CHAPTER

TWENTYTWO

INTERNET PROTOCOLS AND SUPPORT

The modules described in this chapter implement Internet protocols and support for related technology. They are all implemented in Python. Most of these modules require the presence of the system-dependent module *socket*, which is currently supported on most popular platforms. Here is an overview:

22.1 webbrowser — Convenient Web-browser controller

Source code: Lib/webbrowser.py

The *webbrowser* module provides a high-level interface to allow displaying Web-based documents to users. Under most circumstances, simply calling the *open()* function from this module will do the right thing.

Under Unix, graphical browsers are preferred under X11, but text-mode browsers will be used if graphical browsers are not available or an X11 display isn't available. If text-mode browsers are used, the calling process will block until the user exits the browser.

If the environment variable BROWSER exists, it is interpreted as the *os.pathsep*-separated list of browsers to try ahead of the platform defaults. When the value of a list part contains the string %s, then it is interpreted as a literal browser command line to be used with the argument URL substituted for %s; if the part does not contain %s, it is simply interpreted as the name of the browser to launch.[1]

For non-Unix platforms, or when a remote browser is available on Unix, the controlling process will not wait for the user to finish with the browser, but allow the remote browser to maintain its own windows on the display. If remote browsers are not available on Unix, the controlling process will launch a new browser and wait.

The script **webbrowser** can be used as a command-line interface for the module. It accepts a URL as the argument. It accepts the following optional parameters: -n opens the URL in a new browser window, if possible; -t opens the URL in a new browser page ("tab"). The options are, naturally, mutually exclusive. Usage example:

```
python -m webbrowser -t "http://www.python.org"
```

The following exception is defined:

exception webbrowser.Error
Exception raised when a browser control error occurs.

The following functions are defined:

webbrowser.open(*url*, *new=0*, *autoraise=True*)
Display *url* using the default browser. If *new* is 0, the *url* is opened in the same browser window if possible. If *new* is 1, a new browser window is opened if possible. If *new* is 2, a new browser page

[1] Executables named here without a full path will be searched in the directories given in the PATH environment variable.

("tab") is opened if possible. If *autoraise* is `True`, the window is raised if possible (note that under many window managers this will occur regardless of the setting of this variable).

Note that on some platforms, trying to open a filename using this function, may work and start the operating system's associated program. However, this is neither supported nor portable.

`webbrowser.open_new(`*url*`)`
Open *url* in a new window of the default browser, if possible, otherwise, open *url* in the only browser window.

`webbrowser.open_new_tab(`*url*`)`
Open *url* in a new page ("tab") of the default browser, if possible, otherwise equivalent to *open_new()*.

`webbrowser.get(`*using=None*`)`
Return a controller object for the browser type *using*. If *using* is `None`, return a controller for a default browser appropriate to the caller's environment.

`webbrowser.register(`*name*, *constructor*, *instance=None*, ***, *preferred=False*`)`
Register the browser type *name*. Once a browser type is registered, the *get()* function can return a controller for that browser type. If *instance* is not provided, or is `None`, *constructor* will be called without parameters to create an instance when needed. If *instance* is provided, *constructor* will never be called, and may be `None`.

Setting *preferred* to `True` makes this browser a preferred result for a *get()* call with no argument. Otherwise, this entry point is only useful if you plan to either set the `BROWSER` variable or call *get()* with a nonempty argument matching the name of a handler you declare.

Changed in version 3.7: *preferred* keyword-only parameter was added.

A number of browser types are predefined. This table gives the type names that may be passed to the *get()* function and the corresponding instantiations for the controller classes, all defined in this module.

Type Name	Class Name	Notes
`'mozilla'`	`Mozilla('mozilla')`	
`'firefox'`	`Mozilla('mozilla')`	
`'netscape'`	`Mozilla('netscape')`	
`'galeon'`	`Galeon('galeon')`	
`'epiphany'`	`Galeon('epiphany')`	
`'skipstone'`	`BackgroundBrowser('skipstone')`	
`'kfmclient'`	`Konqueror()`	(1)
`'konqueror'`	`Konqueror()`	(1)
`'kfm'`	`Konqueror()`	(1)
`'mosaic'`	`BackgroundBrowser('mosaic')`	
`'opera'`	`Opera()`	
`'grail'`	`Grail()`	
`'links'`	`GenericBrowser('links')`	
`'elinks'`	`Elinks('elinks')`	
`'lynx'`	`GenericBrowser('lynx')`	
`'w3m'`	`GenericBrowser('w3m')`	
`'windows-default'`	`WindowsDefault`	(2)
`'macosx'`	`MacOSX('default')`	(3)
`'safari'`	`MacOSX('safari')`	(3)
`'google-chrome'`	`Chrome('google-chrome')`	
`'chrome'`	`Chrome('chrome')`	
`'chromium'`	`Chromium('chromium')`	
`'chromium-browser'`	`Chromium('chromium-browser')`	

Notes:

(1) "Konqueror" is the file manager for the KDE desktop environment for Unix, and only makes sense to use if KDE is running. Some way of reliably detecting KDE would be nice; the `KDEDIR` variable is not sufficient. Note also that the name "kfm" is used even when using the **`konqueror`** command with KDE 2 — the implementation selects the best strategy for running Konqueror.

(2) Only on Windows platforms.

(3) Only on Mac OS X platform.

New in version 3.3: Support for Chrome/Chromium has been added.

Here are some simple examples:

```
url = 'http://docs.python.org/'

# Open URL in a new tab, if a browser window is already open.
webbrowser.open_new_tab(url)

# Open URL in new window, raising the window if possible.
webbrowser.open_new(url)
```

22.1.1 Browser Controller Objects

Browser controllers provide these methods which parallel three of the module-level convenience functions:

`controller.open`(*url*, *new=0*, *autoraise=True*)
Display *url* using the browser handled by this controller. If *new* is 1, a new browser window is opened if possible. If *new* is 2, a new browser page ("tab") is opened if possible.

`controller.open_new`(*url*)
Open *url* in a new window of the browser handled by this controller, if possible, otherwise, open *url* in the only browser window. Alias *open_new()*.

`controller.open_new_tab`(*url*)
Open *url* in a new page ("tab") of the browser handled by this controller, if possible, otherwise equivalent to *open_new()*.

22.2 `cgi` — Common Gateway Interface support

Source code: Lib/cgi.py

Support module for Common Gateway Interface (CGI) scripts.

This module defines a number of utilities for use by CGI scripts written in Python.

22.2.1 Introduction

A CGI script is invoked by an HTTP server, usually to process user input submitted through an HTML `<FORM>` or `<ISINDEX>` element.

Most often, CGI scripts live in the server's special `cgi-bin` directory. The HTTP server places all sorts of information about the request (such as the client's hostname, the requested URL, the query string, and lots of other goodies) in the script's shell environment, executes the script, and sends the script's output back to the client.

The script's input is connected to the client too, and sometimes the form data is read this way; at other times the form data is passed via the "query string" part of the URL. This module is intended to take care of the different cases and provide a simpler interface to the Python script. It also provides a number of utilities that help in debugging scripts, and the latest addition is support for file uploads from a form (if your browser supports it).

The output of a CGI script should consist of two sections, separated by a blank line. The first section contains a number of headers, telling the client what kind of data is following. Python code to generate a minimal header section looks like this:

```
print("Content-Type: text/html")    # HTML is following
print()                             # blank line, end of headers
```

The second section is usually HTML, which allows the client software to display nicely formatted text with header, in-line images, etc. Here's Python code that prints a simple piece of HTML:

```
print("<TITLE>CGI script output</TITLE>")
print("<H1>This is my first CGI script</H1>")
print("Hello, world!")
```

22.2.2 Using the cgi module

Begin by writing `import cgi`.

When you write a new script, consider adding these lines:

```
import cgitb
cgitb.enable()
```

This activates a special exception handler that will display detailed reports in the Web browser if any errors occur. If you'd rather not show the guts of your program to users of your script, you can have the reports saved to files instead, with code like this:

```
import cgitb
cgitb.enable(display=0, logdir="/path/to/logdir")
```

It's very helpful to use this feature during script development. The reports produced by *cgitb* provide information that can save you a lot of time in tracking down bugs. You can always remove the `cgitb` line later when you have tested your script and are confident that it works correctly.

To get at submitted form data, use the `FieldStorage` class. If the form contains non-ASCII characters, use the *encoding* keyword parameter set to the value of the encoding defined for the document. It is usually contained in the META tag in the HEAD section of the HTML document or by the *Content-Type* header). This reads the form contents from the standard input or the environment (depending on the value of various environment variables set according to the CGI standard). Since it may consume standard input, it should be instantiated only once.

The `FieldStorage` instance can be indexed like a Python dictionary. It allows membership testing with the `in` operator, and also supports the standard dictionary method *keys()* and the built-in function *len()*. Form fields containing empty strings are ignored and do not appear in the dictionary; to keep such values, provide a true value for the optional *keep_blank_values* keyword parameter when creating the `FieldStorage` instance.

For instance, the following code (which assumes that the *Content-Type* header and blank line have already been printed) checks that the fields `name` and `addr` are both set to a non-empty string:

```
form = cgi.FieldStorage()
if "name" not in form or "addr" not in form:
    print("<H1>Error</H1>")
    print("Please fill in the name and addr fields.")
    return
print("<p>name:", form["name"].value)
print("<p>addr:", form["addr"].value)
...further form processing here...
```

Here the fields, accessed through `form[key]`, are themselves instances of `FieldStorage` (or `MiniFieldStorage`, depending on the form encoding). The `value` attribute of the instance yields the string value of the field. The `getvalue()` method returns this string value directly; it also accepts an optional second argument as a default to return if the requested key is not present.

If the submitted form data contains more than one field with the same name, the object retrieved by `form[key]` is not a `FieldStorage` or `MiniFieldStorage` instance but a list of such instances. Similarly, in this situation, `form.getvalue(key)` would return a list of strings. If you expect this possibility (when your HTML form contains multiple fields with the same name), use the *getlist()* method, which always returns a list of values (so that you do not need to special-case the single item case). For example, this code concatenates any number of username fields, separated by commas:

```
value = form.getlist("username")
usernames = ",".join(value)
```

If a field represents an uploaded file, accessing the value via the `value` attribute or the `getvalue()` method reads the entire file in memory as bytes. This may not be what you want. You can test for an uploaded file by testing either the `filename` attribute or the `file` attribute. You can then read the data from the `file` attribute before it is automatically closed as part of the garbage collection of the `FieldStorage` instance (the *read()* and *readline()* methods will return bytes):

```
fileitem = form["userfile"]
if fileitem.file:
    # It's an uploaded file; count lines
    linecount = 0
    while True:
        line = fileitem.file.readline()
        if not line: break
        linecount = linecount + 1
```

`FieldStorage` objects also support being used in a `with` statement, which will automatically close them when done.

If an error is encountered when obtaining the contents of an uploaded file (for example, when the user interrupts the form submission by clicking on a Back or Cancel button) the `done` attribute of the object for the field will be set to the value -1.

The file upload draft standard entertains the possibility of uploading multiple files from one field (using a recursive *multipart/** encoding). When this occurs, the item will be a dictionary-like `FieldStorage` item. This can be determined by testing its `type` attribute, which should be *multipart/form-data* (or perhaps another MIME type matching *multipart/**). In this case, it can be iterated over recursively just like the top-level form object.

When a form is submitted in the "old" format (as the query string or as a single data part of type *application/x-www-form-urlencoded*), the items will actually be instances of the class `MiniFieldStorage`. In this case, the `list`, `file`, and `filename` attributes are always `None`.

A form submitted via POST that also has a query string will contain both `FieldStorage` and `MiniFieldStorage` items.

Changed in version 3.4: The `file` attribute is automatically closed upon the garbage collection of the creating `FieldStorage` instance.

Changed in version 3.5: Added support for the context management protocol to the `FieldStorage` class.

22.2.3 Higher Level Interface

The previous section explains how to read CGI form data using the `FieldStorage` class. This section describes a higher level interface which was added to this class to allow one to do it in a more readable and intuitive way. The interface doesn't make the techniques described in previous sections obsolete — they are still useful to process file uploads efficiently, for example.

The interface consists of two simple methods. Using the methods you can process form data in a generic way, without the need to worry whether only one or more values were posted under one name.

