`!imaplib` --- IMAP4 protocol client

- IMAP4 Objects
- IMAP4 Example

imaplib

Source code: `Lib/imaplib.py`

pair: IMAP4; protocol pair: IMAP4_SSL; protocol pair: IMAP4_stream; protocol

This module defines three classes, `IMAP4`, `IMAP4_SSL` and `IMAP4_stream`, which encapsulate a connection to an IMAP4 server and implement a large subset of the IMAP4rev1 client protocol as defined in `3501`. It is backward compatible with IMAP4 (`1730`) servers, but note that the `STATUS` command is not supported in IMAP4.

not WASI.

This module does not work or is not available on WebAssembly. See `wasm-availability` for more information.

Three classes are provided by the `!imaplib` module, `IMAP4` is the base class:

This class implements the actual IMAP4 protocol. The connection is created and protocol version (IMAP4 or IMAP4rev1) is determined when the instance is initialized. If *host* is not specified, `''` (the local host) is used. If *port* is omitted, the standard IMAP4 port (143) is used. The optional *timeout* parameter specifies a timeout in seconds for the connection attempt. If timeout is not given or is `None`, the global default socket timeout is used.

The `IMAP4` class supports the `with` statement. When used like this, the IMAP4 `LOGOUT` command is issued automatically when the `!with` statement exits. E.g.:

```
>>> from imaplib import IMAP4
>>> with IMAP4("domain.org") as M:
```

```
...     M.noop()
...
('OK', [b'Nothing Accomplished. d25if65hy903weo.87'])
```

3.5 Support for the `with` statement was added.

3.9 The optional *timeout* parameter was added.

Three exceptions are defined as attributes of the `IMAP4` class:

IMAP4.error

Exception raised on any errors. The reason for the exception is passed to the constructor as a string.

IMAP4.abort

IMAP4 server errors cause this exception to be raised. This is a sub-class of `IMAP4.error`. Note that closing the instance and instantiating a new one will usually allow recovery from this exception.

IMAP4.readonly

This exception is raised when a writable mailbox has its status changed by the server. This is a sub-class of `IMAP4.error`. Some other client now has write permission, and the mailbox will need to be re-opened to re-obtain write permission.

There's also a subclass for secure connections:

This is a subclass derived from `IMAP4` that connects over an SSL encrypted socket (to use this class you need a socket module that was compiled with SSL support). If *host* is not specified, `''` (the local host) is used. If *port* is omitted, the standard IMAP4-over-SSL port (993) is used. *ssl_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 `ssl-security` for best practices.

> [!NOTE] With the default *ssl_context*, the connection is encrypted but the server certificate and hostname are not verified. To verify them, pass a context created by `ssl.create_default_context`.

The optional *timeout* parameter specifies a timeout in seconds for the connection attempt. If timeout is not given or is `None`, the global default socket timeout is used.

3.3 *ssl_context* parameter was added.

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

3.9 The optional *timeout* parameter was added.

3.12 The deprecated *keyfile* and *certfile* parameters have been removed.

The second subclass allows for connections created by a child process:

This is a subclass derived from `IMAP4` that connects to the `stdin/stdout` file descriptors created by passing *command* to `subprocess.Popen()`.

The following utility functions are defined:

Internaldate2tuple(resp)

Parse a `bytes-like object` containing an IMAP4 `INTERNALDATE` response and return the corresponding local time. The return value is a `time.struct_time` tuple or `None` if the input has wrong format.

Int2AP(num)

Converts an integer into a bytes representation using characters from the set [`A` .. `P`].

ParseFlags(resp)

Converts a `bytes-like object` containing an IMAP4 `FLAGS` response to a tuple of individual flags as `bytes`. The return value is an empty tuple if the input has wrong format.

