21.16. `nntplib` — NNTP protocol client

Source code: Lib/nntplib.py

This module defines the class `NNTP` which implements the client side of the Network News Transfer Protocol. It can be used to implement a news reader or poster, or automated news processors. It is compatible with **RFC 3977** as well as the older **RFC 977** and **RFC 2980**.

Here are two small examples of how it can be used. To list some statistics about a newsgroup and print the subjects of the last 10 articles:

```
>>> s = nntplib.NNTP('news.gmane.org')
>>> resp, count, first, last, name = s.group('gmane.comp.python.commit
>>> print('Group', name, 'has', count, 'articles, range', first, 'to',
Group gmane.comp.python.committers has 1096 articles, range 1 to 1096
>>> resp, overviews = s.over((last - 9, last))
>>> for id, over in overviews:
...     print(id, nntplib.decode_header(over['subject']))
...
1087 Re: Commit privileges for Łukasz Langa
1088 Re: 3.2 alpha 2 freeze
1089 Re: 3.2 alpha 2 freeze
1090 Re: Commit privileges for Łukasz Langa
1091 Re: Commit privileges for Łukasz Langa
1092 Updated ssh key
1093 Re: Updated ssh key
1094 Re: Updated ssh key
1095 Hello fellow committers!
1096 Re: Hello fellow committers!
>>> s.quit()
'205 Bye!'
```

To post an article from a binary file (this assumes that the article has valid headers, and that you have right to post on the particular newsgroup):

```
>>> s = nntplib.NNTP('news.gmane.org')
>>> f = open('article.txt', 'rb')
>>> s.post(f)
'240 Article posted successfully.'
>>> s.quit()
'205 Bye!'
```

The module itself defines the following classes:

class `nntplib.`**`NNTP`**(*host*, *port=119*, *user=None*, *password=None*, *readermode=None*, *usenetrc=False*[, *timeout*])

Return a new `NNTP` object, representing a connection to the NNTP server running on host *host*, listening at port *port*. An optional *timeout* can be specified for the socket connection. If the optional *user* and *password* are provided, or if suitable credentials are present in `/.netrc` and the optional flag *usenetrc* is true, the `AUTHINFO USER` and `AUTHINFO PASS` commands are used to identify and authenticate the user to the server. If the optional flag *readermode* is true, then a `mode reader` command is sent before authentication is performed. Reader mode is sometimes necessary if you are connecting to an NNTP server on the local machine and intend to call reader-specific commands, such as `group`. If you get unexpected `NNTPPermanentError`s, you might need to set *readermode*. The `NNTP` class supports the `with` statement to unconditionally consume `OSError` exceptions and to close the NNTP connection when done, e.g.:

```
>>> from nntplib import NNTP
>>> with NNTP('news.gmane.org') as n:
...     n.group('gmane.comp.python.committers')
...
('211 1755 1 1755 gmane.comp.python.committers', 1755, 1, 1755, 'g
>>>
```

Changed in version 3.2: usenetrc is now `False` by default.

Changed in version 3.3: Support for the `with` statement was added.

class `nntplib.`**`NNTP_SSL`**(*host*, *port=563*, *user=None*, *password=None*, *ssl_context=None*, *readermode=None*, *usenetrc=False*[, *timeout*])

Return a new `NNTP_SSL` object, representing an encrypted connection to the NNTP server running on host *host*, listening at port *port*. `NNTP_SSL` objects have the same methods as `NNTP` objects. If *port* is omitted, port 563 (NNTPS) is used. *ssl_context* is also optional, and is a `SSLContext` object. Please read Security considerations for best practices. All other parameters behave the same as for `NNTP`.

Note that SSL-on-563 is discouraged per **RFC 4642**, in favor of STARTTLS as described below. However, some servers only support the former.

New in version 3.2.

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

exception `nntplib.`**`NNTPError`**

Derived from the standard exception `Exception`, this is the base class for all exceptions raised by the `nntplib` module. Instances of this class have the following attribute:

`response`

The response of the server if available, as a `str` object.

exception `nntplib.`**`NNTPReplyError`**

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

exception `nntplib.`**`NNTPTemporaryError`**

Exception raised when a response code in the range 400–499 is received.

exception `nntplib.`**`NNTPPermanentError`**

Exception raised when a response code in the range 500–599 is received.

exception `nntplib.`**`NNTPProtocolError`**

Exception raised when a reply is received from the server that does not begin with a digit in the range 1–5.

exception `nntplib.`**`NNTPDataError`**

Exception raised when there is some error in the response data.