In the previous section, you learned to write following code anytime you expected a user to post more than one value under one name:

```
item = form.getvalue("item")
if isinstance(item, list):
    # The user is requesting more than one item.
else:
    # The user is requesting only one item.
```

This situation is common for example when a form contains a group of multiple checkboxes with the same name:

```
<input type="checkbox" name="item" value="1" />
<input type="checkbox" name="item" value="2" />
```

In most situations, however, there's only one form control with a particular name in a form and then you expect and need only one value associated with this name. So you write a script containing for example this code:

```
user = form.getvalue("user").upper()
```

The problem with the code is that you should never expect that a client will provide valid input to your scripts. For example, if a curious user appends another `user=foo` pair to the query string, then the script would crash, because in this situation the `getvalue("user")` method call returns a list instead of a string. Calling the *upper()* method on a list is not valid (since lists do not have a method of this name) and results in an *AttributeError* exception.

Therefore, the appropriate way to read form data values was to always use the code which checks whether the obtained value is a single value or a list of values. That's annoying and leads to less readable scripts.

A more convenient approach is to use the methods *getfirst()* and *getlist()* provided by this higher level interface.

`FieldStorage.getfirst`(*name*, *default=None*)

This method always returns only one value associated with form field *name*. The method returns only the first value in case that more values were posted under such name. Please note that the order in which the values are received may vary from browser to browser and should not be counted on.[1] If no

[1] Note that some recent versions of the HTML specification do state what order the field values should be supplied in, but knowing whether a request was received from a conforming browser, or even from a browser at all, is tedious and error-prone.

such form field or value exists then the method returns the value specified by the optional parameter *default*. This parameter defaults to `None` if not specified.

`FieldStorage.getlist(`*name*`)`
This method always returns a list of values associated with form field *name*. The method returns an empty list if no such form field or value exists for *name*. It returns a list consisting of one item if only one such value exists.

Using these methods you can write nice compact code:

```
import cgi
form = cgi.FieldStorage()
user = form.getfirst("user", "").upper()    # This way it's safe.
for item in form.getlist("item"):
    do_something(item)
```

22.2.4 Functions

These are useful if you want more control, or if you want to employ some of the algorithms implemented in this module in other circumstances.

`cgi.parse(`*fp=None*, *environ=os.environ*, *keep_blank_values=False*, *strict_parsing=False*`)`
Parse a query in the environment or from a file (the file defaults to `sys.stdin`). The *keep_blank_values* and *strict_parsing* parameters are passed to *urllib.parse.parse_qs()* unchanged.

`cgi.parse_qs(`*qs*, *keep_blank_values=False*, *strict_parsing=False*`)`
This function is deprecated in this module. Use *urllib.parse.parse_qs()* instead. It is maintained here only for backward compatibility.

`cgi.parse_qsl(`*qs*, *keep_blank_values=False*, *strict_parsing=False*`)`
This function is deprecated in this module. Use *urllib.parse.parse_qsl()* instead. It is maintained here only for backward compatibility.

`cgi.parse_multipart(`*fp*, *pdict*, *encoding="utf-8"*, *errors="replace"*`)`
Parse input of type *multipart/form-data* (for file uploads). Arguments are *fp* for the input file, *pdict* for a dictionary containing other parameters in the *Content-Type* header, and *encoding*, the request encoding.

Returns a dictionary just like *urllib.parse.parse_qs()*: keys are the field names, each value is a list of values for that field. For non-file fields, the value is a list of strings.

This is easy to use but not much good if you are expecting megabytes to be uploaded — in that case, use the `FieldStorage` class instead which is much more flexible.

Changed in version 3.7: Added the *encoding* and *errors* parameters. For non-file fields, the value is now a list of strings, not bytes.

`cgi.parse_header(`*string*`)`
Parse a MIME header (such as *Content-Type*) into a main value and a dictionary of parameters.

`cgi.test()`
Robust test CGI script, usable as main program. Writes minimal HTTP headers and formats all information provided to the script in HTML form.

`cgi.print_environ()`
Format the shell environment in HTML.

`cgi.print_form(`*form*`)`
Format a form in HTML.

cgi.print_directory()
Format the current directory in HTML.

cgi.print_environ_usage()
Print a list of useful (used by CGI) environment variables in HTML.

cgi.escape(*s*, *quote=False*)
Convert the characters '&', '<' and '>' in string *s* to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. If the optional flag *quote* is true, the quotation mark character (") is also translated; this helps for inclusion in an HTML attribute value delimited by double quotes, as in <a href="...">. Note that single quotes are never translated.

Deprecated since version 3.2: This function is unsafe because *quote* is false by default, and therefore deprecated. Use *html.escape()* instead.

22.2.5 Caring about security

There's one important rule: if you invoke an external program (via the *os.system()* or *os.popen()* functions. or others with similar functionality), make very sure you don't pass arbitrary strings received from the client to the shell. This is a well-known security hole whereby clever hackers anywhere on the Web can exploit a gullible CGI script to invoke arbitrary shell commands. Even parts of the URL or field names cannot be trusted, since the request doesn't have to come from your form!

To be on the safe side, if you must pass a string gotten from a form to a shell command, you should make sure the string contains only alphanumeric characters, dashes, underscores, and periods.

22.2.6 Installing your CGI script on a Unix system

Read the documentation for your HTTP server and check with your local system administrator to find the directory where CGI scripts should be installed; usually this is in a directory cgi-bin in the server tree.

Make sure that your script is readable and executable by "others"; the Unix file mode should be 0o755 octal (use chmod 0755 filename). Make sure that the first line of the script contains #! starting in column 1 followed by the pathname of the Python interpreter, for instance:

```
#!/usr/local/bin/python
```

Make sure the Python interpreter exists and is executable by "others".

Make sure that any files your script needs to read or write are readable or writable, respectively, by "others" — their mode should be 0o644 for readable and 0o666 for writable. This is because, for security reasons, the HTTP server executes your script as user "nobody", without any special privileges. It can only read (write, execute) files that everybody can read (write, execute). The current directory at execution time is also different (it is usually the server's cgi-bin directory) and the set of environment variables is also different from what you get when you log in. In particular, don't count on the shell's search path for executables (PATH) or the Python module search path (PYTHONPATH) to be set to anything interesting.

If you need to load modules from a directory which is not on Python's default module search path, you can change the path in your script, before importing other modules. For example:

```
import sys
sys.path.insert(0, "/usr/home/joe/lib/python")
sys.path.insert(0, "/usr/local/lib/python")
```

(This way, the directory inserted last will be searched first!)

Instructions for non-Unix systems will vary; check your HTTP server's documentation (it will usually have a section on CGI scripts).

22.2.7 Testing your CGI script

Unfortunately, a CGI script will generally not run when you try it from the command line, and a script that works perfectly from the command line may fail mysteriously when run from the server. There's one reason why you should still test your script from the command line: if it contains a syntax error, the Python interpreter won't execute it at all, and the HTTP server will most likely send a cryptic error to the client.

Assuming your script has no syntax errors, yet it does not work, you have no choice but to read the next section.

22.2.8 Debugging CGI scripts

First of all, check for trivial installation errors — reading the section above on installing your CGI script carefully can save you a lot of time. If you wonder whether you have understood the installation procedure correctly, try installing a copy of this module file (`cgi.py`) as a CGI script. When invoked as a script, the file will dump its environment and the contents of the form in HTML form. Give it the right mode etc, and send it a request. If it's installed in the standard `cgi-bin` directory, it should be possible to send it a request by entering a URL into your browser of the form:

```
http://yourhostname/cgi-bin/cgi.py?name=Joe+Blow&addr=At+Home
```

If this gives an error of type 404, the server cannot find the script – perhaps you need to install it in a different directory. If it gives another error, there's an installation problem that you should fix before trying to go any further. If you get a nicely formatted listing of the environment and form content (in this example, the fields should be listed as "addr" with value "At Home" and "name" with value "Joe Blow"), the `cgi.py` script has been installed correctly. If you follow the same procedure for your own script, you should now be able to debug it.

The next step could be to call the *cgi* module's *test()* function from your script: replace its main code with the single statement

```
cgi.test()
```

This should produce the same results as those gotten from installing the `cgi.py` file itself.

When an ordinary Python script raises an unhandled exception (for whatever reason: of a typo in a module name, a file that can't be opened, etc.), the Python interpreter prints a nice traceback and exits. While the Python interpreter will still do this when your CGI script raises an exception, most likely the traceback will end up in one of the HTTP server's log files, or be discarded altogether.

Fortunately, once you have managed to get your script to execute *some* code, you can easily send tracebacks to the Web browser using the *cgitb* module. If you haven't done so already, just add the lines:

```
import cgitb
cgitb.enable()
```

to the top of your script. Then try running it again; when a problem occurs, you should see a detailed report that will likely make apparent the cause of the crash.

If you suspect that there may be a problem in importing the *cgitb* module, you can use an even more robust approach (which only uses built-in modules):

```
import sys
sys.stderr = sys.stdout
print("Content-Type: text/plain")
print()
...your code here...
```

This relies on the Python interpreter to print the traceback. The content type of the output is set to plain text, which disables all HTML processing. If your script works, the raw HTML will be displayed by your client. If it raises an exception, most likely after the first two lines have been printed, a traceback will be displayed. Because no HTML interpretation is going on, the traceback will be readable.

22.2.9 Common problems and solutions

- Most HTTP servers buffer the output from CGI scripts until the script is completed. This means that it is not possible to display a progress report on the client's display while the script is running.
- Check the installation instructions above.
- Check the HTTP server's log files. (`tail -f logfile` in a separate window may be useful!)
- Always check a script for syntax errors first, by doing something like `python script.py`.
- If your script does not have any syntax errors, try adding `import cgitb; cgitb.enable()` to the top of the script.
- When invoking external programs, make sure they can be found. Usually, this means using absolute path names — `PATH` is usually not set to a very useful value in a CGI script.
- When reading or writing external files, make sure they can be read or written by the userid under which your CGI script will be running: this is typically the userid under which the web server is running, or some explicitly specified userid for a web server's `suexec` feature.
- Don't try to give a CGI script a set-uid mode. This doesn't work on most systems, and is a security liability as well.

22.3 `cgitb` — Traceback manager for CGI scripts

Source code: Lib/cgitb.py

The *cgitb* module provides a special exception handler for Python scripts. (Its name is a bit misleading. It was originally designed to display extensive traceback information in HTML for CGI scripts. It was later generalized to also display this information in plain text.) After this module is activated, if an uncaught exception occurs, a detailed, formatted report will be displayed. The report includes a traceback showing excerpts of the source code for each level, as well as the values of the arguments and local variables to currently running functions, to help you debug the problem. Optionally, you can save this information to a file instead of sending it to the browser.

To enable this feature, simply add this to the top of your CGI script:

```
import cgitb
cgitb.enable()
```

The options to the *enable()* function control whether the report is displayed in the browser and whether the report is logged to a file for later analysis.

`cgitb.enable`(*display=1*, *logdir=None*, *context=5*, *format="html"*)

This function causes the *cgitb* module to take over the interpreter's default handling for exceptions by setting the value of *sys.excepthook*.

The optional argument *display* defaults to `1` and can be set to `0` to suppress sending the traceback to the browser. If the argument *logdir* is present, the traceback reports are written to files. The value of *logdir* should be a directory where these files will be placed. The optional argument *context* is the number of lines of context to display around the current line of source code in the traceback; this

defaults to 5. If the optional argument *format* is "html", the output is formatted as HTML. Any other value forces plain text output. The default value is "html".

cgitb.**text**(*info*, *context=5*)
: This function handles the exception described by *info* (a 3-tuple containing the result of *sys.exc_info()*), formatting its traceback as text and returning the result as a string. The optional argument *context* is the number of lines of context to display around the current line of source code in the traceback; this defaults to 5.

cgitb.**html**(*info*, *context=5*)
: This function handles the exception described by *info* (a 3-tuple containing the result of *sys.exc_info()*), formatting its traceback as HTML and returning the result as a string. The optional argument *context* is the number of lines of context to display around the current line of source code in the traceback; this defaults to 5.

cgitb.**handler**(*info=None*)
: This function handles an exception using the default settings (that is, show a report in the browser, but don't log to a file). This can be used when you've caught an exception and want to report it using *cgitb*. The optional *info* argument should be a 3-tuple containing an exception type, exception value, and traceback object, exactly like the tuple returned by *sys.exc_info()*. If the *info* argument is not supplied, the current exception is obtained from *sys.exc_info()*.

22.4 wsgiref — WSGI Utilities and Reference Implementation

The Web Server Gateway Interface (WSGI) is a standard interface between web server software and web applications written in Python. Having a standard interface makes it easy to use an application that supports WSGI with a number of different web servers.

Only authors of web servers and programming frameworks need to know every detail and corner case of the WSGI design. You don't need to understand every detail of WSGI just to install a WSGI application or to write a web application using an existing framework.

wsgiref is a reference implementation of the WSGI specification that can be used to add WSGI support to a web server or framework. It provides utilities for manipulating WSGI environment variables and response headers, base classes for implementing WSGI servers, a demo HTTP server that serves WSGI applications, and a validation tool that checks WSGI servers and applications for conformance to the WSGI specification (**PEP 3333**).