Time2Internaldate(date_time)

Convert *date_time* to an IMAP4 `INTERNALDATE` representation. The return value is a string in the form: `"DD-Mmm-YYYY HH:MM:SS +HHMM"` (including double-quotes). The *date_time* argument can be a number (int or float) representing seconds since epoch (as returned by `time.time`), a 9-tuple representing local time an instance of `time.struct_time` (as returned by `time.localtime`), an aware instance of `datetime.datetime`, or a double-quoted string. In the last case, it is assumed to already be in the correct format.

Note that IMAP4 message numbers change as the mailbox changes; in particular, after an `EXPUNGE` command performs deletions the remaining messages are renumbered. So it is highly advisable to use UIDs instead, with the UID command.

At the end of the module, there is a test section that contains a more extensive example of usage.

Documents describing the protocol, sources for servers implementing it, by the University of Washington's IMAP Information Center can all be found at (**Source Code**) https://github.com/uw-imap/imap (**Not Maintained**).

IMAP4 Objects

All IMAP4rev1 commands are represented by methods of the same name, either uppercase or lowercase.

All arguments to commands are converted to strings, except for `AUTHENTICATE`, and the last argument to `APPEND` which is passed as an IMAP4 literal. If necessary (the string contains IMAP4 protocol-sensitive characters and isn't enclosed with either parentheses or double quotes) each string is quoted. However, the *password* argument to the `LOGIN` command is always quoted. If you want to avoid having an argument string quoted (eg: the *flags* argument to `STORE`) then enclose the string in parentheses (eg: `r'(\Deleted)'`). Or you can quote the string yourself; an argument that is already enclosed in double quotes is left unchanged. In general, however, it is better to pass arguments unquoted and let the module quote them as needed.

Mailbox names are encoded as modified UTF-7 (`3501`, section 5.1.3), so a mailbox name containing non-ASCII characters can be passed as an ordinary `str`. A `str` that is already valid modified UTF-7 is left unchanged, so that a name obtained from `~IMAP4.list` (raw `bytes` decoded to text) round-trips; pass `bytes` to send the exact bytes with no encoding. When `UTF8=ACCEPT` is enabled (see `~IMAP4.enable`), mailbox names are sent as UTF-8 instead.

3.16 Non-ASCII mailbox names are automatically encoded as modified UTF-7.

Most commands return a tuple: `(type, [data, ...])` where *type* is usually `'OK'` or `'NO'`, and *data* is either the text from the command response, or mandated results from the command. Each *data* is either a `bytes`, or a tuple. If a tuple, then the first part is the header of the response, and the second part contains the data (ie: 'literal' value).

The *message_set* options to the commands below can be a string specifying one or more messages to be acted upon. It may be a simple message number (`'1'`), a range of message numbers (`'2:4'`), or a group of non-contiguous ranges separated by commas (`'1:3,6:9'`). A range can contain an asterisk to indicate an infinite upper bound (`'3:*'`).

Alternatively it can be specified using integers and `range` objects. It may be a single message number or a sequence. The sequence items may be integers, `(start, stop)` tuples (where `None` or `'*'` stands for the last message), or `range` objects. For example, `[1, (3, 5), 8]` and `[range(1, 6), 8]` are both equivalent to `'1,3:5,8'`.

3.16 Added support for the structured *message_set*.

Command arguments that are parenthesized lists of atoms --- such as the *flag_list* argument of `~IMAP4.store` and the *flags* argument of `~IMAP4.append`, the *names* argument of `~IMAP4.status`, the *sort_criteria* argument of `~IMAP4.sort`, or the *message_parts* argument of `~IMAP4.fetch` --- can be passed as a sequence of strings instead of a single preformatted string. For example, `[r'\Seen', r'\Answered']` is equivalent to `(\Seen \Answered)`.