21.16.1. NNTP Objects

When connected, `NNTP` and `NNTP_SSL` objects support the following methods and attributes.

21.16.1.1. Attributes

`NNTP.`**`nntp_version`**

An integer representing the version of the NNTP protocol supported by the server. In practice, this should be `2` for servers advertising **RFC 3977** compliance and `1` for others.

New in version 3.2.

`NNTP.`**`nntp_implementation`**

A string describing the software name and version of the NNTP server, or `None` if not advertised by the server.

New in version 3.2.

21.16.1.2. Methods

The *response* that is returned as the first item in the return tuple of almost all methods is the server's response: a string beginning with a three-digit code. If the server's response indicates an error, the method raises one of the above exceptions.

Many of the following methods take an optional keyword-only argument *file*. When the *file* argument is supplied, it must be either a file object opened for binary writing, or the name of an on-disk file to be written to. The method will then write any data returned by the server (except for the response line and the terminating dot) to the file; any list of lines, tuples or objects that the method normally returns will be empty.

Changed in version 3.2: Many of the following methods have been reworked and fixed, which makes them incompatible with their 3.1 counterparts.

`NNTP.`**`quit`**`()`

Send a `QUIT` command and close the connection. Once this method has been called, no other methods of the NNTP object should be called.

`NNTP.`**`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.)

`NNTP.`**`getcapabilities`**`()`

Return the **RFC 3977** capabilities advertised by the server, as a `dict` instance mapping capability names to (possibly empty) lists of values. On legacy servers which don't understand the `CAPABILITIES` command, an empty dictionary is returned instead.

```
>>> s = NNTP('news.gmane.org')
>>> 'POST' in s.getcapabilities()
True
```

New in version 3.2.

`NNTP.`**`login`**`(`*user=None*, *password=None*, *usenetrc=True*`)`

Send `AUTHINFO` commands with the user name and password. If *user* and *password* are `None` and *usenetrc* is true, credentials from `~/.netrc` will be used if possible.

Unless intentionally delayed, login is normally performed during the `NNTP` object initialization and separately calling this function is unnecessary. To force authentication to be delayed, you must not set *user* or *password* when creating the object, and must set *usenetrc* to False.

New in version 3.2.

NNTP.**starttls**(*ssl_context=None*)

Send a `STARTTLS` command. This will enable encryption on the NNTP connection. The *ssl_context* argument is optional and should be a `ssl.SSLContext` object. Please read Security considerations for best practices.

Note that this may not be done after authentication information has been transmitted, and authentication occurs by default if possible during a `NNTP` object initialization. See `NNTP.login()` for information on suppressing this behavior.

New in version 3.2.

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

NNTP.**newgroups**(*date*, ***, *file=None*)

Send a `NEWGROUPS` command. The *date* argument should be a `datetime.date` or `datetime.datetime` object. Return a pair `(response, groups)` where *groups* is a list representing the groups that are new since the given *date*. If *file* is supplied, though, then *groups* will be empty.

```
>>> from datetime import date, timedelta
>>> resp, groups = s.newgroups(date.today() - timedelta(days=3))
>>> len(groups)
85
>>> groups[0]
GroupInfo(group='gmane.network.tor.devel', last='4', first='1', fl
```

NNTP.**newnews**(*group*, *date*, ***, *file=None*)

Send a `NEWNEWS` command. Here, *group* is a group name or `'*'`, and *date* has the same meaning as for `newgroups()`. Return a pair `(response, articles)` where *articles* is a list of message ids.

This command is frequently disabled by NNTP server administrators.

NNTP.**list**(*group_pattern=None*, ***, *file=None*)

Send a `LIST` or `LIST ACTIVE` command. Return a pair `(response, list)` where *list* is a list of tuples representing all the groups available from this NNTP server, optionally matching the pattern string *group_pattern*. Each tuple has the form `(group, last, first, flag)`, where *group* is a group name, *last* and *first* are the last and first article numbers, and *flag* usually takes one of these values:

- `y`: Local postings and articles from peers are allowed.
- `m`: The group is moderated and all postings must be approved.
- `n`: No local postings are allowed, only articles from peers.
- `j`: Articles from peers are filed in the junk group instead.
- `x`: No local postings, and articles from peers are ignored.
- `=foo.bar`: Articles are filed in the `foo.bar` group instead.

