the file is text data.

Parameters **size** (*int*) – maximum number of bytes to read

Returns data read from the file (as bytes), or an empty string if EOF was encountered immediately

readline(*size=None*)
Read one entire line from the file. A trailing newline character is kept in the string (but may be absent when a file ends with an incomplete line). If the size argument is present and non-negative, it is a maximum byte count (including the trailing newline) and an incomplete line may be returned. An empty string is returned only when EOF is encountered immediately.

Note: Unlike stdio's `fgets`, the returned string contains null characters (`'\0'`) if they occurred in the input.

Parameters **size** (*int*) – maximum length of returned string.

Returns

next line of the file, or an empty string if the end of the file has been reached.

If the file was opened in binary (`'b'`) mode: bytes are returned Else: the encoding of the file is assumed to be UTF-8 and character strings (`str`) are returned

readlines(*sizehint=None*)

Read all remaining lines using `readline` and return them as a list. If the optional `sizehint` argument is present, instead of reading up to EOF, whole lines totalling approximately sizehint bytes (possibly after rounding up to an internal buffer size) are read.

Parameters **sizehint** (*int*) – desired maximum number of bytes to read.

Returns `list` of lines read from the file.

readv(*chunks*)

Read a set of blocks from the file by (offset, length). This is more efficient than doing a series of `seek` and `read` calls, since the prefetch machinery is used to retrieve all the requested blocks at once.

Parameters **chunks** (*list(tuple(long, int))*) – a list of (offset, length) tuples indicating which sections of the file to read

Returns a list of blocks read, in the same order as in `chunks`

New in version 1.5.4.

set_pipelined(*pipelined=True*)

Turn on/off the pipelining of write operations to this file. When pipelining is on, paramiko won't wait for the server response after each write operation. Instead, they're collected as they come in. At the first non-write operation (including `close`), all remaining server responses are collected. This means that if there was an error with one of your later writes, an exception might be thrown from within `close` instead of `write`.

By default, files are not pipelined.

Parameters **pipelined** (*bool*) – `True` if pipelining should be turned on for this file; `False` otherwise

New in version 1.5.

setblocking(*blocking*)

Set blocking or non-blocking mode on the underiying socket or ssh `Channel`.

Parameters **blocking** (*int*) – 0 to set non-blocking mode; non-0 to set blocking mode.

See also:

`Channel.setblocking`

settimeout(*timeout*)

Set a timeout on read/write operations on the underlying socket or ssh `Channel`.

Parameters **timeout** (*float*) – seconds to wait for a pending read/write operation before raising `socket.timeout`, or `None` for no timeout

See also:

`Channel.settimeout`

stat()
Retrieve information about this file from the remote system. This is exactly like SFTPClient.stat, except that it operates on an already-open file.

Returns an SFTPAttributes object containing attributes about this file.

tell()
Return the file's current position. This may not be accurate or useful if the underlying file doesn't support random access, or was opened in append mode.

Returns file position (number of bytes).

truncate(*size*)
Change the size of this file. This usually extends or shrinks the size of the file, just like the truncate() method on Python file objects.

Parameters **size** (*int or long*) – the new size of the file

utime(*times*)
Set the access and modified times of this file. If times is None, then the file's access and modified times are set to the current time. Otherwise, times must be a 2-tuple of numbers, of the form (atime, mtime), which is used to set the access and modified times, respectively. This bizarre API is mimicked from Python for the sake of consistency – I apologize.

Parameters **times** (*tuple*) – None or a tuple of (access time, modified time) in standard internet epoch time (seconds since 01 January 1970 GMT)

write(*data*)
Write data to the file. If write buffering is on (bufsize was specified and non-zero), some or all of the data may not actually be written yet. (Use flush or close to force buffered data to be written out.)

Parameters **data** (*str*) – data to write

writelines(*sequence*)
Write a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. (The name is intended to match readlines; writelines does not add line separators.)

Parameters **sequence** (*iterable*) – an iterable sequence of strings.

xreadlines()
Identical to iter(f). This is a deprecated file interface that predates Python iterator support.

Abstraction of an SFTP file handle (for server mode).

class paramiko.sftp_handle.**SFTPHandle**(*flags=0*)
Abstract object representing a handle to an open file (or folder) in an SFTP server implementation. Each handle has a string representation used by the client to refer to the underlying file.

Server implementations can (and should) subclass SFTPHandle to implement features of a file handle, like stat or chattr.