3.16 Added support for passing these arguments as a sequence.

The value-bearing arguments of the search and fetch commands can be quoted by hand, but this is error prone. Instead, they may contain `?` placeholders that are substituted, and quoted as required, from a *params* keyword argument, in the manner of `sqlite3` parameter substitution:

```
# SEARCH FROM me@example.com SUBJECT "trip report"
M.search(None, 'FROM ? SUBJECT ?', params=['me@example.com', 'trip report'])


# FETCH 1:5 (FLAGS BODY[HEADER.FIELDS (DATE FROM)])
M.fetch('1:5', 'FLAGS BODY[HEADER.FIELDS ?]', params=[['DATE', 'FROM']])
```

The placeholders are:

- `?` --- an `astring`: a string (which will be quoted if necessary), an integer, or a list of integers and/or strings (which will be sent as a parenthesized list);
- `?f` --- a flag or a list of flags, sent verbatim without quoting;
- `?s` --- a *message_set* in the structured form described above.

`??` stands for a literal `?`.

Substitution is only performed when *params* is given; if no *params* are given, an argument containing a literal `?` is unchanged. The *params* keyword is accepted by `~IMAP4.search`, `~IMAP4.fetch`, `~IMAP4.sort`, `~IMAP4.thread` and `~IMAP4.uid`.

3.16 The *params* keyword argument.

An `IMAP4` instance has the following methods:

IMAP4.append(mailbox, flags, date_time, message, *, translate_line_endings=True)

Append *message* to named mailbox.

flags may be `None` or a string of IMAP flag tokens. Multiple flags are separated by spaces, for example `r'\Seen \Answered'`. If *flags* is not already enclosed in parentheses, parentheses are added automatically.

If *translate_line_endings* is true (the default), line endings in *message* are translated to CRLF. Pass `False` to send the message literal exactly as given, which is required to preserve messages that

contain bare CR or LF. In that case *message* must already use CRLF line endings as required by `3501`; for example, serialize `email` messages using `email.policy.SMTP`.

3.16 Added the *translate_line_endings* parameter.

IMAP4.authenticate(mechanism, authobject)

Authenticate command --- requires response processing.

mechanism specifies which authentication mechanism is to be used - it should appear in the instance variable `capabilities` in the form `AUTH=mechanism`.

authobject must be a callable object:

```
data = authobject(response)
```

It will be called to process server continuation responses; the *response* argument it is passed will be `bytes`. It should return `bytes` *data* that will be base64 encoded and sent to the server. It should return `None` if the client abort response `*` should be sent instead.

3.5 string usernames and passwords are now encoded to `utf-8` instead of being limited to ASCII.

IMAP4.check()

Checkpoint mailbox on server.

IMAP4.close()

Close currently selected mailbox. Deleted messages are removed from writable mailbox. This is the recommended command before `LOGOUT`.

IMAP4.copy(message_set, new_mailbox, *, uid=False)

Copy *message_set* messages onto end of *new_mailbox*.

If *uid* is true, *message_set* is a set of UIDs and the `UID COPY` command is used instead of `COPY`.

3.16 Added the *uid* parameter.

IMAP4.create(mailbox)

Create new mailbox named *mailbox*.

IMAP4.delete(mailbox)

Delete old mailbox named *mailbox*.

IMAP4.deleteacl(mailbox, who)

Delete the ACLs (remove any rights) set for who on mailbox.

IMAP4.enable(capability)

Enable *capability* (see `5161`). Most capabilities do not need to be enabled. Currently only the `UTF8=ACCEPT` capability is supported (see RFC 6855).

3.5 The `enable` method itself, and RFC 6855 support.

IMAP4.expunge(message_set=None, *, uid=False)

Permanently remove deleted items from selected mailbox. Generates an `EXPUNGE` response for each deleted message. Returned data contains a list of `EXPUNGE` message numbers in order received.

If *uid* is true, the `UID EXPUNGE` command (`4315`) is used to remove only the messages that both are marked as deleted and have a UID in *message_set*. *message_set* is required in this case, and must be omitted otherwise.

3.16 Added the *message_set* and *uid* parameters.

IMAP4.fetch(message_set, message_parts, *, uid=False, params=None)

Fetch (parts of) messages. *message_parts* should be a string of message part names enclosed within parentheses, eg: `"(UID BODY[TEXT])"`. Returned data are tuples of message part envelope and data.