If *flag* has another value, then the status of the newsgroup should be considered unknown.

This command can return very large results, especially if *group_pattern* is not specified. It is best to cache the results offline unless you really need to refresh them.

Changed in version 3.2: group_pattern was added.

NNTP.**descriptions**(*grouppattern*)

Send a `LIST NEWSGROUPS` command, where *grouppattern* is a wildmat string as specified in **RFC 3977** (it's essentially the same as DOS or UNIX shell wildcard strings). Return a pair `(response, descriptions)`, where *descriptions* is a dictionary mapping group names to textual descriptions.

```
>>> resp, descs = s.descriptions('gmane.comp.python.*')
>>> len(descs)
295
>>> descs.popitem()
('gmane.comp.python.bio.general', 'BioPython discussion list (Mode
```

NNTP.**description**(*group*)

Get a description for a single group *group*. If more than one group matches (if 'group' is a real wildmat string), return the first match. If no group matches, return an empty string.

This elides the response code from the server. If the response code is needed, use `descriptions()`.

NNTP.**group**(*name*)

Send a `GROUP` command, where *name* is the group name. The group is selected as the current group, if it exists. Return a tuple `(response, count, first, last, name)` where *count* is the (estimated) number of articles in the group, *first* is the first article number in the group, *last* is the last article number in the group, and *name* is the group name.

NNTP.**over**(*message_spec*, *, *file=None*)

Send an OVER command, or an XOVER command on legacy servers. *message_spec* can be either a string representing a message id, or a (first, last) tuple of numbers indicating a range of articles in the current group, or a (first, None) tuple indicating a range of articles starting from *first* to the last article in the current group, or None to select the current article in the current group.

Return a pair (response, overviews). *overviews* is a list of (article_number, overview) tuples, one for each article selected by *message_spec*. Each *overview* is a dictionary with the same number of items, but this number depends on the server. These items are either message headers (the key is then the lower-cased header name) or metadata items (the key is then the metadata name prepended with ":"). The following items are guaranteed to be present by the NNTP specification:

- the subject, from, date, message-id and references headers
- the :bytes metadata: the number of bytes in the entire raw article (including headers and body)
- the :lines metadata: the number of lines in the article body

The value of each item is either a string, or None if not present.

It is advisable to use the decode_header() function on header values when they may contain non-ASCII characters:

```
>>> _, _, first, last, _ = s.group('gmane.comp.python.devel')
>>> resp, overviews = s.over((last, last))
>>> art_num, over = overviews[0]
>>> art_num
117216
>>> list(over.keys())
['xref', 'from', ':lines', ':bytes', 'references', 'date', 'messag
>>> over['from']
'=?UTF-8?B?Ik1hcnRpbiB2LiBMw7Z3aXMi?= <martin@v.loewis.de>'
>>> nntplib.decode_header(over['from'])
'"Martin v. Löwis" <martin@v.loewis.de>'
```

New in version 3.2.

NNTP.**help**(*, *file=None*)

Send a HELP command. Return a pair (response, list) where *list* is a list of help strings.

NNTP.**stat**(*message_spec=None*)

Send a `STAT` command, where *message_spec* is either a message id (enclosed in `'<'` and `'>'`) or an article number in the current group. If *message_spec* is omitted or `None`, the current article in the current group is considered. Return a triple `(response, number, id)` where *number* is the article number and *id* is the message id.

```
>>> _, _, first, last, _ = s.group('gmane.comp.python.devel')
>>> resp, number, message_id = s.stat(first)
>>> number, message_id
(9099, '<20030112190404.GE29873@epoch.metaslash.com>')
```

`NNTP.`**`next`**()

Send a `NEXT` command. Return as for `stat()`.

`NNTP.`**`last`**()

Send a `LAST` command. Return as for `stat()`.