__init__(*flags=0*)
Create a new file handle representing a local file being served over SFTP. If flags is passed in, it's used to determine if the file is open in append mode.

Parameters **flags** (*int*) – optional flags as passed to SFTPServerInterface.open

__weakref__
list of weak references to the object (if defined)

chattr(*attr*)
Change the attributes of this file. The `attr` object will contain only those fields provided by the client in its request, so you should check for the presence of fields before using them.

Parameters **attr** (*.SFTPAttributes*) – the attributes to change on this file.

Returns an `int` error code like `SFTP_OK`.

close()
When a client closes a file, this method is called on the handle. Normally you would use this method to close the underlying OS level file object(s).

The default implementation checks for attributes on `self` named `readfile` and/or `writefile`, and if either or both are present, their `close()` methods are called. This means that if you are using the default implementations of `read` and `write`, this method's default implementation should be fine also.

read(*offset*, *length*)
Read up to `length` bytes from this file, starting at position `offset`. The offset may be a Python long, since SFTP allows it to be 64 bits.

If the end of the file has been reached, this method may return an empty string to signify EOF, or it may also return `SFTP_EOF`.

The default implementation checks for an attribute on `self` named `readfile`, and if present, performs the read operation on the Python file-like object found there. (This is meant as a time saver for the common case where you are wrapping a Python file object.)

Parameters

- **offset** (*int or long*) – position in the file to start reading from.
- **length** (*int*) – number of bytes to attempt to read.

Returns data read from the file, or an SFTP error code, as a `str`.

stat()
Return an `SFTPAttributes` object referring to this open file, or an error code. This is equivalent to `SFTPServerInterface.stat`, except it's called on an open file instead of a path.

Returns an attributes object for the given file, or an SFTP error code (like `SFTP_PERMISSION_DENIED`).

Return type `SFTPAttributes` or error code

write(*offset*, *data*)
Write `data` into this file at position `offset`. Extending the file past its original end is expected. Unlike Python's normal `write()` methods, this method cannot do a partial write: it must write all of `data` or else return an error.

The default implementation checks for an attribute on `self` named `writefile`, and if present, performs the write operation on the Python file-like object found there. The attribute is named differently from `readfile` to make it easy to implement read-only (or write-only) files, but if both attributes are present, they should refer to the same file.

Parameters

- **offset** (*int or long*) – position in the file to start reading from.
- **data** (*str*) – data to write into the file.

Returns an SFTP error code like `SFTP_OK`.

An interface to override for SFTP server support.

class paramiko.sftp_si.**SFTPServerInterface**(*server*, **largs*, ***kwargs*)

This class defines an interface for controlling the behavior of paramiko when using the SFTPServer subsystem to provide an SFTP server.

Methods on this class are called from the SFTP session's thread, so you can block as long as necessary without affecting other sessions (even other SFTP sessions). However, raising an exception will usually cause the SFTP session to abruptly end, so you will usually want to catch exceptions and return an appropriate error code.

All paths are in string form instead of unicode because not all SFTP clients & servers obey the requirement that paths be encoded in UTF-8.

__init__(*server*, **largs*, ***kwargs*)

Create a new SFTPServerInterface object. This method does nothing by default and is meant to be overridden by subclasses.

Parameters **server** (*.ServerInterface*) – the server object associated with this channel and SFTP subsystem

__weakref__

list of weak references to the object (if defined)

canonicalize(*path*)

Return the canonical form of a path on the server. For example, if the server's home folder is /home/foo, the path "../betty" would be canonicalized to "/home/betty". Note the obvious security issues: if you're serving files only from a specific folder, you probably don't want this method to reveal path names outside that folder.

You may find the Python methods in os.path useful, especially os.path.normpath and os.path.realpath.

The default implementation returns os.path.normpath('/' + path).

chattr(*path*, *attr*)

Change the attributes of a file. The attr object will contain only those fields provided by the client in its request, so you should check for the presence of fields before using them.

Parameters

- **path** (*str*) – requested path (relative or absolute) of the file to change.
- **attr** – requested attributes to change on the file (an SFTPAttributes object)

Returns an error code int like SFTP_OK.

list_folder(*path*)

Return a list of files within a given folder. The path will use posix notation ("/" separates folder names) and may be an absolute or relative path.

The list of files is expected to be a list of SFTPAttributes objects, which are similar in structure to the objects returned by os.stat. In addition, each object should have its filename field filled in, since this is important to a directory listing and not normally present in os.stat results. The method SFTPAttributes.from_stat will usually do what you want.