If *uid* is true, *message_set* is a set of UIDs and the message numbers in the response are UIDs (`UID FETCH`).

If *params* is given, `?` placeholders in *message_parts* are substituted with the quoted parameters (see `the placeholders <imap4-params>`).

3.16 Added the *params* and *uid* parameters.

IMAP4.getacl(mailbox)

Get the `ACL`s for *mailbox*. The method is non-standard, but is supported by the `Cyrus` server.

IMAP4.getannotation(mailbox, entry, attribute)

Retrieve the specified `ANNOTATION`s for *mailbox*. The method is non-standard, but is supported by the `Cyrus` server.

IMAP4.getquota(root)

Get the `quota` *root*'s resource usage and limits. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.getquotaroot(mailbox)

Get the list of `quota` `roots` for the named *mailbox*. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.id(fields=None)

Send client identification information to the server and return the identification information sent back by the server (the `ID` command, defined in `2971`). *fields* is a mapping of field names to values (for example, `{'name': 'myclient', 'version': '1.0'}`); a value can be `None`. The server must support the `ID` capability.

3.16

IMAP4.idle(duration=None)

Return an `!Idler`: an iterable context manager implementing the IMAP4 `IDLE` command as defined in `2177`.

The returned object sends the `IDLE` command when activated by the `with` statement, produces IMAP untagged responses via the `iterator` protocol, and sends `DONE` upon context exit.

All untagged responses that arrive after sending the `IDLE` command (including any that arrive before the server acknowledges the command) will be available via iteration. Any leftover responses (those not iterated in the `with` context) can be retrieved in the usual way after `IDLE` ends, using `IMAP4.response`.

Responses are represented as `(type, [data, ...])` tuples, as described in `IMAP4 Objects <imap4-objects>`.

The *duration* argument sets a maximum duration (in seconds) to keep idling, after which any ongoing iteration will stop. It can be an `int` or `float`, or `None` for no time limit. Callers wishing to avoid inactivity timeouts on servers that impose them should keep this at most 29 minutes (1740 seconds). Requires a socket connection; *duration* must be `None` on `IMAP4_stream` connections.

```
>>> with M.idle(duration=29 * 60) as idler:
...     for typ, data in idler:
...         print(typ, data)
...
EXISTS [b'1']
```

```
RECENT [b'1']
```

Idler.burst(interval=0.1)

Yield a burst of responses no more than *interval* seconds apart (expressed as an `int` or `float`).

This `generator` is an alternative to iterating one response at a time, intended to aid in efficient batch processing. It retrieves the next response along with any immediately available subsequent responses. (For example, a rapid series of `EXPUNGE` responses after a bulk delete.)

Requires a socket connection; does not work on `IMAP4_stream` connections.

```
>>> with M.idle() as idler:
...     # get a response and any others following by < 0.1 seconds
...     batch = list(idler.burst())
...     print(f'processing {len(batch)} responses...')
...     print(batch)
...
processing 3 responses...
[('EXPUNGE', [b'2']), ('EXPUNGE', [b'1']), ('RECENT', [b'0'])]
```

> [!TIP] The `IDLE` context's maximum duration, as passed to `IMAP4.idle`, is respected when waiting for the first response in a burst. Therefore, an expired `!Idler` will cause this generator to return immediately without producing anything. Callers should consider this if using it in a loop.

> [!NOTE] The iterator returned by `IMAP4.idle` is usable only within a `with` statement. Before or after that context, unsolicited responses are collected internally whenever a command finishes, and can be retrieved with `IMAP4.response`.

> [!NOTE] The `!Idler` class name and structure are internal interfaces, subject to change. Calling code can rely on its context management, iteration, and public method to remain stable, but should not subclass, instantiate, compare, or otherwise directly reference the class.

3.14

IMAP4.list(directory='', pattern='*')

List mailbox names in *directory* matching *pattern*. *directory* defaults to the top-level mail folder, and *pattern* defaults to match anything. Returned data contains a list of `LIST` responses.