See wsgi.readthedocs.io for more information about WSGI, and links to tutorials and other resources.

22.4.1 wsgiref.util – WSGI environment utilities

This module provides a variety of utility functions for working with WSGI environments. A WSGI environment is a dictionary containing HTTP request variables as described in **PEP 3333**. All of the functions taking an *environ* parameter expect a WSGI-compliant dictionary to be supplied; please see **PEP 3333** for a detailed specification.

wsgiref.util.**guess_scheme**(*environ*)
: Return a guess for whether wsgi.url_scheme should be "http" or "https", by checking for a HTTPS environment variable in the *environ* dictionary. The return value is a string.

  This function is useful when creating a gateway that wraps CGI or a CGI-like protocol such as FastCGI. Typically, servers providing such protocols will include a HTTPS variable with a value of "1" "yes", or "on" when a request is received via SSL. So, this function returns "https" if such a value is found, and "http" otherwise.

wsgiref.util.request_uri(*environ*, *include_query=True*)
: Return the full request URI, optionally including the query string, using the algorithm found in the "URL Reconstruction" section of **PEP 3333**. If *include_query* is false, the query string is not included in the resulting URI.

wsgiref.util.application_uri(*environ*)
: Similar to *request_uri()*, except that the PATH_INFO and QUERY_STRING variables are ignored. The result is the base URI of the application object addressed by the request.

wsgiref.util.shift_path_info(*environ*)
: Shift a single name from PATH_INFO to SCRIPT_NAME and return the name. The *environ* dictionary is *modified* in-place; use a copy if you need to keep the original PATH_INFO or SCRIPT_NAME intact.

  If there are no remaining path segments in PATH_INFO, None is returned.

  Typically, this routine is used to process each portion of a request URI path, for example to treat the path as a series of dictionary keys. This routine modifies the passed-in environment to make it suitable for invoking another WSGI application that is located at the target URI. For example, if there is a WSGI application at /foo, and the request URI path is /foo/bar/baz, and the WSGI application at /foo calls *shift_path_info()*, it will receive the string "bar", and the environment will be updated to be suitable for passing to a WSGI application at /foo/bar. That is, SCRIPT_NAME will change from /foo to /foo/bar, and PATH_INFO will change from /bar/baz to /baz.

  When PATH_INFO is just a "/", this routine returns an empty string and appends a trailing slash to SCRIPT_NAME, even though empty path segments are normally ignored, and SCRIPT_NAME doesn't normally end in a slash. This is intentional behavior, to ensure that an application can tell the difference between URIs ending in /x from ones ending in /x/ when using this routine to do object traversal.

wsgiref.util.setup_testing_defaults(*environ*)
: Update *environ* with trivial defaults for testing purposes.

  This routine adds various parameters required for WSGI, including HTTP_HOST, SERVER_NAME, SERVER_PORT, REQUEST_METHOD, SCRIPT_NAME, PATH_INFO, and all of the **PEP 3333**-defined wsgi.* variables. It only supplies default values, and does not replace any existing settings for these variables.

  This routine is intended to make it easier for unit tests of WSGI servers and applications to set up dummy environments. It should NOT be used by actual WSGI servers or applications, since the data is fake!

  Example usage:

```
from wsgiref.util import setup_testing_defaults
from wsgiref.simple_server import make_server

# A relatively simple WSGI application. It's going to print out the
# environment dictionary after being updated by setup_testing_defaults
def simple_app(environ, start_response):
    setup_testing_defaults(environ)

    status = '200 OK'
    headers = [('Content-type', 'text/plain; charset=utf-8')]

    start_response(status, headers)

    ret = [("%s: %s\n" % (key, value)).encode("utf-8")
           for key, value in environ.items()]
    return ret
```

(continues on next page)

(continued from previous page)

```
with make_server('', 8000, simple_app) as httpd:
    print("Serving on port 8000...")
    httpd.serve_forever()
```

In addition to the environment functions above, the *wsgiref.util* module also provides these miscellaneous utilities:

wsgiref.util.is_hop_by_hop(*header_name*)
Return true if 'header_name' is an HTTP/1.1 "Hop-by-Hop" header, as defined by **RFC 2616**.

class wsgiref.util.FileWrapper(*filelike*, *blksize=8192*)
A wrapper to convert a file-like object to an *iterator*. The resulting objects support both `__getitem__()` and `__iter__()` iteration styles, for compatibility with Python 2.1 and Jython. As the object is iterated over, the optional *blksize* parameter will be repeatedly passed to the *filelike* object's `read()` method to obtain bytestrings to yield. When `read()` returns an empty bytestring, iteration is ended and is not resumable.

If *filelike* has a `close()` method, the returned object will also have a `close()` method, and it will invoke the *filelike* object's `close()` method when called.

Example usage:

```
from io import StringIO
from wsgiref.util import FileWrapper

# We're using a StringIO-buffer for as the file-like object
filelike = StringIO("This is an example file-like object"*10)
wrapper = FileWrapper(filelike, blksize=5)

for chunk in wrapper:
    print(chunk)
```

22.4.2 wsgiref.headers – WSGI response header tools

This module provides a single class, *Headers*, for convenient manipulation of WSGI response headers using a mapping-like interface.

class wsgiref.headers.Headers([*headers*])
Create a mapping-like object wrapping *headers*, which must be a list of header name/value tuples as described in **PEP 3333**. The default value of *headers* is an empty list.

Headers objects support typical mapping operations including `__getitem__()`, `get()`, `__setitem__()`, `setdefault()`, `__delitem__()` and `__contains__()`. For each of these methods, the key is the header name (treated case-insensitively), and the value is the first value associated with that header name. Setting a header deletes any existing values for that header, then adds a new value at the end of the wrapped header list. Headers' existing order is generally maintained, with new headers added to the end of the wrapped list.

Unlike a dictionary, *Headers* objects do not raise an error when you try to get or delete a key that isn't in the wrapped header list. Getting a nonexistent header just returns `None`, and deleting a nonexistent header does nothing.

Headers objects also support `keys()`, `values()`, and `items()` methods. The lists returned by `keys()` and `items()` can include the same key more than once if there is a multi-valued header. The `len()` of a *Headers* object is the same as the length of its `items()`, which is the same as the length of the wrapped header list. In fact, the `items()` method just returns a copy of the wrapped header list.

Calling `bytes()` on a *Headers* object returns a formatted bytestring suitable for transmission as HTTP response headers. Each header is placed on a line with its value, separated by a colon and a space. Each line is terminated by a carriage return and line feed, and the bytestring is terminated with a blank line.

In addition to their mapping interface and formatting features, *Headers* objects also have the following methods for querying and adding multi-valued headers, and for adding headers with MIME parameters:

get_all(*name*)
Return a list of all the values for the named header.

The returned list will be sorted in the order they appeared in the original header list or were added to this instance, and may contain duplicates. Any fields deleted and re-inserted are always appended to the header list. If no fields exist with the given name, returns an empty list.

add_header(*name*, *value*, ***_params*)
Add a (possibly multi-valued) header, with optional MIME parameters specified via keyword arguments.

name is the header field to add. Keyword arguments can be used to set MIME parameters for the header field. Each parameter must be a string or `None`. Underscores in parameter names are converted to dashes, since dashes are illegal in Python identifiers, but many MIME parameter names include dashes. If the parameter value is a string, it is added to the header value parameters in the form `name="value"`. If it is `None`, only the parameter name is added. (This is used for MIME parameters without a value.) Example usage:

```
h.add_header('content-disposition', 'attachment', filename='bud.gif')
```

The above will add a header that looks like this:

```
Content-Disposition: attachment; filename="bud.gif"
```

Changed in version 3.5: *headers* parameter is optional.

22.4.3 wsgiref.simple_server – a simple WSGI HTTP server

This module implements a simple HTTP server (based on *http.server*) that serves WSGI applications. Each server instance serves a single WSGI application on a given host and port. If you want to serve multiple applications on a single host and port, you should create a WSGI application that parses `PATH_INFO` to select which application to invoke for each request. (E.g., using the `shift_path_info()` function from *wsgiref.util*.)

wsgiref.simple_server.make_server(*host*, *port*, *app*, *server_class=WSGIServer*, *handler_class=WSGIRequestHandler*)
Create a new WSGI server listening on *host* and *port*, accepting connections for *app*. The return value is an instance of the supplied *server_class*, and will process requests using the specified *handler_class*. *app* must be a WSGI application object, as defined by **PEP 3333**.

Example usage:

```
from wsgiref.simple_server import make_server, demo_app

with make_server('', 8000, demo_app) as httpd:
    print("Serving HTTP on port 8000...")

    # Respond to requests until process is killed
```

(continues on next page)

(continued from previous page)

```
    httpd.serve_forever()

    # Alternative: serve one request, then exit
    httpd.handle_request()
```

wsgiref.simple_server.**demo_app**(*environ*, *start_response*)
This function is a small but complete WSGI application that returns a text page containing the message "Hello world!" and a list of the key/value pairs provided in the *environ* parameter. It's useful for verifying that a WSGI server (such as *wsgiref.simple_server*) is able to run a simple WSGI application correctly.

class wsgiref.simple_server.**WSGIServer**(*server_address*, *RequestHandlerClass*)
Create a *WSGIServer* instance. *server_address* should be a `(host,port)` tuple, and *RequestHandlerClass* should be the subclass of *http.server.BaseHTTPRequestHandler* that will be used to process requests.

You do not normally need to call this constructor, as the *make_server()* function can handle all the details for you.

WSGIServer is a subclass of *http.server.HTTPServer*, so all of its methods (such as `serve_forever()` and `handle_request()`) are available. *WSGIServer* also provides these WSGI-specific methods:

set_app(*application*)
Sets the callable *application* as the WSGI application that will receive requests.

get_app()
Returns the currently-set application callable.

Normally, however, you do not need to use these additional methods, as *set_app()* is normally called by *make_server()*, and the *get_app()* exists mainly for the benefit of request handler instances.

class wsgiref.simple_server.**WSGIRequestHandler**(*request*, *client_address*, *server*)
Create an HTTP handler for the given *request* (i.e. a socket), *client_address* (a `(host,port)` tuple), and *server* (*WSGIServer* instance).

You do not need to create instances of this class directly; they are automatically created as needed by *WSGIServer* objects. You can, however, subclass this class and supply it as a *handler_class* to the *make_server()* function. Some possibly relevant methods for overriding in subclasses:

get_environ()
Returns a dictionary containing the WSGI environment for a request. The default implementation copies the contents of the *WSGIServer* object's `base_environ` dictionary attribute and then adds various headers derived from the HTTP request. Each call to this method should return a new dictionary containing all of the relevant CGI environment variables as specified in **PEP 3333**.

get_stderr()
Return the object that should be used as the `wsgi.errors` stream. The default implementation just returns `sys.stderr`.

handle()
Process the HTTP request. The default implementation creates a handler instance using a *wsgiref.handlers* class to implement the actual WSGI application interface.

22.4.4 wsgiref.validate — WSGI conformance checker

When creating new WSGI application objects, frameworks, servers, or middleware, it can be useful to validate the new code's conformance using *wsgiref.validate*. This module provides a function that creates WSGI

application objects that validate communications between a WSGI server or gateway and a WSGI application object, to check both sides for protocol conformance.

Note that this utility does not guarantee complete **PEP 3333** compliance; an absence of errors from this module does not necessarily mean that errors do not exist. However, if this module does produce an error, then it is virtually certain that either the server or application is not 100% compliant.

This module is based on the `paste.lint` module from Ian Bicking's "Python Paste" library.

`wsgiref.validate.validator`(*application*)
Wrap *application* and return a new WSGI application object. The returned application will forward all requests to the original *application*, and will check that both the *application* and the server invoking it are conforming to the WSGI specification and to **RFC 2616**.

Any detected nonconformance results in an *AssertionError* being raised; note, however, that how these errors are handled is server-dependent. For example, *wsgiref.simple_server* and other servers based on *wsgiref.handlers* (that don't override the error handling methods to do something else) will simply output a message that an error has occurred, and dump the traceback to `sys.stderr` or some other error stream.

This wrapper may also generate output using the *warnings* module to indicate behaviors that are questionable but which may not actually be prohibited by **PEP 3333**. Unless they are suppressed using Python command-line options or the *warnings* API, any such warnings will be written to `sys.stderr` (*not* `wsgi.errors`, unless they happen to be the same object).

