# 21.13. `ftplib` — FTP protocol client

**Source code:** Lib/ftplib.py

This module defines the class `FTP` and a few related items. The `FTP` class implements the client side of the FTP protocol. You can use this to write Python programs that perform a variety of automated FTP jobs, such as mirroring other FTP servers. It is also used by the module `urllib.request` to handle URLs that use FTP. For more information on FTP (File Transfer Protocol), see Internet **RFC 959**.

Here's a sample session using the `ftplib` module:

```
>>> from ftplib import FTP
>>> ftp = FTP('ftp.debian.org')     # connect to host, default port
>>> ftp.login()                     # user anonymous, passwd anonymous
'230 Login successful.'
>>> ftp.cwd('debian')               # change into "debian" directory
>>> ftp.retrlines('LIST')           # list directory contents
-rw-rw-r--    1 1176     1176         1063 Jun 15 10:18 README
...
drwxr-sr-x    5 1176     1176         4096 Dec 19  2000 pool
drwxr-sr-x    4 1176     1176         4096 Nov 17  2008 project
drwxr-xr-x    3 1176     1176         4096 Oct 10  2012 tools
'226 Directory send OK.'
>>> ftp.retrbinary('RETR README', open('README', 'wb').write)
'226 Transfer complete.'
>>> ftp.quit()
```

The module defines the following items:

*class* `ftplib.`**`FTP`**(*host='', user='', passwd='', acct='', timeout=None, source_address=None*)

Return a new instance of the `FTP` class. When *host* is given, the method call `connect(host)` is made. When *user* is given, additionally the method call `login(user, passwd, acct)` is made (where *passwd* and *acct* default to the empty string when not given). The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if is not specified, the global default timeout setting will be used). *source_address* is a 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.

The `FTP` class supports the `with` statement, e.g.:

```
>>> from ftplib import FTP
>>> with FTP("ftp1.at.proftpd.org") as ftp:
...     ftp.login()
...     ftp.dir()
...
'230 Anonymous login ok, restrictions apply.'
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 .
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 ..
dr-xr-xr-x   5 ftp      ftp          4096 May  6 10:43 CentOS
dr-xr-xr-x   3 ftp      ftp            18 Jul 10  2008 Fedora
>>>
```

*Changed in version 3.2:* Support for the `with` statement was added.

*Changed in version 3.3: source_address* parameter was added.

*class* `ftplib.`**`FTP_TLS`**(*host='', user='', passwd='', acct='', keyfile=None, certfile=None, context=None, timeout=None, source_address=None*)

A `FTP` subclass which adds TLS support to FTP as described in **RFC 4217**. Connect as usual to port 21 implicitly securing the FTP control connection before authenticating. Securing the data connection requires the user to explicitly ask for it by calling the `prot_p()` method. *context* is a `ssl.SSLContext` object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read Security considerations for best practices.

*keyfile* and *certfile* are a legacy alternative to *context* – they can point to PEM-formatted private key and certificate chain files (respectively) for the SSL connection.

*New in version 3.2.*

*Changed in version 3.3: source_address* parameter was added.

*Changed in version 3.4:* The class now supports hostname check with `ssl.SSLContext.check_hostname` and *Server Name Indication* (see `ssl.HAS_SNI`).

*Deprecated since version 3.6: keyfile* and *certfile* are deprecated in favor of *context*. Please use `ssl.SSLContext.load_cert_chain()` instead, or let `ssl.create_default_context()` select the system's trusted CA certificates for you.

Here's a sample session using the `FTP_TLS` class:

```
>>> ftps = FTP_TLS('ftp.pureftpd.org')
>>> ftps.login()
```

```
'230 Anonymous user logged in'
>>> ftps.prot_p()
'200 Data protection level set to "private"'
>>> ftps.nlst()
['6jack', 'OpenBSD', 'antilink', 'blogbench', 'bsdcam', 'clockspee
```

*exception* ftplib.**error_reply**

Exception raised when an unexpected reply is received from the server.

*exception* ftplib.**error_temp**

Exception raised when an error code signifying a temporary error (response codes in the range 400–499) is received.

*exception* ftplib.**error_perm**

Exception raised when an error code signifying a permanent error (response codes in the range 500–599) is received.

*exception* ftplib.**error_proto**

Exception raised when a reply is received from the server that does not fit the response specifications of the File Transfer Protocol, i.e. begin with a digit in the range 1–5.

ftplib.**all_errors**

The set of all exceptions (as a tuple) that methods of `FTP` instances may raise as a result of problems with the FTP connection (as opposed to programming errors made by the caller). This set includes the four exceptions listed above as well as `OSError`.