In case of an error, you should return one of the SFTP_* error codes, such as SFTP_PERMISSION_DENIED.

Parameters **path** (*str*) – the requested path (relative or absolute) to be listed.

Returns a list of the files in the given folder, using SFTPAttributes objects.

Note: You should normalize the given path first (see the os.path module) and check appropriate permissions before returning the list of files. Be careful of malicious clients attempting to use relative

paths to escape restricted folders, if you're doing a direct translation from the SFTP server path to your local filesystem.

---

lstat(*path*)

Return an SFTPAttributes object for a path on the server, or an error code. If your server supports symbolic links (also known as "aliases"), you should not follow them – instead, you should return data on the symlink or alias itself. (stat is the corresponding call that follows symlinks/aliases.)

Parameters **path** (*str*) – the requested path (relative or absolute) to fetch file statistics for.

Returns an SFTPAttributes object for the given file, or an SFTP error code (like SFTP_PERMISSION_DENIED).

mkdir(*path*, *attr*)

Create a new directory with the given attributes. The attr object may be considered a "hint" and ignored.

The attr object will contain only those fields provided by the client in its request, so you should use hasattr to check for the presense of fields before using them. In some cases, the attr object may be completely empty.

Parameters

- **path** (*str*) – requested path (relative or absolute) of the new folder.
- **attr** (*.SFTPAttributes*) – requested attributes of the new folder.

Returns an SFTP error code int like SFTP_OK.

open(*path*, *flags*, *attr*)

Open a file on the server and create a handle for future operations on that file. On success, a new object subclassed from SFTPHandle should be returned. This handle will be used for future operations on the file (read, write, etc). On failure, an error code such as SFTP_PERMISSION_DENIED should be returned.

flags contains the requested mode for opening (read-only, write-append, etc) as a bitset of flags from the os module:

- os.O_RDONLY
- os.O_WRONLY
- os.O_RDWR
- os.O_APPEND
- os.O_CREAT
- os.O_TRUNC
- os.O_EXCL

(One of os.O_RDONLY, os.O_WRONLY, or os.O_RDWR will always be set.)

The attr object contains requested attributes of the file if it has to be created. Some or all attribute fields may be missing if the client didn't specify them.

Note: The SFTP protocol defines all files to be in "binary" mode. There is no equivalent to Python's "text" mode.

Parameters

- **path** (*str*) – the requested path (relative or absolute) of the file to be opened.

- **flags** (*int*) – flags or'd together from the `os` module indicating the requested mode for opening the file.
- **attr** (*.SFTPAttributes*) – requested attributes of the file if it is newly created.

Returns a new `SFTPHandle` or error code.

readlink(*path*)

Return the target of a symbolic link (or shortcut) on the server. If the specified path doesn't refer to a symbolic link, an error should be returned.

Parameters **path** (*str*) – path (relative or absolute) of the symbolic link.

Returns the target `str` path of the symbolic link, or an error code like `SFTP_NO_SUCH_FILE`.

remove(*path*)

Delete a file, if possible.

Parameters **path** (*str*) – the requested path (relative or absolute) of the file to delete.

Returns an SFTP error code `int` like `SFTP_OK`.

rename(*oldpath*, *newpath*)

Rename (or move) a file. The SFTP specification implies that this method can be used to move an existing file into a different folder, and since there's no other (easy) way to move files via SFTP, it's probably a good idea to implement "move" in this method too, even for files that cross disk partition boundaries, if at all possible.

Note: You should return an error if a file with the same name as `newpath` already exists. (The rename operation should be non-desctructive.)

Parameters

- **oldpath** (*str*) – the requested path (relative or absolute) of the existing file.
- **newpath** (*str*) – the requested new path of the file.

Returns an SFTP error code `int` like `SFTP_OK`.

rmdir(*path*)

Remove a directory if it exists. The `path` should refer to an existing, empty folder – otherwise this method should return an error.

Parameters **path** (*str*) – requested path (relative or absolute) of the folder to remove.

Returns an SFTP error code `int` like `SFTP_OK`.

session_ended()

The SFTP server session has just ended, either cleanly or via an exception. This method is meant to be overridden to perform any necessary cleanup before this `SFTPServerInterface` object is destroyed.

session_started()

The SFTP server session has just started. This method is meant to be overridden to perform any necessary setup before handling callbacks from SFTP operations.

stat(*path*)

Return an `SFTPAttributes` object for a path on the server, or an error code. If your server supports symbolic links (also known as "aliases"), you should follow them. (`lstat` is the corresponding call that doesn't follow symlinks/aliases.)