Example usage:

```
from wsgiref.validate import validator
from wsgiref.simple_server import make_server

# Our callable object which is intentionally not compliant to the
# standard, so the validator is going to break
def simple_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain')]  # HTTP Headers
    start_response(status, headers)

    # This is going to break because we need to return a list, and
    # the validator is going to inform us
    return b"Hello World"

# This is the application wrapped in a validator
validator_app = validator(simple_app)

with make_server('', 8000, validator_app) as httpd:
    print("Listening on port 8000....")
    httpd.serve_forever()
```

22.4.5 `wsgiref.handlers` – server/gateway base classes

This module provides base handler classes for implementing WSGI servers and gateways. These base classes handle most of the work of communicating with a WSGI application, as long as they are given a CGI-like environment, along with input, output, and error streams.

`class wsgiref.handlers.CGIHandler`
CGI-based invocation via `sys.stdin`, `sys.stdout`, `sys.stderr` and `os.environ`. This is useful when

you have a WSGI application and want to run it as a CGI script. Simply invoke `CGIHandler().run(app)`, where `app` is the WSGI application object you wish to invoke.

This class is a subclass of *BaseCGIHandler* that sets `wsgi.run_once` to true, `wsgi.multithread` to false, and `wsgi.multiprocess` to true, and always uses *sys* and *os* to obtain the necessary CGI streams and environment.

class wsgiref.handlers.IISCGIHandler

A specialized alternative to *CGIHandler*, for use when deploying on Microsoft's IIS web server, without having set the config allowPathInfo option (IIS>=7) or metabase allowPathInfoForScriptMappings (IIS<7).

By default, IIS gives a `PATH_INFO` that duplicates the `SCRIPT_NAME` at the front, causing problems for WSGI applications that wish to implement routing. This handler strips any such duplicated path.

IIS can be configured to pass the correct `PATH_INFO`, but this causes another bug where `PATH_TRANSLATED` is wrong. Luckily this variable is rarely used and is not guaranteed by WSGI. On IIS<7, though, the setting can only be made on a vhost level, affecting all other script mappings, many of which break when exposed to the `PATH_TRANSLATED` bug. For this reason IIS<7 is almost never deployed with the fix. (Even IIS7 rarely uses it because there is still no UI for it.)

There is no way for CGI code to tell whether the option was set, so a separate handler class is provided. It is used in the same way as *CGIHandler*, i.e., by calling `IISCGIHandler().run(app)`, where `app` is the WSGI application object you wish to invoke.

New in version 3.2.

class wsgiref.handlers.BaseCGIHandler(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

Similar to *CGIHandler*, but instead of using the *sys* and *os* modules, the CGI environment and I/O streams are specified explicitly. The *multithread* and *multiprocess* values are used to set the `wsgi.multithread` and `wsgi.multiprocess` flags for any applications run by the handler instance.

This class is a subclass of *SimpleHandler* intended for use with software other than HTTP "origin servers". If you are writing a gateway protocol implementation (such as CGI, FastCGI, SCGI, etc.) that uses a `Status:` header to send an HTTP status, you probably want to subclass this instead of *SimpleHandler*.

class wsgiref.handlers.SimpleHandler(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

Similar to *BaseCGIHandler*, but designed for use with HTTP origin servers. If you are writing an HTTP server implementation, you will probably want to subclass this instead of *BaseCGIHandler*.

This class is a subclass of *BaseHandler*. It overrides the `__init__()`, `get_stdin()`, `get_stderr()`, `add_cgi_vars()`, `_write()`, and `_flush()` methods to support explicitly setting the environment and streams via the constructor. The supplied environment and streams are stored in the `stdin`, `stdout`, `stderr`, and `environ` attributes.

The *write()* method of *stdout* should write each chunk in full, like *io.BufferedIOBase*.

class wsgiref.handlers.BaseHandler

This is an abstract base class for running WSGI applications. Each instance will handle a single HTTP request, although in principle you could create a subclass that was reusable for multiple requests.

BaseHandler instances have only one method intended for external use:

run(*app*)

Run the specified WSGI application, *app*.

All of the other *BaseHandler* methods are invoked by this method in the process of running the application, and thus exist primarily to allow customizing the process.

The following methods MUST be overridden in a subclass:

_write(*data*)
Buffer the bytes *data* for transmission to the client. It's okay if this method actually transmits the data; *BaseHandler* just separates write and flush operations for greater efficiency when the underlying system actually has such a distinction.

_flush()
Force buffered data to be transmitted to the client. It's okay if this method is a no-op (i.e., if *_write()* actually sends the data).

get_stdin()
Return an input stream object suitable for use as the `wsgi.input` of the request currently being processed.

get_stderr()
Return an output stream object suitable for use as the `wsgi.errors` of the request currently being processed.

add_cgi_vars()
Insert CGI variables for the current request into the `environ` attribute.

Here are some other methods and attributes you may wish to override. This list is only a summary, however, and does not include every method that can be overridden. You should consult the docstrings and source code for additional information before attempting to create a customized *BaseHandler* subclass.

Attributes and methods for customizing the WSGI environment:

wsgi_multithread
The value to be used for the `wsgi.multithread` environment variable. It defaults to true in *BaseHandler*, but may have a different default (or be set by the constructor) in the other subclasses.

wsgi_multiprocess
The value to be used for the `wsgi.multiprocess` environment variable. It defaults to true in *BaseHandler*, but may have a different default (or be set by the constructor) in the other subclasses.

wsgi_run_once
The value to be used for the `wsgi.run_once` environment variable. It defaults to false in *BaseHandler*, but *CGIHandler* sets it to true by default.

os_environ
The default environment variables to be included in every request's WSGI environment. By default, this is a copy of `os.environ` at the time that *wsgiref.handlers* was imported, but subclasses can either create their own at the class or instance level. Note that the dictionary should be considered read-only, since the default value is shared between multiple classes and instances.

server_software
If the *origin_server* attribute is set, this attribute's value is used to set the default `SERVER_SOFTWARE` WSGI environment variable, and also to set a default `Server:` header in HTTP responses. It is ignored for handlers (such as *BaseCGIHandler* and *CGIHandler*) that are not HTTP origin servers.

Changed in version 3.3: The term "Python" is replaced with implementation specific term like "CPython", "Jython" etc.

get_scheme()
Return the URL scheme being used for the current request. The default implementation uses the `guess_scheme()` function from *wsgiref.util* to guess whether the scheme should be "http" or "https", based on the current request's `environ` variables.

setup_environ()
Set the `environ` attribute to a fully-populated WSGI environment. The default implementation uses all of the above methods and attributes, plus the *get_stdin()*, *get_stderr()*, and *add_cgi_vars()* methods and the *wsgi_file_wrapper* attribute. It also inserts a `SERVER_SOFTWARE` key if not present, as long as the *origin_server* attribute is a true value and the *server_software* attribute is set.

Methods and attributes for customizing exception handling:

log_exception(*exc_info*)
Log the *exc_info* tuple in the server log. *exc_info* is a `(type, value, traceback)` tuple. The default implementation simply writes the traceback to the request's `wsgi.errors` stream and flushes it. Subclasses can override this method to change the format or retarget the output, mail the traceback to an administrator, or whatever other action may be deemed suitable.

traceback_limit
The maximum number of frames to include in tracebacks output by the default *log_exception()* method. If `None`, all frames are included.

error_output(*environ*, *start_response*)
This method is a WSGI application to generate an error page for the user. It is only invoked if an error occurs before headers are sent to the client.

This method can access the current error information using `sys.exc_info()`, and should pass that information to *start_response* when calling it (as described in the "Error Handling" section of **PEP 3333**).

The default implementation just uses the *error_status*, *error_headers*, and *error_body* attributes to generate an output page. Subclasses can override this to produce more dynamic error output.

Note, however, that it's not recommended from a security perspective to spit out diagnostics to any old user; ideally, you should have to do something special to enable diagnostic output, which is why the default implementation doesn't include any.

error_status
The HTTP status used for error responses. This should be a status string as defined in **PEP 3333**; it defaults to a 500 code and message.

error_headers
The HTTP headers used for error responses. This should be a list of WSGI response headers (`(name, value)` tuples), as described in **PEP 3333**. The default list just sets the content type to `text/plain`.

error_body
The error response body. This should be an HTTP response body bytestring. It defaults to the plain text, "A server error occurred. Please contact the administrator."

Methods and attributes for **PEP 3333**'s "Optional Platform-Specific File Handling" feature:

wsgi_file_wrapper
A `wsgi.file_wrapper` factory, or `None`. The default value of this attribute is the *wsgiref.util.FileWrapper* class.

sendfile()
Override to implement platform-specific file transmission. This method is called only if the application's return value is an instance of the class specified by the *wsgi_file_wrapper* attribute. It should return a true value if it was able to successfully transmit the file, so that the default transmission code will not be executed. The default implementation of this method just returns a false value.

Miscellaneous methods and attributes:

origin_server
This attribute should be set to a true value if the handler's *_write()* and *_flush()* are being used to communicate directly to the client, rather than via a CGI-like gateway protocol that wants the HTTP status in a special `Status:` header.

This attribute's default value is true in *BaseHandler*, but false in *BaseCGIHandler* and *CGIHandler*.

http_version
If *origin_server* is true, this string attribute is used to set the HTTP version of the response set to the client. It defaults to `"1.0"`.

wsgiref.handlers.read_environ()
Transcode CGI variables from `os.environ` to PEP 3333 "bytes in unicode" strings, returning a new dictionary. This function is used by *CGIHandler* and *IISCGIHandler* in place of directly using `os.environ`, which is not necessarily WSGI-compliant on all platforms and web servers using Python 3 – specifically, ones where the OS's actual environment is Unicode (i.e. Windows), or ones where the environment is bytes, but the system encoding used by Python to decode it is anything other than ISO-8859-1 (e.g. Unix systems using UTF-8).

If you are implementing a CGI-based handler of your own, you probably want to use this routine instead of just copying values out of `os.environ` directly.

New in version 3.2.

22.4.6 Examples

This is a working "Hello World" WSGI application:

```
from wsgiref.simple_server import make_server

# Every WSGI application must have an application object - a callable
# object that accepts two arguments. For that purpose, we're going to
# use a function (note that you're not limited to a function, you can
# use a class for example). The first argument passed to the function
# is a dictionary containing CGI-style environment variables and the
# second variable is the callable object (see PEP 333).
def hello_world_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain; charset=utf-8')]  # HTTP Headers
    start_response(status, headers)

    # The returned object is going to be printed
    return [b"Hello World"]

with make_server('', 8000, hello_world_app) as httpd:
    print("Serving on port 8000...")

    # Serve until process is killed
    httpd.serve_forever()
```

22.5 urllib — URL handling modules

Source code: Lib/urllib/

`urllib` is a package that collects several modules for working with URLs:

- *urllib.request* for opening and reading URLs
- *urllib.error* containing the exceptions raised by *urllib.request*
- *urllib.parse* for parsing URLs
- *urllib.robotparser* for parsing `robots.txt` files

22.6 `urllib.request` — Extensible library for opening URLs

Source code: Lib/urllib/request.py

The *urllib.request* module defines functions and classes which help in opening URLs (mostly HTTP) in a complex world — basic and digest authentication, redirections, cookies and more.

See also:

The Requests package is recommended for a higher-level HTTP client interface.

The *urllib.request* module defines the following functions:

`urllib.request.urlopen`(*url*, *data=None*[, *timeout*], *, *cafile=None*, *capath=None*, *cadefault=False*, *context=None*)

Open the URL *url*, which can be either a string or a *Request* object.

data must be an object specifying additional data to be sent to the server, or `None` if no such data is needed. See *Request* for details.

urllib.request module uses HTTP/1.1 and includes `Connection:close` header in its HTTP requests.

The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). This actually only works for HTTP, HTTPS and FTP connections.

If *context* is specified, it must be a *ssl.SSLContext* instance describing the various SSL options. See *HTTPSConnection* for more details.

The optional *cafile* and *capath* parameters specify a set of trusted CA certificates for HTTPS requests. *cafile* should point to a single file containing a bundle of CA certificates, whereas *capath* should point to a directory of hashed certificate files. More information can be found in *ssl.SSLContext.load_verify_locations()*.

The *cadefault* parameter is ignored.

This function always returns an object which can work as a *context manager* and has methods such as

- `geturl()` — return the URL of the resource retrieved, commonly used to determine if a redirect was followed
- `info()` — return the meta-information of the page, such as headers, in the form of an *email.message_from_string()* instance (see Quick Reference to HTTP Headers)
- `getcode()` – return the HTTP status code of the response.

For HTTP and HTTPS URLs, this function returns a *http.client.HTTPResponse* object slightly modified. In addition to the three new methods above, the msg attribute contains the same information as the *reason* attribute — the reason phrase returned by server — instead of the response headers as it is specified in the documentation for *HTTPResponse*.

For FTP, file, and data URLs and requests explicitly handled by legacy *URLopener* and *FancyURLopener* classes, this function returns a `urllib.response.addinfourl` object.