**See also:**

**Module** `netrc`

Parser for the `.netrc` file format. The file `.netrc` is typically used by FTP clients to load user authentication information before prompting the user.

## 21.13.1. FTP Objects

Several methods are available in two flavors: one for handling text files and another for binary files. These are named for the command which is used followed by `lines` for the text version or `binary` for the binary version.

`FTP` instances have the following methods:

FTP.**set_debuglevel**(*level*)

Set the instance’s debugging level. This controls the amount of debugging output printed. The default, `0`, produces no debugging output. A value of `1` produces a moderate amount of debugging output, generally a single line per request. A value of `2` or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

`FTP.`**`connect`**(*host=''*, *port=0*, *timeout=None*, *source_address=None*)

Connect to the given host and port. The default port number is `21`, as specified by the FTP protocol specification. It is rarely needed to specify a different port number. This function should be called only once for each instance; it should not be called at all if a host was given when the instance was created. All other methods can only be used after a connection has been made. The optional *timeout* parameter specifies a timeout in seconds for the connection attempt. If no *timeout* is passed, the global default timeout setting will be used. *source_address* is a 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.

*Changed in version 3.3: source_address* parameter was added.

`FTP.`**`getwelcome`**()

Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

`FTP.`**`login`**(*user='anonymous'*, *passwd=''*, *acct=''*)

Log in as the given *user*. The *passwd* and *acct* parameters are optional and default to the empty string. If no *user* is specified, it defaults to `'anonymous'`. If *user* is `'anonymous'`, the default *passwd* is `'anonymous@'`. This function should be called only once for each instance, after a connection has been established; it should not be called at all if a host and user were given when the instance was created. Most FTP commands are only allowed after the client has logged in. The *acct* parameter supplies “accounting information”; few systems implement this.

`FTP.`**`abort`**()

Abort a file transfer that is in progress. Using this does not always work, but it’s worth a try.

`FTP.`**`sendcmd`**(*cmd*)

Send a simple command string to the server and return the response string.

`FTP.`**`voidcmd`**(*cmd*)

Send a simple command string to the server and handle the response. Return nothing if a response code corresponding to success (codes in the range 200–299) is received. Raise `error_reply` otherwise.

`FTP.`**`retrbinary`**(*cmd*, *callback*, *blocksize=8192*, *rest=None*)

Retrieve a file in binary transfer mode. *cmd* should be an appropriate `RETR` command: `'RETR filename'`. The *callback* function is called for each block of data received, with a single bytes argument giving the data block. The optional *blocksize* argument specifies the maximum chunk size to read on the low-level socket object created to do the actual transfer (which will also be the largest size of the data blocks passed to *callback*). A reasonable default is chosen. *rest* means the same thing as in the `transfercmd()` method.

`FTP.`**`retrlines`**(*cmd*, *callback=None*)

Retrieve a file or directory listing in ASCII transfer mode. *cmd* should be an appropriate `RETR` command (see `retrbinary()`) or a command such as `LIST` or `NLST` (usually just the string `'LIST'`). `LIST` retrieves a list of files and information about those files. `NLST` retrieves a list of file names. The *callback* function is called for each line with a string argument containing the line with the trailing CRLF stripped. The default *callback* prints the line to `sys.stdout`.

`FTP.`**`set_pasv`**(*val*)

Enable “passive” mode if *val* is true, otherwise disable passive mode. Passive mode is on by default.

`FTP.`**`storbinary`**(*cmd*, *fp*, *blocksize=8192*, *callback=None*, *rest=None*)

Store a file in binary transfer mode. *cmd* should be an appropriate `STOR` command: `"STOR filename"`. *fp* is a file object (opened in binary mode) which is read until EOF using its `read()` method in blocks of size *blocksize* to provide the data to be stored. The *blocksize* argument defaults to 8192. *callback* is an optional single parameter callable that is called on each block of data after it is sent. *rest* means the same thing as in the `transfercmd()` method.

*Changed in version 3.2: rest* parameter added.

`FTP.`**`storlines`**(*cmd*, *fp*, *callback=None*)

Store a file in ASCII transfer mode. *cmd* should be an appropriate `STOR` command (see `storbinary()`). Lines are read until EOF from the file object *fp* (opened in binary mode) using its `readline()` method to provide the data to be stored. *callback* is an optional single parameter callable that is called on each line after it is sent.