Parameters **path** (*str*) – the requested path (relative or absolute) to fetch file statistics for.

Returns an SFTPAttributes object for the given file, or an SFTP error code (like SFTP_PERMISSION_DENIED).

symlink(*target_path*, *path*)
Create a symbolic link on the server, as new pathname path, with target_path as the target of the link.

Parameters

- **target_path** (*str*) – path (relative or absolute) of the target for this new symbolic link.
- **path** (*str*) – path (relative or absolute) of the symbolic link to create.

Returns an error code int like SFTP_OK.

1.4 Miscellany

1.4.1 Buffered pipes

Attempt to generalize the "feeder" part of a Channel: an object which can be read from and closed, but is reading from a buffer fed by another thread. The read operations are blocking and can have a timeout set.

class paramiko.buffered_pipe.**BufferedPipe**
A buffer that obeys normal read (with timeout) & close semantics for a file or socket, but is fed data from another thread. This is used by Channel.

__len__()
Return the number of bytes buffered.

Returns number (int) of bytes buffered

__weakref__
list of weak references to the object (if defined)

close()
Close this pipe object. Future calls to read after the buffer has been emptied will return immediately with an empty string.

empty()
Clear out the buffer and return all data that was in it.

Returns any data that was in the buffer prior to clearing it out, as a str

feed(*data*)
Feed new data into this pipe. This method is assumed to be called from a separate thread, so synchronization is done.

Parameters **data** – the data to add, as a str

read(*nbytes*, *timeout=None*)
Read data from the pipe. The return value is a string representing the data received. The maximum amount of data to be received at once is specified by nbytes. If a string of length zero is returned, the pipe has been closed.

The optional timeout argument can be a nonnegative float expressing seconds, or None for no timeout. If a float is given, a PipeTimeout will be raised if the timeout period value has elapsed before any data arrives.

Parameters

- **nbytes** (*int*) – maximum number of bytes to read

- **timeout** (*float*) – maximum seconds to wait (or `None`, the default, to wait forever)

Returns the read data, as a `str`

Raises PipeTimeout if a timeout was specified and no data was ready before that timeout

read_ready()
Returns true if data is buffered and ready to be read from this feeder. A `False` result does not mean that the feeder has closed; it means you may need to wait before more data arrives.

Returns `True` if a `read` call would immediately return at least one byte; `False` otherwise.

set_event(*event*)
Set an event on this buffer. When data is ready to be read (or the buffer has been closed), the event will be set. When no data is ready, the event will be cleared.

Parameters **event** (*threading.Event*) – the event to set/clear

exception paramiko.buffered_pipe.**PipeTimeout**
Indicates that a timeout was reached on a read from a `BufferedPipe`.

__weakref__
list of weak references to the object (if defined)

1.4.2 Buffered files

class paramiko.file.**BufferedFile**
Reusable base class to implement Python-style file buffering around a simpler stream.

__iter__()
Returns an iterator that can be used to iterate over the lines in this file. This iterator happens to return the file itself, since a file is its own iterator.

Raises ValueError if the file is closed.

__weakref__
list of weak references to the object (if defined)

close()
Close the file. Future read and write operations will fail.

flush()
Write out any data in the write buffer. This may do nothing if write buffering is not turned on.

next()
Returns the next line from the input, or raises `StopIteration` when EOF is hit. Unlike Python file objects, it's okay to mix calls to `next` and `readline`.

Raises StopIteration when the end of the file is reached.

Returns a line (`str`) read from the file.

read(*size=None*)
Read at most `size` bytes from the file (less if we hit the end of the file first). If the `size` argument is negative or omitted, read all the remaining data in the file.

Note: `'b'` mode flag is ignored (`self.FLAG_BINARY` in `self._flags`), because SSH treats all files as binary, since we have no idea what encoding the file is in, or even if the file is text data.

Parameters **size** (*int*) – maximum number of bytes to read

Returns data read from the file (as bytes), or an empty string if EOF was encountered immediately

readline(*size=None*)

Read one entire line from the file. A trailing newline character is kept in the string (but may be absent when a file ends with an incomplete line). If the size argument is present and non-negative, it is a maximum byte count (including the trailing newline) and an incomplete line may be returned. An empty string is returned only when EOF is encountered immediately.

Note: Unlike stdio's `fgets`, the returned string contains null characters (`'\0'`) if they occurred in the input.