Raises *URLError* on protocol errors.

Note that `None` may be returned if no handler handles the request (though the default installed global *OpenerDirector* uses *UnknownHandler* to ensure this never happens).

In addition, if proxy settings are detected (for example, when a `*_proxy` environment variable like `http_proxy` is set), *ProxyHandler* is default installed and makes sure the requests are handled through the proxy.

The legacy `urllib.urlopen` function from Python 2.6 and earlier has been discontinued; *urllib.request.urlopen()* corresponds to the old `urllib2.urlopen`. Proxy handling, which was done by passing a dictionary parameter to `urllib.urlopen`, can be obtained by using *ProxyHandler* objects.

Changed in version 3.2: *cafile* and *capath* were added.

Changed in version 3.2: HTTPS virtual hosts are now supported if possible (that is, if *ssl.HAS_SNI* is true).

New in version 3.2: *data* can be an iterable object.

Changed in version 3.3: *cadefault* was added.

Changed in version 3.4.3: *context* was added.

Deprecated since version 3.6: *cafile*, *capath* and *cadefault* 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.

`urllib.request.install_opener(`*opener*`)`
Install an *OpenerDirector* instance as the default global opener. Installing an opener is only necessary if you want urlopen to use that opener; otherwise, simply call *OpenerDirector.open()* instead of *urlopen()*. The code does not check for a real *OpenerDirector*, and any class with the appropriate interface will work.

`urllib.request.build_opener(`[*handler*, ...]`)`
Return an *OpenerDirector* instance, which chains the handlers in the order given. *handler*s can be either instances of *BaseHandler*, or subclasses of *BaseHandler* (in which case it must be possible to call the constructor without any parameters). Instances of the following classes will be in front of the *handler*s, unless the *handler*s contain them, instances of them or subclasses of them: *ProxyHandler* (if proxy settings are detected), *UnknownHandler*, *HTTPHandler*, *HTTPDefaultErrorHandler*, *HTTPRedirectHandler*, *FTPHandler*, *FileHandler*, *HTTPErrorProcessor*.

If the Python installation has SSL support (i.e., if the *ssl* module can be imported), *HTTPSHandler* will also be added.

A *BaseHandler* subclass may also change its `handler_order` attribute to modify its position in the handlers list.

`urllib.request.pathname2url(`*path*`)`
Convert the pathname *path* from the local syntax for a path to the form used in the path component of a URL. This does not produce a complete URL. The return value will already be quoted using the *quote()* function.

`urllib.request.url2pathname(`*path*`)`
Convert the path component *path* from a percent-encoded URL to the local syntax for a path. This does not accept a complete URL. This function uses *unquote()* to decode *path*.

`urllib.request.getproxies()`
This helper function returns a dictionary of scheme to proxy server URL mappings. It scans the environment for variables named `<scheme>_proxy`, in a case insensitive approach, for all operating systems first, and when it cannot find it, looks for proxy information from Mac OSX System Configuration for

Mac OS X and Windows Systems Registry for Windows. If both lowercase and uppercase environment variables exist (and disagree), lowercase is preferred.

Note: If the environment variable `REQUEST_METHOD` is set, which usually indicates your script is running in a CGI environment, the environment variable `HTTP_PROXY` (uppercase `_PROXY`) will be ignored. This is because that variable can be injected by a client using the "Proxy:" HTTP header. If you need to use an HTTP proxy in a CGI environment, either use `ProxyHandler` explicitly, or make sure the variable name is in lowercase (or at least the `_proxy` suffix).

The following classes are provided:

class urllib.request.Request(*url*, *data=None*, *headers={}*, *origin_req_host=None*, *unverifiable=False*, *method=None*)

This class is an abstraction of a URL request.

url should be a string containing a valid URL.

data must be an object specifying additional data to send to the server, or `None` if no such data is needed. Currently HTTP requests are the only ones that use *data*. The supported object types include bytes, file-like objects, and iterables. If no `Content-Length` nor `Transfer-Encoding` header field has been provided, *HTTPHandler* will set these headers according to the type of *data*. `Content-Length` will be used to send bytes objects, while `Transfer-Encoding: chunked` as specified in **RFC 7230**, Section 3.3.1 will be used to send files and other iterables.

For an HTTP POST request method, *data* should be a buffer in the standard *application/x-www-form-urlencoded* format. The *urllib.parse.urlencode()* function takes a mapping or sequence of 2-tuples and returns an ASCII string in this format. It should be encoded to bytes before being used as the *data* parameter.

headers should be a dictionary, and will be treated as if *add_header()* was called with each key and value as arguments. This is often used to "spoof" the `User-Agent` header value, which is used by a browser to identify itself – some HTTP servers only allow requests coming from common browsers as opposed to scripts. For example, Mozilla Firefox may identify itself as `"Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11"`, while *urllib*'s default user agent string is `"Python-urllib/2.6"` (on Python 2.6).

An appropriate `Content-Type` header should be included if the *data* argument is present. If this header has not been provided and *data* is not None, `Content-Type: application/x-www-form-urlencoded` will be added as a default.

The final two arguments are only of interest for correct handling of third-party HTTP cookies:

origin_req_host should be the request-host of the origin transaction, as defined by **RFC 2965**. It defaults to `http.cookiejar.request_host(self)`. This is the host name or IP address of the original request that was initiated by the user. For example, if the request is for an image in an HTML document, this should be the request-host of the request for the page containing the image.

unverifiable should indicate whether the request is unverifiable, as defined by **RFC 2965**. It defaults to `False`. An unverifiable request is one whose URL the user did not have the option to approve. For example, if the request is for an image in an HTML document, and the user had no option to approve the automatic fetching of the image, this should be true.

method should be a string that indicates the HTTP request method that will be used (e.g. `'HEAD'`). If provided, its value is stored in the *method* attribute and is used by *get_method()*. The default is `'GET'` if *data* is `None` or `'POST'` otherwise. Subclasses may indicate a different default method by setting the *method* attribute in the class itself.

Note: The request will not work as expected if the data object is unable to deliver its content more than once (e.g. a file or an iterable that can produce the content only once) and the request is retried for

HTTP redirects or authentication. The *data* is sent to the HTTP server right away after the headers. There is no support for a 100-continue expectation in the library.

Changed in version 3.3: *Request.method* argument is added to the Request class.

Changed in version 3.4: Default *Request.method* may be indicated at the class level.

Changed in version 3.6: Do not raise an error if the `Content-Length` has not been provided and *data* is neither `None` nor a bytes object. Fall back to use chunked transfer encoding instead.

class urllib.request.OpenerDirector
The *OpenerDirector* class opens URLs via *BaseHandler*s chained together. It manages the chaining of handlers, and recovery from errors.

class urllib.request.BaseHandler
This is the base class for all registered handlers — and handles only the simple mechanics of registration.

class urllib.request.HTTPDefaultErrorHandler
A class which defines a default handler for HTTP error responses; all responses are turned into *HTTPError* exceptions.

class urllib.request.HTTPRedirectHandler
A class to handle redirections.

class urllib.request.HTTPCookieProcessor(*cookiejar=None*)
A class to handle HTTP Cookies.

class urllib.request.ProxyHandler(*proxies=None*)
Cause requests to go through a proxy. If *proxies* is given, it must be a dictionary mapping protocol names to URLs of proxies. The default is to read the list of proxies from the environment variables `<protocol>_proxy`. If no proxy environment variables are set, then in a Windows environment proxy settings are obtained from the registry's Internet Settings section, and in a Mac OS X environment proxy information is retrieved from the OS X System Configuration Framework.

To disable autodetected proxy pass an empty dictionary.

The `no_proxy` environment variable can be used to specify hosts which shouldn't be reached via proxy; if set, it should be a comma-separated list of hostname suffixes, optionally with `:port` appended, for example `cern.ch,ncsa.uiuc.edu,some.host:8080`.

Note: `HTTP_PROXY` will be ignored if a variable `REQUEST_METHOD` is set; see the documentation on *getproxies()*.

class urllib.request.HTTPPasswordMgr
Keep a database of `(realm, uri) -> (user, password)` mappings.

class urllib.request.HTTPPasswordMgrWithDefaultRealm
Keep a database of `(realm, uri) -> (user, password)` mappings. A realm of `None` is considered a catch-all realm, which is searched if no other realm fits.

class urllib.request.HTTPPasswordMgrWithPriorAuth
A variant of *HTTPPasswordMgrWithDefaultRealm* that also has a database of `uri -> is_authenticated` mappings. Can be used by a BasicAuth handler to determine when to send authentication credentials immediately instead of waiting for a `401` response first.

New in version 3.5.

class urllib.request.AbstractBasicAuthHandler(*password_mgr=None*)
This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported. If *passwd_mgr*

also provides `is_authenticated` and `update_authenticated` methods (see *HTTPPasswordMgrWithPriorAuth Objects*), then the handler will use the `is_authenticated` result for a given URI to determine whether or not to send authentication credentials with the request. If `is_authenticated` returns `True` for the URI, credentials are sent. If `is_authenticated` is `False`, credentials are not sent, and then if a `401` response is received the request is re-sent with the authentication credentials. If authentication succeeds, `update_authenticated` is called to set `is_authenticated True` for the URI, so that subsequent requests to the URI or any of its super-URIs will automatically include the authentication credentials.

New in version 3.5: Added `is_authenticated` support.

class urllib.request.HTTPBasicAuthHandler(*password_mgr=None*)
Handle authentication with the remote host. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported. HTTPBasicAuthHandler will raise a *ValueError* when presented with a wrong Authentication scheme.

class urllib.request.ProxyBasicAuthHandler(*password_mgr=None*)
Handle authentication with the proxy. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported.

class urllib.request.AbstractDigestAuthHandler(*password_mgr=None*)
This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported.

class urllib.request.HTTPDigestAuthHandler(*password_mgr=None*)
Handle authentication with the remote host. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported. When both Digest Authentication Handler and Basic Authentication Handler are both added, Digest Authentication is always tried first. If the Digest Authentication returns a 40x response again, it is sent to Basic Authentication handler to Handle. This Handler method will raise a *ValueError* when presented with an authentication scheme other than Digest or Basic.

Changed in version 3.3: Raise *ValueError* on unsupported Authentication Scheme.

class urllib.request.ProxyDigestAuthHandler(*password_mgr=None*)
Handle authentication with the proxy. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported.

class urllib.request.HTTPHandler
A class to handle opening of HTTP URLs.

class urllib.request.HTTPSHandler(*debuglevel=0*, *context=None*, *check_hostname=None*)
A class to handle opening of HTTPS URLs. *context* and *check_hostname* have the same meaning as in *http.client.HTTPSConnection*.

Changed in version 3.2: *context* and *check_hostname* were added.

class urllib.request.FileHandler
Open local files.

class urllib.request.DataHandler
Open data URLs.

New in version 3.4.

class urllib.request.FTPHandler
Open FTP URLs.

class urllib.request.CacheFTPHandler
Open FTP URLs, keeping a cache of open FTP connections to minimize delays.

class urllib.request.UnknownHandler
A catch-all class to handle unknown URLs.

class urllib.request.HTTPErrorProcessor
Process HTTP error responses.

22.6.1 Request Objects

The following methods describe *Request*'s public interface, and so all may be overridden in subclasses. It also defines several public attributes that can be used by clients to inspect the parsed request.

Request.full_url
The original URL passed to the constructor.

Changed in version 3.4.

Request.full_url is a property with setter, getter and a deleter. Getting *full_url* returns the original request URL with the fragment, if it was present.

Request.type
The URI scheme.

Request.host
The URI authority, typically a host, but may also contain a port separated by a colon.

Request.origin_req_host
The original host for the request, without port.

Request.selector
The URI path. If the *Request* uses a proxy, then selector will be the full URL that is passed to the proxy.

Request.data
The entity body for the request, or None if not specified.

Changed in version 3.4: Changing value of *Request.data* now deletes "Content-Length" header if it was previously set or calculated.

Request.unverifiable
boolean, indicates whether the request is unverifiable as defined by **RFC 2965**.

Request.method
The HTTP request method to use. By default its value is *None*, which means that *get_method()* will do its normal computation of the method to be used. Its value can be set (thus overriding the default computation in *get_method()*) either by providing a default value by setting it at the class level in a *Request* subclass, or by passing a value in to the *Request* constructor via the *method* argument.

New in version 3.3.

Changed in version 3.4: A default value can now be set in subclasses; previously it could only be set via the constructor argument.

Request.get_method()
Return a string indicating the HTTP request method. If *Request.method* is not None, return its value, otherwise return 'GET' if *Request.data* is None, or 'POST' if it's not. This is only meaningful for HTTP requests.