`FTP.`**`transfercmd`**(*cmd*, *rest=None*)

Initiate a transfer over the data connection. If the transfer is active, send an `EPRT` or `PORT` command and the transfer command specified by *cmd*, and accept the connection. If the server is passive, send an `EPSV` or `PASV` command, connect to it, and start the transfer command. Either way, return the socket for the connection.

If optional *rest* is given, a `REST` command is sent to the server, passing *rest* as an argument. *rest* is usually a byte offset into the requested file, telling the server to restart sending the file's bytes at the requested offset, skipping over the initial bytes. Note however that **RFC 959** requires only that *rest* be a string containing characters in the printable range from ASCII code 33 to ASCII code 126. The `transfercmd()` method, therefore, converts *rest* to a string, but no check is performed on the string's contents. If the server does not recognize the `REST` command, an `error_reply` exception will be raised. If this happens, simply call `transfercmd()` without a *rest* argument.

`FTP.`**`ntransfercmd`**(*cmd*, *rest=None*)

Like `transfercmd()`, but returns a tuple of the data connection and the expected size of the data. If the expected size could not be computed, `None` will be returned as the expected size. *cmd* and *rest* means the same thing as in `transfercmd()`.

`FTP.`**`mlsd`**(*path=""*, *facts=[]*)

List a directory in a standardized format by using `MLSD` command (**RFC 3659**). If *path* is omitted the current directory is assumed. *facts* is a list of strings representing the type of information desired (e.g. `["type", "size", "perm"]`). Return a generator object yielding a tuple of two elements for every file found in path. First element is the file name, the second one is a dictionary containing facts about the file name. Content of this dictionary might be limited by the *facts* argument but server is not guaranteed to return all requested facts.

*New in version 3.3.*

`FTP.`**`nlst`**(*argument*[, ...])

Return a list of file names as returned by the `NLST` command. The optional *argument* is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the `NLST` command.

> **Note:** If your server supports the command, `mlsd()` offers a better API.

`FTP.`**`dir`**(*argument*[, ...])

Produce a directory listing as returned by the `LIST` command, printing it to standard output. The optional *argument* is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the `LIST` command. If the last argument is a function, it is used as a *callback* function as for `retrlines()`; the default prints to `sys.stdout`. This method returns `None`.

> **Note:** If your server supports the command, `mlsd()` offers a better API.

`FTP.`**`rename`**(*fromname*, *toname*)

Rename file *fromname* on the server to *toname*.

`FTP.`**`delete`**(*filename*)

Remove the file named *filename* from the server. If successful, returns the text of the response, otherwise raises `error_perm` on permission errors or `error_reply` on other errors.

`FTP.`**`cwd`**(*pathname*)

Set the current directory on the server.

`FTP.`**`mkd`**(*pathname*)

Create a new directory on the server.

`FTP.`**`pwd`**()

Return the pathname of the current directory on the server.

`FTP.`**`rmd`**(*dirname*)

Remove the directory named *dirname* on the server.

`FTP.`**`size`**(*filename*)

Request the size of the file named *filename* on the server. On success, the size of the file is returned as an integer, otherwise `None` is returned. Note that the `SIZE` command is not standardized, but is supported by many common server implementations.

`FTP.`**`quit`**()

Send a `QUIT` command to the server and close the connection. This is the "polite" way to close a connection, but it may raise an exception if the server responds with an error to the `QUIT` command. This implies a call to the `close()` method which renders the `FTP` instance useless for subsequent calls (see below).

`FTP.`**`close`**()

Close the connection unilaterally. This should not be applied to an already closed connection such as after a successful call to `quit()`. After this call the

`FTP` instance should not be used any more (after a call to `close()` or `quit()` you cannot reopen the connection by issuing another `login()` method).

## 21.13.2. FTP_TLS Objects

`FTP_TLS` class inherits from `FTP`, defining these additional objects:

`FTP_TLS.`**`ssl_version`**

The SSL version to use (defaults to `ssl.PROTOCOL_SSLv23`).

`FTP_TLS.`**`auth`**()

Set up a secure control connection by using TLS or SSL, depending on what is specified in the `ssl_version` attribute.

*Changed in version 3.4:* The method now supports hostname check with `ssl.SSLContext.check_hostname` and *Server Name Indication* (see `ssl.HAS_SNI`).

`FTP_TLS.`**`ccc`**()

Revert control channel back to plaintext. This can be useful to take advantage of firewalls that know how to handle NAT with non-secure FTP without opening fixed ports.

*New in version 3.3.*

`FTP_TLS.`**`prot_p`**()

Set up secure data connection.

`FTP_TLS.`**`prot_c`**()

Set up clear text data connection.