IMAP4.login(user, password)

Identify the client using a plaintext password. The *password* will be quoted.

IMAP4.login_cram_md5(user, password)

Force use of `CRAM-MD5` authentication when identifying the client to protect the password. It will only work if the server `CAPABILITY` response includes the phrase `AUTH=CRAM-MD5`.

3.15 An `IMAP4.error` is raised if MD5 support is not available.

IMAP4.login_plain(user, password)

Authenticate using the `PLAIN` SASL mechanism (`4616`).

This is a plaintext authentication mechanism that can be used instead of `login` when UTF-8 support is required (see `6855`). Since the credentials are only base64-encoded, not encrypted, this method should only be used over a TLS-protected connection, such as `IMAP4_SSL` or after `starttls`.

It will only work if the server supports the `PLAIN` mechanism, which it need not advertise as `AUTH=PLAIN` in its `CAPABILITY` response.

3.16

IMAP4.logout()

Shutdown connection to server. Returns server `BYE` response.

3.8 The method no longer ignores silently arbitrary exceptions.

IMAP4.lsub(directory='', pattern='*')

List subscribed mailbox names in directory matching pattern. *directory* defaults to the top level directory and *pattern* defaults to match any mailbox. Returned data are tuples of message part envelope and data.

IMAP4.move(message_set, new_mailbox, *, uid=False)

Move *message_set* messages onto end of *new_mailbox*.

The server must support the `MOVE` capability (`6851`).

If *uid* is true, *message_set* is a set of UIDs and the `UID MOVE` command is used instead of `MOVE`.

3.16

IMAP4.myrights(mailbox)

Show my ACLs for a mailbox (i.e. the rights that I have on mailbox).

IMAP4.namespace()

Returns IMAP namespaces as defined in `2342`.

IMAP4.noop()

Send `NOOP` to server.

IMAP4.open(host, port, timeout=None)

Opens socket to *port* at *host*. The optional *timeout* parameter specifies a timeout in seconds for the connection attempt. If timeout is not given or is `None`, the global default socket timeout is used. Also note that if the *timeout* parameter is set to be zero, it will raise a `ValueError` to reject creating a non-blocking socket. This method is implicitly called by the `IMAP4` constructor. The connection objects established by this method will be used in the `IMAP4.read`, `IMAP4.readline`, `IMAP4.send`, and `IMAP4.shutdown` methods. You may override this method.

imaplib.open self,host,port imaplib.IMAP4.open

3.9 The *timeout* parameter was added.

IMAP4.partial(message_num, message_part, start, length)

Fetch truncated part of a message. Returned data is a tuple of message part envelope and data.

IMAP4.proxyauth(user)

Assume authentication as *user*. Allows an authorised administrator to proxy into any user's mailbox.

IMAP4.read(size)

Reads *size* bytes from the remote server. You may override this method.

IMAP4.readline()

Reads one line from the remote server. You may override this method.

IMAP4.recent()

Prompt server for an update. Returned data is `None` if no new messages, else value of `RECENT` response.

IMAP4.rename(oldmailbox, newmailbox)

Rename mailbox named *oldmailbox* to *newmailbox*.

IMAP4.response(code)

Return data for response *code* if received, or `None`. Returns the given code, instead of the usual type.

IMAP4.search(charset, criterion[, ...], *, uid=False, params=None)

Search mailbox for matching messages. *charset* may be `None`, in which case no `CHARSET` will be specified in the request to the server. The IMAP protocol requires that at least one criterion be specified; an exception will be raised when the server returns an error. *charset* must be `None` if the `UTF8=ACCEPT` capability was enabled using the `enable` command.

If *uid* is true, the message numbers in the response are UIDs (`UID SEARCH`).

A criterion passed as `str` is encoded to *charset* (which must name a codec known to Python); pass `bytes` to send a criterion that is already encoded, for example when *charset* is one that Python does not support. When *charset* is `None` (as it must be under `UTF8=ACCEPT`), the criterion is sent using the connection's encoding instead.