Changed in version 3.3: get_method now looks at the value of *Request.method*.

`Request.add_header(`*key*, *val*`)`
Add another header to the request. Headers are currently ignored by all handlers except HTTP handlers, where they are added to the list of headers sent to the server. Note that there cannot be more than one header with the same name, and later calls will overwrite previous calls in case the *key* collides. Currently, this is no loss of HTTP functionality, since all headers which have meaning when used more than once have a (header-specific) way of gaining the same functionality using only one header.

`Request.add_unredirected_header(`*key*, *header*`)`
Add a header that will not be added to a redirected request.

`Request.has_header(`*header*`)`
Return whether the instance has the named header (checks both regular and unredirected).

`Request.remove_header(`*header*`)`
Remove named header from the request instance (both from regular and unredirected headers).

New in version 3.4.

`Request.get_full_url()`
Return the URL given in the constructor.

Changed in version 3.4.

Returns *Request.full_url*

`Request.set_proxy(`*host*, *type*`)`
Prepare the request by connecting to a proxy server. The *host* and *type* will replace those of the instance, and the instance's selector will be the original URL given in the constructor.

`Request.get_header(`*header_name*, *default=None*`)`
Return the value of the given header. If the header is not present, return the default value.

`Request.header_items()`
Return a list of tuples (header_name, header_value) of the Request headers.

Changed in version 3.4: The request methods add_data, has_data, get_data, get_type, get_host, get_selector, get_origin_req_host and is_unverifiable that were deprecated since 3.3 have been removed.

22.6.2 OpenerDirector Objects

OpenerDirector instances have the following methods:

`OpenerDirector.add_handler(`*handler*`)`
handler should be an instance of *BaseHandler*. The following methods are searched, and added to the possible chains (note that HTTP errors are a special case).

- `protocol_open()` — signal that the handler knows how to open *protocol* URLs.
- `http_error_type()` — signal that the handler knows how to handle HTTP errors with HTTP error code *type*.
- `protocol_error()` — signal that the handler knows how to handle errors from (non-`http`) *protocol*.
- `protocol_request()` — signal that the handler knows how to pre-process *protocol* requests.
- `protocol_response()` — signal that the handler knows how to post-process *protocol* responses.

`OpenerDirector.open(`*url*, *data=None*[, *timeout*]`)`
Open the given *url* (which can be a request object or a string), optionally passing the given *data*. Arguments, return values and exceptions raised are the same as those of *urlopen()* (which simply calls the *open()* method on the currently installed global *OpenerDirector*). The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not

specified, the global default timeout setting will be used). The timeout feature actually works only for HTTP, HTTPS and FTP connections).

`OpenerDirector.error(`*proto*, **args*`)`
Handle an error of the given protocol. This will call the registered error handlers for the given protocol with the given arguments (which are protocol specific). The HTTP protocol is a special case which uses the HTTP response code to determine the specific error handler; refer to the `http_error_*()` methods of the handler classes.

Return values and exceptions raised are the same as those of *urlopen()*.

OpenerDirector objects open URLs in three stages:

The order in which these methods are called within each stage is determined by sorting the handler instances.

1. Every handler with a method named like `protocol_request()` has that method called to pre-process the request.
2. Handlers with a method named like `protocol_open()` are called to handle the request. This stage ends when a handler either returns a non-*None* value (ie. a response), or raises an exception (usually *URLError*). Exceptions are allowed to propagate.

   In fact, the above algorithm is first tried for methods named `default_open()`. If all such methods return *None*, the algorithm is repeated for methods named like `protocol_open()`. If all such methods return *None*, the algorithm is repeated for methods named `unknown_open()`.

   Note that the implementation of these methods may involve calls of the parent *OpenerDirector* instance's *open()* and *error()* methods.
3. Every handler with a method named like `protocol_response()` has that method called to post-process the response.

22.6.3 BaseHandler Objects

BaseHandler objects provide a couple of methods that are directly useful, and others that are meant to be used by derived classes. These are intended for direct use:

`BaseHandler.add_parent(`*director*`)`
Add a director as parent.

`BaseHandler.close()`
Remove any parents.

The following attribute and methods should only be used by classes derived from *BaseHandler*.

Note: The convention has been adopted that subclasses defining `protocol_request()` or `protocol_response()` methods are named `*Processor`; all others are named `*Handler`.

`BaseHandler.parent`
A valid *OpenerDirector*, which can be used to open using a different protocol, or handle errors.

`BaseHandler.default_open(`*req*`)`
This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to catch all URLs.

This method, if implemented, will be called by the parent *OpenerDirector*. It should return a file-like object as described in the return value of the *open()* of *OpenerDirector*, or `None`. It should raise *URLError*, unless a truly exceptional thing happens (for example, *MemoryError* should not be mapped to `URLError`).

This method will be called before any protocol-specific open method.

`BaseHandler.protocol_open(req)`
This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to handle URLs with the given protocol.

This method, if defined, will be called by the parent *OpenerDirector*. Return values should be the same as for *default_open()*.

`BaseHandler.unknown_open(req)`
This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to catch all URLs with no specific registered handler to open it.

This method, if implemented, will be called by the *parent OpenerDirector*. Return values should be the same as for *default_open()*.

`BaseHandler.http_error_default(req, fp, code, msg, hdrs)`
This method is *not* defined in *BaseHandler*, but subclasses should override it if they intend to provide a catch-all for otherwise unhandled HTTP errors. It will be called automatically by the *OpenerDirector* getting the error, and should not normally be called in other circumstances.

req will be a *Request* object, *fp* will be a file-like object with the HTTP error body, *code* will be the three-digit code of the error, *msg* will be the user-visible explanation of the code and *hdrs* will be a mapping object with the headers of the error.

Return values and exceptions raised should be the same as those of *urlopen()*.

`BaseHandler.http_error_nnn(req, fp, code, msg, hdrs)`
nnn should be a three-digit HTTP error code. This method is also not defined in *BaseHandler*, but will be called, if it exists, on an instance of a subclass, when an HTTP error with code *nnn* occurs.

Subclasses should override this method to handle specific HTTP errors.

Arguments, return values and exceptions raised should be the same as for *http_error_default()*.

`BaseHandler.protocol_request(req)`
This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to pre-process requests of the given protocol.

This method, if defined, will be called by the parent *OpenerDirector*. *req* will be a *Request* object. The return value should be a *Request* object.

`BaseHandler.protocol_response(req, response)`
This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to post-process responses of the given protocol.

This method, if defined, will be called by the parent *OpenerDirector*. *req* will be a *Request* object. *response* will be an object implementing the same interface as the return value of *urlopen()*. The return value should implement the same interface as the return value of *urlopen()*.

22.6.4 HTTPRedirectHandler Objects

Note: Some HTTP redirections require action from this module's client code. If this is the case, *HTTPError* is raised. See **RFC 2616** for details of the precise meanings of the various redirection codes.

An `HTTPError` exception raised as a security consideration if the HTTPRedirectHandler is presented with a redirected URL which is not an HTTP, HTTPS or FTP URL.

`HTTPRedirectHandler.redirect_request(req, fp, code, msg, hdrs, newurl)`
Return a *Request* or `None` in response to a redirect. This is called by the default implementations of the `http_error_30*()` methods when a redirection is received from the server. If a redirection should take place, return a new *Request* to allow `http_error_30*()` to perform the redirect to *newurl*. Otherwise,

raise *HTTPError* if no other handler should try to handle this URL, or return `None` if you can't but another handler might.

Note: The default implementation of this method does not strictly follow **RFC 2616**, which says that 301 and 302 responses to `POST` requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a `GET`, and the default implementation reproduces this behavior.

`HTTPRedirectHandler.http_error_301`(*req*, *fp*, *code*, *msg*, *hdrs*)
Redirect to the `Location:` or `URI:` URL. This method is called by the parent *OpenerDirector* when getting an HTTP 'moved permanently' response.

`HTTPRedirectHandler.http_error_302`(*req*, *fp*, *code*, *msg*, *hdrs*)
The same as *http_error_301()*, but called for the 'found' response.

`HTTPRedirectHandler.http_error_303`(*req*, *fp*, *code*, *msg*, *hdrs*)
The same as *http_error_301()*, but called for the 'see other' response.

`HTTPRedirectHandler.http_error_307`(*req*, *fp*, *code*, *msg*, *hdrs*)
The same as *http_error_301()*, but called for the 'temporary redirect' response.

22.6.5 HTTPCookieProcessor Objects

HTTPCookieProcessor instances have one attribute:

`HTTPCookieProcessor.cookiejar`
The *http.cookiejar.CookieJar* in which cookies are stored.

22.6.6 ProxyHandler Objects

`ProxyHandler.protocol_open`(*request*)
The *ProxyHandler* will have a method `protocol_open()` for every *protocol* which has a proxy in the *proxies* dictionary given in the constructor. The method will modify requests to go through the proxy, by calling `request.set_proxy()`, and call the next handler in the chain to actually execute the protocol.

22.6.7 HTTPPasswordMgr Objects

These methods are available on *HTTPPasswordMgr* and *HTTPPasswordMgrWithDefaultRealm* objects.

`HTTPPasswordMgr.add_password`(*realm*, *uri*, *user*, *passwd*)
uri can be either a single URI, or a sequence of URIs. *realm*, *user* and *passwd* must be strings. This causes `(user, passwd)` to be used as authentication tokens when authentication for *realm* and a super-URI of any of the given URIs is given.

`HTTPPasswordMgr.find_user_password`(*realm*, *authuri*)
Get user/password for given realm and URI, if any. This method will return `(None, None)` if there is no matching user/password.

For *HTTPPasswordMgrWithDefaultRealm* objects, the realm `None` will be searched if the given *realm* has no matching user/password.

22.6.8 HTTPPasswordMgrWithPriorAuth Objects

This password manager extends *HTTPPasswordMgrWithDefaultRealm* to support tracking URIs for which authentication credentials should always be sent.

HTTPPasswordMgrWithPriorAuth.**add_password**(*realm*, *uri*, *user*, *passwd*, *is_authenticated=False*)
realm, *uri*, *user*, *passwd* are as for *HTTPPasswordMgr.add_password()*. *is_authenticated* sets the initial value of the `is_authenticated` flag for the given URI or list of URIs. If *is_authenticated* is specified as `True`, *realm* is ignored.

HTTPPasswordMgr.**find_user_password**(*realm*, *authuri*)
Same as for *HTTPPasswordMgrWithDefaultRealm* objects

HTTPPasswordMgrWithPriorAuth.**update_authenticated**(*self*, *uri*, *is_authenticated=False*)
Update the `is_authenticated` flag for the given *uri* or list of URIs.

HTTPPasswordMgrWithPriorAuth.**is_authenticated**(*self*, *authuri*)
Returns the current state of the `is_authenticated` flag for the given URI.

22.6.9 AbstractBasicAuthHandler Objects

AbstractBasicAuthHandler.**http_error_auth_reqed**(*authreq*, *host*, *req*, *headers*)
Handle an authentication request by getting a user/password pair, and re-trying the request. *authreq* should be the name of the header where the information about the realm is included in the request, *host* specifies the URL and path to authenticate for, *req* should be the (failed) *Request* object, and *headers* should be the error headers.

host is either an authority (e.g. `"python.org"`) or a URL containing an authority component (e.g. `"http://python.org/"`). In either case, the authority must not contain a userinfo component (so, `"python.org"` and `"python.org:80"` are fine, `"joe:password@python.org"` is not).

22.6.10 HTTPBasicAuthHandler Objects

HTTPBasicAuthHandler.**http_error_401**(*req*, *fp*, *code*, *msg*, *hdrs*)
Retry the request with authentication information, if available.

22.6.11 ProxyBasicAuthHandler Objects

ProxyBasicAuthHandler.**http_error_407**(*req*, *fp*, *code*, *msg*, *hdrs*)
Retry the request with authentication information, if available.

22.6.12 AbstractDigestAuthHandler Objects

AbstractDigestAuthHandler.**http_error_auth_reqed**(*authreq*, *host*, *req*, *headers*)
authreq should be the name of the header where the information about the realm is included in the request, *host* should be the host to authenticate to, *req* should be the (failed) *Request* object, and *headers* should be the error headers.

22.6.13 HTTPDigestAuthHandler Objects

HTTPDigestAuthHandler.**http_error_401**(*req*, *fp*, *code*, *msg*, *hdrs*)
Retry the request with authentication information, if available.

22.6.14 ProxyDigestAuthHandler Objects

ProxyDigestAuthHandler.**http_error_407**(*req*, *fp*, *code*, *msg*, *hdrs*)
Retry the request with authentication information, if available.

22.6.15 HTTPHandler Objects

HTTPHandler.**http_open**(*req*)
Send an HTTP request, which can be either GET or POST, depending on `req.has_data()`.