`NNTP.`**`article`**(*message_spec=None*, ***, *file=None*)

Send an `ARTICLE` command, where *message_spec* has the same meaning as for `stat()`. Return a tuple `(response, info)` where *info* is a `namedtuple` with three attributes *number*, *message_id* and *lines* (in that order). *number* is the article number in the group (or 0 if the information is not available), *message_id* the message id as a string, and *lines* a list of lines (without terminating newlines) comprising the raw message including headers and body.

```
>>> resp, info = s.article('<20030112190404.GE29873@epoch.metaslas
>>> info.number
0
>>> info.message_id
'<20030112190404.GE29873@epoch.metaslash.com>'
>>> len(info.lines)
65
>>> info.lines[0]
b'Path: main.gmane.org!not-for-mail'
>>> info.lines[1]
b'From: Neal Norwitz <neal@metaslash.com>'
>>> info.lines[-3:]
[b'There is a patch for 2.3 as well as 2.2.', b'', b'Neal']
```

`NNTP.`**`head`**(*message_spec=None*, ***, *file=None*)

Same as `article()`, but sends a `HEAD` command. The *lines* returned (or written to *file*) will only contain the message headers, not the body.

`NNTP.`**`body`**(*message_spec=None*, ***, *file=None*)

Same as `article()`, but sends a `BODY` command. The *lines* returned (or written to *file*) will only contain the message body, not the headers.

`NNTP.`**`post`**(*data*)

Post an article using the `POST` command. The *data* argument is either a file object opened for binary reading, or any iterable of bytes objects (representing raw lines of the article to be posted). It should represent a well-formed news article, including the required headers. The `post()` method automatically escapes lines beginning with `.` and appends the termination line.

If the method succeeds, the server's response is returned. If the server refuses posting, a `NNTPReplyError` is raised.

`NNTP.`**`ihave`**(*message_id*, *data*)

Send an `IHAVE` command. *message_id* is the id of the message to send to the server (enclosed in `'<'` and `'>'`). The *data* parameter and the return value are the same as for `post()`.

`NNTP.`**`date`**()

Return a pair `(response, date)`. *date* is a `datetime` object containing the current date and time of the server.

`NNTP.`**`slave`**()

Send a `SLAVE` command. Return the server's *response*.

`NNTP.`**`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 or response. A value of `2` or higher produces the maximum amount of debugging output, logging each line sent and received on the connection (including message text).

The following are optional NNTP extensions defined in **RFC 2980**. Some of them have been superseded by newer commands in **RFC 3977**.

`NNTP.`**`xhdr`**(*hdr*, *str*, ***, *file=None*)

Send an `XHDR` command. The *hdr* argument is a header keyword, e.g. `'subject'`. The *str* argument should have the form `'first-last'` where *first* and *last* are the first and last article numbers to search. Return a pair `(response, list)`, where *list* is a list of pairs `(id, text)`, where *id* is an article number (as a string) and *text* is the text of the requested header for that article. If the *file* parameter is supplied, then the output of the `XHDR` command is stored in a file. If *file* is a string, then the method will open a file with that name, write to it then close it. If *file* is a file object, then it will start calling `write()` on it to store the lines of the command output. If *file* is supplied, then the returned *list* is an empty list.

`NNTP.`**`xover`**(*start*, *end*, *, *file=None*)

Send an `XOVER` command. *start* and *end* are article numbers delimiting the range of articles to select. The return value is the same of for `over()`. It is recommended to use `over()` instead, since it will automatically use the newer `OVER` command if available.

`NNTP.`**`xpath`**(*id*)

Return a pair `(resp, path)`, where *path* is the directory path to the article with message ID *id*. Most of the time, this extension is not enabled by NNTP server administrators.

Deprecated since version 3.3: The XPATH extension is not actively used.

21.16.2. Utility functions

The module also defines the following utility function:

`nntplib.`**`decode_header`**(*header_str*)

Decode a header value, un-escaping any escaped non-ASCII characters. *header_str* must be a `str` object. The unescaped value is returned. Using this function is recommended to display some headers in a human readable form:

```
>>> decode_header("Some subject")
'Some subject'
>>> decode_header("=?ISO-8859-15?Q?D=E9buter_en_Python?=")
'Débuter en Python'
>>> decode_header("Re: =?UTF-8?B?cHJvYmzDqG1lIGRlIG1hdHJpY2U=?=")
'Re: problème de matrice'
```