If *params* is given, `?` placeholders in the criteria are substituted with the quoted parameters (see `the placeholders <imap4-params>`).

Example:

```
# M is a connected IMAP4 instance...
typ, msgnums = M.search(None, 'FROM', '"John Smith"')

# or:
typ, msgnums = M.search(None, '(FROM "John Smith")')

# or, letting the module quote the value (this is recommended):
typ, msgnums = M.search(None, 'FROM ?', params=['John Smith'])
```

3.16 Added the *params* and *uid* parameters. `str` search criteria are encoded to *charset*.

IMAP4.select(mailbox='INBOX', readonly=False)

Select a mailbox. Returned data is the count of messages in *mailbox* (`EXISTS` response). The default *mailbox* is `'INBOX'`. If the *readonly* flag is set, modifications to the mailbox are not allowed.

IMAP4.send(data)

Sends `data` to the remote server. You may override this method.

imaplib.send self,data imaplib.IMAP4.send

IMAP4.setacl(mailbox, who, what)

Set an `ACL` for *mailbox*. The method is non-standard, but is supported by the `Cyrus` server.

IMAP4.setannotation(mailbox, entry, attribute[, ...])

Set `ANNOTATION`s for *mailbox*. The method is non-standard, but is supported by the `Cyrus` server.

IMAP4.setquota(root, limits)

Set the `quota` *root*'s resource *limits*. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.shutdown()

Close connection established in `open`. This method is implicitly called by `IMAP4.logout`. You may override this method.

IMAP4.socket()

Returns socket instance used to connect to server.

IMAP4.sort(sort_criteria, charset, search_criterion[, ...], *, uid=False, params=None)

The `sort` command is a variant of `search` with sorting semantics for the results. Returned data contains a space separated list of matching message numbers.

Sort has two arguments before the *search_criterion* argument(s); a parenthesized list of *sort_criteria*, and the searching *charset*. Note that unlike `search`, the searching *charset* argument is mandatory. There is also a `uid sort` command which corresponds to `sort` the way that `uid search` corresponds to `search`. The `sort` command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the numbers of matching messages.

If *uid* is true, the message numbers in the response are UIDs (`UID SORT`).

As with `search`, a *search_criterion* passed as `str` is encoded to *charset*; pass `bytes` to send one already encoded.

If *params* is given, `?` placeholders in the search criteria are substituted with the quoted parameters (see `the placeholders <imap4-params>`).

This is an `IMAP4rev1` extension command.

3.16 Added the *params* and *uid* parameters. `str` search criteria are encoded to *charset*.

IMAP4.starttls(ssl_context=None)

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

> [!NOTE] With the default *ssl_context*, the connection is encrypted but the server certificate and hostname are not verified. To verify them, pass a context created by `ssl.create_default_context`.

3.2

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

IMAP4.status(mailbox, names)

Request named status conditions for *mailbox*.

IMAP4.store(message_set, command, flag_list, *, uid=False)

Alters flag dispositions for messages in mailbox. *command* is specified by section 6.4.6 of `3501` as being one of "FLAGS", "+FLAGS", or "-FLAGS", optionally with a suffix of ".SILENT".

If *uid* is true, *message_set* is a set of UIDs and the `UID STORE` command is used instead of `STORE`.

For example, to set the delete flag on all messages:

```
typ, data = M.search(None, 'ALL')
for num in data[0].split():
   M.store(num, '+FLAGS', r'\Deleted')
M.expunge()
```

> [!NOTE] Creating flags containing ']' (for example: "[test]") violates `3501` (the IMAP protocol). However, imaplib has historically allowed creation of such flags, and popular IMAP servers, such as Gmail, accept and produce such flags. There are non-Python programs which also create such flags. Although it is an RFC violation and IMAP clients and servers are supposed to be strict, imaplib still continues to allow such flags to be created for backward compatibility reasons, and as of Python 3.6, handles them if they are sent from the server, since this improves real-world compatibility.