22.6.16 HTTPSHandler Objects

HTTPSHandler.**https_open**(*req*)
Send an HTTPS request, which can be either GET or POST, depending on `req.has_data()`.

22.6.17 FileHandler Objects

FileHandler.**file_open**(*req*)
Open the file locally, if there is no host name, or the host name is `'localhost'`.

Changed in version 3.2: This method is applicable only for local hostnames. When a remote hostname is given, an *URLError* is raised.

22.6.18 DataHandler Objects

DataHandler.**data_open**(*req*)
Read a data URL. This kind of URL contains the content encoded in the URL itself. The data URL syntax is specified in **RFC 2397**. This implementation ignores white spaces in base64 encoded data URLs so the URL may be wrapped in whatever source file it comes from. But even though some browsers don't mind about a missing padding at the end of a base64 encoded data URL, this implementation will raise an *ValueError* in that case.

22.6.19 FTPHandler Objects

FTPHandler.**ftp_open**(*req*)
Open the FTP file indicated by *req*. The login is always done with empty username and password.

22.6.20 CacheFTPHandler Objects

CacheFTPHandler objects are *FTPHandler* objects with the following additional methods:

CacheFTPHandler.**setTimeout**(*t*)
Set timeout of connections to *t* seconds.

CacheFTPHandler.**setMaxConns**(*m*)
Set maximum number of cached connections to *m*.

22.6.21 UnknownHandler Objects

UnknownHandler.**unknown_open**()
Raise a *URLError* exception.

22.6.22 HTTPErrorProcessor Objects

`HTTPErrorProcessor.http_response(request, response)`
Process HTTP error responses.

For 200 error codes, the response object is returned immediately.

For non-200 error codes, this simply passes the job on to the `protocol_error_code()` handler methods, via *OpenerDirector.error()*. Eventually, *HTTPDefaultErrorHandler* will raise an *HTTPError* if no other handler handles the error.

`HTTPErrorProcessor.https_response(request, response)`
Process HTTPS error responses.

The behavior is same as *http_response()*.

22.6.23 Examples

In addition to the examples below, more examples are given in urllib-howto.

This example gets the python.org main page and displays the first 300 bytes of it.

```
>>> import urllib.request
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(300))
...
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">\n\n\n<html
xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">\n\n<head>\n
<meta http-equiv="content-type" content="text/html; charset=utf-8" />\n
<title>Python Programming '
```

Note that urlopen returns a bytes object. This is because there is no way for urlopen to automatically determine the encoding of the byte stream it receives from the HTTP server. In general, a program will decode the returned bytes object to string once it determines or guesses the appropriate encoding.

The following W3C document, https://www.w3.org/International/O-charset, lists the various ways in which an (X)HTML or an XML document could have specified its encoding information.

As the python.org website uses *utf-8* encoding as specified in its meta tag, we will use the same for decoding the bytes object.

```
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(100).decode('utf-8'))
...
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtm
```

It is also possible to achieve the same result without using the *context manager* approach.

```
>>> import urllib.request
>>> f = urllib.request.urlopen('http://www.python.org/')
>>> print(f.read(100).decode('utf-8'))
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtm
```

In the following example, we are sending a data-stream to the stdin of a CGI and reading the data it returns to us. Note that this example will only work when the Python installation supports SSL.

```
>>> import urllib.request
>>> req = urllib.request.Request(url='https://localhost/cgi-bin/test.cgi',
...                       data=b'This data is passed to stdin of the CGI')
>>> with urllib.request.urlopen(req) as f:
...     print(f.read().decode('utf-8'))
...
Got Data: "This data is passed to stdin of the CGI"
```

The code for the sample CGI used in the above example is:

```
#!/usr/bin/env python
import sys
data = sys.stdin.read()
print('Content-type: text/plain\n\nGot Data: "%s"' % data)
```

Here is an example of doing a `PUT` request using *Request*:

```
import urllib.request
DATA = b'some data'
req = urllib.request.Request(url='http://localhost:8080', data=DATA,method='PUT')
with urllib.request.urlopen(req) as f:
    pass
print(f.status)
print(f.reason)
```

Use of Basic HTTP Authentication:

```
import urllib.request
# Create an OpenerDirector with support for Basic HTTP Authentication...
auth_handler = urllib.request.HTTPBasicAuthHandler()
auth_handler.add_password(realm='PDQ Application',
                          uri='https://mahler:8092/site-updates.py',
                          user='klem',
                          passwd='kadidd!ehopper')
opener = urllib.request.build_opener(auth_handler)
# ...and install it globally so it can be used with urlopen.
urllib.request.install_opener(opener)
urllib.request.urlopen('http://www.example.com/login.html')
```

build_opener() provides many handlers by default, including a *ProxyHandler*. By default, *ProxyHandler* uses the environment variables named `<scheme>_proxy`, where `<scheme>` is the URL scheme involved. For example, the `http_proxy` environment variable is read to obtain the HTTP proxy's URL.

This example replaces the default *ProxyHandler* with one that uses programmatically-supplied proxy URLs, and adds proxy authorization support with *ProxyBasicAuthHandler*.

```
proxy_handler = urllib.request.ProxyHandler({'http': 'http://www.example.com:3128/'})
proxy_auth_handler = urllib.request.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')

opener = urllib.request.build_opener(proxy_handler, proxy_auth_handler)
# This time, rather than install the OpenerDirector, we use it directly:
opener.open('http://www.example.com/login.html')
```

Adding HTTP headers:

Use the *headers* argument to the *Request* constructor, or:

```
import urllib.request
req = urllib.request.Request('http://www.example.com/')
req.add_header('Referer', 'http://www.python.org/')
# Customize the default User-Agent header value:
req.add_header('User-Agent', 'urllib-example/0.1 (Contact: . . .)')
r = urllib.request.urlopen(req)
```

OpenerDirector automatically adds a *User-Agent* header to every *Request*. To change this:

```
import urllib.request
opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
opener.open('http://www.example.com/')
```

Also, remember that a few standard headers (*Content-Length*, *Content-Type* and *Host*) are added when the *Request* is passed to *urlopen()* (or *OpenerDirector.open()*).

Here is an example session that uses the `GET` method to retrieve a URL containing parameters:

```
>>> import urllib.request
>>> import urllib.parse
>>> params = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> url = "http://www.musi-cal.com/cgi-bin/query?%s" % params
>>> with urllib.request.urlopen(url) as f:
...     print(f.read().decode('utf-8'))
...
```

The following example uses the `POST` method instead. Note that params output from urlencode is encoded to bytes before it is sent to urlopen as data:

```
>>> import urllib.request
>>> import urllib.parse
>>> data = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> data = data.encode('ascii')
>>> with urllib.request.urlopen("http://requestb.in/xrbl82xr", data) as f:
...     print(f.read().decode('utf-8'))
...
```

The following example uses an explicitly specified HTTP proxy, overriding environment settings:

```
>>> import urllib.request
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
>>> opener = urllib.request.FancyURLopener(proxies)
>>> with opener.open("http://www.python.org") as f:
...     f.read().decode('utf-8')
...
```

The following example uses no proxies at all, overriding environment settings:

```
>>> import urllib.request
>>> opener = urllib.request.FancyURLopener({})
>>> with opener.open("http://www.python.org/") as f:
...     f.read().decode('utf-8')
...
```

22.6.24 Legacy interface

The following functions and classes are ported from the Python 2 module `urllib` (as opposed to `urllib2`). They might become deprecated at some point in the future.

`urllib.request.urlretrieve(`*url*, *filename=None*, *reporthook=None*, *data=None*`)`

Copy a network object denoted by a URL to a local file. If the URL points to a local file, the object will not be copied unless filename is supplied. Return a tuple `(filename, headers)` where *filename* is the local file name under which the object can be found, and *headers* is whatever the `info()` method of the object returned by *urlopen()* returned (for a remote object). Exceptions are the same as for *urlopen()*.

The second argument, if present, specifies the file location to copy to (if absent, the location will be a tempfile with a generated name). The third argument, if present, is a callable that will be called once on establishment of the network connection and once after each block read thereafter. The callable will be passed three arguments; a count of blocks transferred so far, a block size in bytes, and the total size of the file. The third argument may be `-1` on older FTP servers which do not return a file size in response to a retrieval request.

The following example illustrates the most common usage scenario:

```
>>> import urllib.request
>>> local_filename, headers = urllib.request.urlretrieve('http://python.org/')
>>> html = open(local_filename)
>>> html.close()
```

If the *url* uses the `http:` scheme identifier, the optional *data* argument may be given to specify a `POST` request (normally the request type is `GET`). The *data* argument must be a bytes object in standard *application/x-www-form-urlencoded* format; see the *urllib.parse.urlencode()* function.

urlretrieve() will raise `ContentTooShortError` when it detects that the amount of data available was less than the expected amount (which is the size reported by a *Content-Length* header). This can occur, for example, when the download is interrupted.

The *Content-Length* is treated as a lower bound: if there's more data to read, urlretrieve reads more data, but if less data is available, it raises the exception.

You can still retrieve the downloaded data in this case, it is stored in the `content` attribute of the exception instance.

If no *Content-Length* header was supplied, urlretrieve can not check the size of the data it has downloaded, and just returns it. In this case you just have to assume that the download was successful.

`urllib.request.urlcleanup()`

Cleans up temporary files that may have been left behind by previous calls to *urlretrieve()*.

`class urllib.request.URLopener(`*proxies=None*, ***x509*`)`

Deprecated since version 3.3.

Base class for opening and reading URLs. Unless you need to support opening objects using schemes other than `http:`, `ftp:`, or `file:`, you probably want to use *FancyURLopener*.

By default, the *URLopener* class sends a *User-Agent* header of `urllib/VVV`, where *VVV* is the *urllib* version number. Applications can define their own *User-Agent* header by subclassing *URLopener* or *FancyURLopener* and setting the class attribute *version* to an appropriate string value in the subclass definition.

The optional *proxies* parameter should be a dictionary mapping scheme names to proxy URLs, where an empty dictionary turns proxies off completely. Its default value is `None`, in which case environmental proxy settings will be used if present, as discussed in the definition of *urlopen()*, above.

Additional keyword parameters, collected in *x509*, may be used for authentication of the client when using the `https:` scheme. The keywords *key_file* and *cert_file* are supported to provide an SSL key and certificate; both are needed to support client authentication.

URLopener objects will raise an *OSError* exception if the server returns an error code.

open(*fullurl*, *data=None*)
Open *fullurl* using the appropriate protocol. This method sets up cache and proxy information, then calls the appropriate open method with its input arguments. If the scheme is not recognized, *open_unknown()* is called. The *data* argument has the same meaning as the *data* argument of *urlopen()*.

open_unknown(*fullurl*, *data=None*)
Overridable interface to open unknown URL types.

retrieve(*url*, *filename=None*, *reporthook=None*, *data=None*)
Retrieves the contents of *url* and places it in *filename*. The return value is a tuple consisting of a local filename and either an *email.message.Message* object containing the response headers (for remote URLs) or `None` (for local URLs). The caller must then open and read the contents of *filename*. If *filename* is not given and the URL refers to a local file, the input filename is returned. If the URL is non-local and *filename* is not given, the filename is the output of *tempfile.mktemp()* with a suffix that matches the suffix of the last path component of the input URL. If *reporthook* is given, it must be a function accepting three numeric parameters: A chunk number, the maximum size chunks are read in and the total size of the download (-1 if unknown). It will be called once at the start and after each chunk of data is read from the network. *reporthook* is ignored for local URLs.

If the *url* uses the `http:` scheme identifier, the optional *data* argument may be given to specify a `POST` request (normally the request type is `GET`). The *data* argument must in standard *application/x-www-form-urlencoded* format; see the *urllib.parse.urlencode()* function.

version
Variable that specifies the user agent of the opener object. To get *urllib* to tell servers that it is a particular user agent, set this in a subclass as a class variable or in the constructor before calling the base constructor.

class urllib.request.FancyURLopener(...)
Deprecated since version 3.3.

FancyURLopener subclasses *URLopener* providing default handling for the following HTTP response codes: 301, 302, 303, 307 and 401. For the 30x response codes listed above, the *Location* header is used to fetch the actual URL. For 401 response codes (authentication required), basic HTTP authentication is performed. For the 30x response codes, recursion is bounded by the value of the *maxtries* attribute, which defaults to 10.

For all other response codes, the method `http_error_default()` is called which you can override in subclasses to handle the error appropriately.

Note: According to the letter of **RFC 2616**, 301 and 302 responses to POST requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a GET, and *urllib* reproduces this behaviour.

The parameters to the constructor are the same as those for *URLopener*.