Parameters **size** (*int*) – maximum length of returned string.

Returns

next line of the file, or an empty string if the end of the file has been reached.

If the file was opened in binary (`'b'`) mode: bytes are returned Else: the encoding of the file is assumed to be UTF-8 and character strings (`str`) are returned

readlines(*sizehint=None*)

Read all remaining lines using `readline` and return them as a list. If the optional `sizehint` argument is present, instead of reading up to EOF, whole lines totalling approximately sizehint bytes (possibly after rounding up to an internal buffer size) are read.

Parameters **sizehint** (*int*) – desired maximum number of bytes to read.

Returns `list` of lines read from the file.

seek(*offset*, *whence=0*)

Set the file's current position, like stdio's `fseek`. Not all file objects support seeking.

Note: If a file is opened in append mode (`'a'` or `'a+'`), any seek operations will be undone at the next write (as the file position will move back to the end of the file).

Parameters

- **offset** (*int*) – position to move to within the file, relative to `whence`.
- **whence** (*int*) – type of movement: 0 = absolute; 1 = relative to the current position; 2 = relative to the end of the file.

Raises IOError if the file doesn't support random access.

tell()

Return the file's current position. This may not be accurate or useful if the underlying file doesn't support random access, or was opened in append mode.

Returns file position (`number` of bytes).

write(*data*)

Write data to the file. If write buffering is on (`bufsize` was specified and non-zero), some or all of the data may not actually be written yet. (Use `flush` or `close` to force buffered data to be written out.)

Parameters **data** (*str*) – data to write

writelines(*sequence*)
Write a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. (The name is intended to match `readlines`; `writelines` does not add line separators.)

Parameters **sequence** (*iterable*) – an iterable sequence of strings.

xreadlines()
Identical to `iter(f)`. This is a deprecated file interface that predates Python iterator support.

1.4.3 Cross-platform pipe implementations

Abstraction of a one-way pipe where the read end can be used in `select.select`. Normally this is trivial, but Windows makes it nearly impossible.

The pipe acts like an Event, which can be set or cleared. When set, the pipe will trigger as readable in `select`.

class paramiko.pipe.**WindowsPipe**
On Windows, only an OS-level "WinSock" may be used in select(), but reads and writes must be to the actual socket object.

__weakref__
list of weak references to the object (if defined)

paramiko.pipe.**make_or_pipe**(*pipe*)
wraps a pipe into two pipe-like objects which are "or"d together to affect the real pipe. if either returned pipe is set, the wrapped pipe is set. when both are cleared, the wrapped pipe is cleared.

1.4.4 Exceptions

exception paramiko.ssh_exception.**AuthenticationException**
Exception raised when authentication failed for some reason. It may be possible to retry with different credentials. (Other classes specify more specific reasons.)

New in version 1.6.

exception paramiko.ssh_exception.**BadAuthenticationType**(*explanation*, *types*)
Exception raised when an authentication type (like password) is used, but the server isn't allowing that type. (It may only allow public-key, for example.)

Variables **allowed_types** (*list*) – list of allowed authentication types provided by the server (possible values are: `"none"`, `"password"`, and `"publickey"`).

New in version 1.1.

exception paramiko.ssh_exception.**BadHostKeyException**(*hostname*, *got_key*, *expected_key*)
The host key given by the SSH server did not match what we were expecting.

Variables

- **hostname** (*str*) – the hostname of the SSH server
- **got_key** (*PKey*) – the host key presented by the server
- **expected_key** (*PKey*) – the host key expected

New in version 1.6.

exception paramiko.ssh_exception.**ChannelException**(*code*, *text*)
Exception raised when an attempt to open a new `Channel` fails.

Variables **code** (*int*) – the error code returned by the server

New in version 1.6.

exception paramiko.ssh_exception.**PartialAuthentication**(*types*)
An internal exception thrown in the case of partial authentication.

exception paramiko.ssh_exception.**PasswordRequiredException**
Exception raised when a password is needed to unlock a private key file.

exception paramiko.ssh_exception.**ProxyCommandFailure**(*command*, *error*)
The "ProxyCommand" found in the .ssh/config file returned an error.

Variables

- **command** (*str*) – The command line that is generating this exception.
- **error** (*str*) – The error captured from the proxy command output.

exception paramiko.ssh_exception.**SSHException**
Exception raised by failures in SSH2 protocol negotiation or logic errors.

__weakref__
list of weak references to the object (if defined)

p

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X