3.16 Added the *uid* parameter.

IMAP4.subscribe(mailbox)

Subscribe to new mailbox.

IMAP4.thread(threading_algorithm, charset, search_criterion[, ...], *, uid=False, params=None)

The `thread` command is a variant of `search` with threading semantics for the results. Returned data contains a space separated list of thread members.

Thread members consist of zero or more messages numbers, delimited by spaces, indicating successive parent and child.

Thread has two arguments before the *search_criterion* argument(s); a *threading_algorithm*, and the searching *charset*. Note that unlike `search`, the searching *charset* argument is mandatory. There is also a `uid thread` command which corresponds to `thread` the way that `uid search` corresponds to `search`. The `thread` command first searches the mailbox for messages that match the given searching criteria using the *charset* argument for the interpretation of strings in the searching criteria. It then returns the matching messages threaded according to the specified threading algorithm.

If *uid* is true, the message numbers in the response are UIDs (`UID THREAD`).

As with `search`, a *search_criterion* passed as `str` is encoded to *charset*; pass `bytes` to send one already encoded.

If *params* is given, `?` placeholders in the search criteria are substituted with the quoted parameters (see `the placeholders <imap4-params>`).

This is an `IMAP4rev1` extension command.

3.16 Added the *params* and *uid* parameters. `str` search criteria are encoded to *charset*.

IMAP4.uid(command, arg[, ...], *, params=None)

Execute command args with messages identified by UID, rather than message number. Returns response appropriate to command. At least one argument must be supplied; if none are provided, the server will return an error and an exception will be raised.

If *params* is given, `?` placeholders in the `SEARCH`, `SORT` and `THREAD` criteria or in the `FETCH` parts are substituted with the quoted parameters (see `the placeholders <imap4-params>`).

3.16 Added the *params* parameter.

IMAP4.unsubscribe(mailbox)

Unsubscribe from old mailbox.

IMAP4.unselect()

`imaplib.IMAP4.unselect` frees server's resources associated with the selected mailbox and returns the server to the authenticated state. This command performs the same actions as `imaplib.IMAP4.close`, except that no messages are permanently removed from the currently selected mailbox.

3.9

IMAP4.xatom(name[, ...])

Allow simple extension commands notified by server in `CAPABILITY` response.

The following attributes are defined on instances of `IMAP4`:

IMAP4.capabilities

A tuple of the capabilities advertised by the server, in upper case.

It is set when the connection is established, and refreshed after a successful `~IMAP4.login`, `~IMAP4.authenticate` or `~IMAP4.starttls`, because the server can advertise different capabilities in different connection states.

3.14.7 Refreshed after `~IMAP4.login` and `~IMAP4.authenticate`.

IMAP4.PROTOCOL_VERSION

The most recent supported protocol in the `CAPABILITY` response from the server.

IMAP4.debug

Integer value to control debugging output. The initialize value is taken from the module variable `Debug`. Values greater than three trace each command.

IMAP4.utf8_enabled

Boolean value that is normally `False`, but is set to `True` if an `enable` command is successfully issued for the `UTF8=ACCEPT` capability.

3.5

IMAP4.file

Internal `~io.BufferedReader` associated with the underlying socket. This property is documented for legacy purposes but not part of the public interface. The caller is responsible to ensure that the current file is closed before changing it.

3.15 3.19

IMAP4 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```
import getpass, imaplib

M = imaplib.IMAP4(host='example.org')
M.login(getpass.getuser(), getpass.getpass())
M.select()
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    typ, data = M.fetch(num, '(RFC822)')
    print('Message %s\n%s\n' % (num, data[0][1]))
M.close()
M.logout()
```

> [!NOTE] A `FETCH` response may contain additional or unsolicited data (see `3501`, section 7.4.2), so production code should inspect the whole response rather than rely on `data[0][1]`.

□□□ #2
□ □□□ □□□ 2026-10-31 17:12:17 UTC
□ □□□ □□□ 2026-10-31 17:12:19 UTC