Note: When performing basic authentication, a *FancyURLopener* instance calls its *prompt_user_passwd()* method. The default implementation asks the users for the required information on the controlling terminal. A subclass may override this method to support more appropriate

behavior if needed.

The *FancyURLopener* class offers one additional method that should be overloaded to provide the appropriate behavior:

prompt_user_passwd(*host*, *realm*)
Return information needed to authenticate the user at the given host in the specified security realm. The return value should be a tuple, `(user, password)`, which can be used for basic authentication.

The implementation prompts for this information on the terminal; an application should override this method to use an appropriate interaction model in the local environment.

22.6.25 `urllib.request` Restrictions

- Currently, only the following protocols are supported: HTTP (versions 0.9 and 1.0), FTP, local files, and data URLs.

  Changed in version 3.4: Added support for data URLs.
- The caching feature of *urlretrieve()* has been disabled until someone finds the time to hack proper processing of Expiration time headers.
- There should be a function to query whether a particular URL is in the cache.
- For backward compatibility, if a URL appears to point to a local file but the file can't be opened, the URL is re-interpreted using the FTP protocol. This can sometimes cause confusing error messages.
- The *urlopen()* and *urlretrieve()* functions can cause arbitrarily long delays while waiting for a network connection to be set up. This means that it is difficult to build an interactive Web client using these functions without using threads.
- The data returned by *urlopen()* or *urlretrieve()* is the raw data returned by the server. This may be binary data (such as an image), plain text or (for example) HTML. The HTTP protocol provides type information in the reply header, which can be inspected by looking at the *Content-Type* header. If the returned data is HTML, you can use the module *html.parser* to parse it.
- The code handling the FTP protocol cannot differentiate between a file and a directory. This can lead to unexpected behavior when attempting to read a URL that points to a file that is not accessible. If the URL ends in a `/`, it is assumed to refer to a directory and will be handled accordingly. But if an attempt to read a file leads to a 550 error (meaning the URL cannot be found or is not accessible, often for permission reasons), then the path is treated as a directory in order to handle the case when a directory is specified by a URL but the trailing `/` has been left off. This can cause misleading results when you try to fetch a file whose read permissions make it inaccessible; the FTP code will try to read it, fail with a 550 error, and then perform a directory listing for the unreadable file. If fine-grained control is needed, consider using the *ftplib* module, subclassing *FancyURLopener*, or changing *_urlopener* to meet your needs.

22.7 `urllib.response` — Response classes used by urllib

The *urllib.response* module defines functions and classes which define a minimal file like interface, including `read()` and `readline()`. The typical response object is an addinfourl instance, which defines an `info()` method and that returns headers and a `geturl()` method that returns the url. Functions defined by this module are used internally by the *urllib.request* module.

22.8 urllib.parse — Parse URLs into components

Source code: Lib/urllib/parse.py

This module defines a standard interface to break Uniform Resource Locator (URL) strings up in components (addressing scheme, network location, path etc.), to combine the components back into a URL string, and to convert a "relative URL" to an absolute URL given a "base URL."

The module has been designed to match the Internet RFC on Relative Uniform Resource Locators. It supports the following URL schemes: `file`, `ftp`, `gopher`, `hdl`, `http`, `https`, `imap`, `mailto`, `mms`, `news`, `nntp`, `prospero`, `rsync`, `rtsp`, `rtspu`, `sftp`, `shttp`, `sip`, `sips`, `snews`, `svn`, `svn+ssh`, `telnet`, `wais`, `ws`, `wss`.

The *urllib.parse* module defines functions that fall into two broad categories: URL parsing and URL quoting. These are covered in detail in the following sections.

22.8.1 URL Parsing

The URL parsing functions focus on splitting a URL string into its components, or on combining URL components into a URL string.

urllib.parse.urlparse(*urlstring*, *scheme=''*, *allow_fragments=True*)

Parse a URL into six components, returning a 6-tuple. This corresponds to the general structure of a URL: `scheme://netloc/path;parameters?query#fragment`. Each tuple item is a string, possibly empty. The components are not broken up in smaller parts (for example, the network location is a single string), and % escapes are not expanded. The delimiters as shown above are not part of the result, except for a leading slash in the *path* component, which is retained if present. For example:

```
>>> from urllib.parse import urlparse
>>> o = urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')
>>> o   # doctest: +NORMALIZE_WHITESPACE
ParseResult(scheme='http', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> o.scheme
'http'
>>> o.port
80
>>> o.geturl()
'http://www.cwi.nl:80/%7Eguido/Python.html'
```

Following the syntax specifications in **RFC 1808**, urlparse recognizes a netloc only if it is properly introduced by '//'. Otherwise the input is presumed to be a relative URL and thus to start with a path component.

```
>>> from urllib.parse import urlparse
>>> urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('www.cwi.nl/%7Eguido/Python.html')
ParseResult(scheme='', netloc='', path='www.cwi.nl/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('help/Python.html')
ParseResult(scheme='', netloc='', path='help/Python.html', params='',
            query='', fragment='')
```

The *scheme* argument gives the default addressing scheme, to be used only if the URL does not specify one. It should be the same type (text or bytes) as *urlstring*, except that the default value `''` is always allowed, and is automatically converted to `b''` if appropriate.

If the *allow_fragments* argument is false, fragment identifiers are not recognized. Instead, they are parsed as part of the path, parameters or query component, and `fragment` is set to the empty string in the return value.

The return value is actually an instance of a subclass of *tuple*. This class has the following additional read-only convenience attributes:

Attribute	Index	Value	Value if not present
`scheme`	0	URL scheme specifier	*scheme* parameter
`netloc`	1	Network location part	empty string
`path`	2	Hierarchical path	empty string
`params`	3	Parameters for last path element	empty string
`query`	4	Query component	empty string
`fragment`	5	Fragment identifier	empty string
`username`		User name	*None*
`password`		Password	*None*
`hostname`		Host name (lower case)	*None*
`port`		Port number as integer, if present	*None*

Reading the `port` attribute will raise a *ValueError* if an invalid port is specified in the URL. See section *Structured Parse Results* for more information on the result object.

Unmatched square brackets in the `netloc` attribute will raise a *ValueError*.

Changed in version 3.2: Added IPv6 URL parsing capabilities.

Changed in version 3.3: The fragment is now parsed for all URL schemes (unless *allow_fragment* is false), in accordance with **RFC 3986**. Previously, a whitelist of schemes that support fragments existed.

Changed in version 3.6: Out-of-range port numbers now raise *ValueError*, instead of returning *None*.

`urllib.parse.parse_qs`(*qs*, *keep_blank_values=False*, *strict_parsing=False*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*)

Parse a query string given as a string argument (data of type *application/x-www-form-urlencoded*). Data are returned as a dictionary. The dictionary keys are the unique query variable names and the values are lists of values for each name.

The optional argument *keep_blank_values* is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument *strict_parsing* is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a *ValueError* exception.

The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the *bytes.decode()* method.

The optional argument *max_num_fields* is the maximum number of fields to read. If set, then throws a *ValueError* if there are more than *max_num_fields* fields read.

Use the *urllib.parse.urlencode()* function (with the `doseq` parameter set to `True`) to convert such dictionaries into query strings.

Changed in version 3.2: Add *encoding* and *errors* parameters.

Changed in version 3.7.2: Added *max_num_fields* parameter.

urllib.parse.parse_qsl(*qs*, *keep_blank_values=False*, *strict_parsing=False*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*)

Parse a query string given as a string argument (data of type *application/x-www-form-urlencoded*). Data are returned as a list of name, value pairs.

The optional argument *keep_blank_values* is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument *strict_parsing* is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a *ValueError* exception.

The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the *bytes.decode()* method.

The optional argument *max_num_fields* is the maximum number of fields to read. If set, then throws a *ValueError* if there are more than *max_num_fields* fields read.

Use the *urllib.parse.urlencode()* function to convert such lists of pairs into query strings.

Changed in version 3.2: Add *encoding* and *errors* parameters.

Changed in version 3.7.2: Added *max_num_fields* parameter.

urllib.parse.urlunparse(*parts*)

Construct a URL from a tuple as returned by `urlparse()`. The *parts* argument can be any six-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a `?` with an empty query; the RFC states that these are equivalent).

urllib.parse.urlsplit(*urlstring*, *scheme=''*, *allow_fragments=True*)

This is similar to *urlparse()*, but does not split the params from the URL. This should generally be used instead of *urlparse()* if the more recent URL syntax allowing parameters to be applied to each segment of the *path* portion of the URL (see **RFC 2396**) is wanted. A separate function is needed to separate the path segments and parameters. This function returns a 5-tuple: (addressing scheme, network location, path, query, fragment identifier).

The return value is actually an instance of a subclass of *tuple*. This class has the following additional read-only convenience attributes:

Attribute	Index	Value	Value if not present
`scheme`	0	URL scheme specifier	*scheme* parameter
`netloc`	1	Network location part	empty string
`path`	2	Hierarchical path	empty string
`query`	3	Query component	empty string
`fragment`	4	Fragment identifier	empty string
`username`		User name	*None*
`password`		Password	*None*
`hostname`		Host name (lower case)	*None*
`port`		Port number as integer, if present	*None*

Reading the `port` attribute will raise a *ValueError* if an invalid port is specified in the URL. See section *Structured Parse Results* for more information on the result object.

Unmatched square brackets in the `netloc` attribute will raise a *ValueError*.

Changed in version 3.6: Out-of-range port numbers now raise *ValueError*, instead of returning *None*.

urllib.parse.urlunsplit(*parts*)

Combine the elements of a tuple as returned by *urlsplit()* into a complete URL as a string. The

parts argument can be any five-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a ? with an empty query; the RFC states that these are equivalent).

`urllib.parse.urljoin`(*base*, *url*, *allow_fragments=True*)

Construct a full ("absolute") URL by combining a "base URL" (*base*) with another URL (*url*). Informally, this uses components of the base URL, in particular the addressing scheme, the network location and (part of) the path, to provide missing components in the relative URL. For example:

```
>>> from urllib.parse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

The *allow_fragments* argument has the same meaning and default as for *urlparse()*.

Note: If *url* is an absolute URL (that is, starting with `//` or `scheme://`), the *url*'s host name and/or scheme will be present in the result. For example:

```
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html',
...         '//www.python.org/%7Eguido')
'http://www.python.org/%7Eguido'
```

If you do not want that behavior, preprocess the *url* with *urlsplit()* and *urlunsplit()*, removing possible *scheme* and *netloc* parts.

Changed in version 3.5: Behaviour updated to match the semantics defined in **RFC 3986**.

`urllib.parse.urldefrag`(*url*)

If *url* contains a fragment identifier, return a modified version of *url* with no fragment identifier, and the fragment identifier as a separate string. If there is no fragment identifier in *url*, return *url* unmodified and an empty string.

The return value is actually an instance of a subclass of *tuple*. This class has the following additional read-only convenience attributes:

Attribute	Index	Value	Value if not present
`url`	0	URL with no fragment	empty string
`fragment`	1	Fragment identifier	empty string

See section *Structured Parse Results* for more information on the result object.

Changed in version 3.2: Result is a structured object rather than a simple 2-tuple.

22.8.2 Parsing ASCII Encoded Bytes

The URL parsing functions were originally designed to operate on character strings only. In practice, it is useful to be able to manipulate properly quoted and encoded URLs as sequences of ASCII bytes. Accordingly, the URL parsing functions in this module all operate on *bytes* and *bytearray* objects in addition to *str* objects.

If *str* data is passed in, the result will also contain only *str* data. If *bytes* or *bytearray* data is passed in, the result will contain only *bytes* data.

Attempting to mix *str* data with *bytes* or *bytearray* in a single function call will result in a *TypeError* being raised, while attempting to pass in non-ASCII byte values will trigger *UnicodeDecodeError*.

To support easier conversion of result objects between *str* and *bytes*, all return values from URL parsing functions provide either an `encode()` method (when the result contains *str* data) or a `decode()` method (when the result contains *bytes* data). The signatures of these methods match those of the corresponding *str* and *bytes* methods (except that the default encoding is `'ascii'` rather than `'utf-8'`). Each produces a value of a corresponding type that contains either *bytes* data (for `encode()` methods) or *str* data (for `decode()` methods).

Applications that need to operate on potentially improperly quoted URLs that may contain non-ASCII data will need to do their own decoding from bytes to characters before invoking the URL parsing methods.

The behaviour described in this section applies only to the URL parsing functions. The URL quoting functions use their own rules when producing or consuming byte sequences as detailed in the documentation of the individual URL quoting functions.

Changed in version 3.2: URL parsing functions now accept ASCII encoded byte sequences

22.8.3 Structured Parse Results

The result objects from the *urlparse()*, *urlsplit()* and *urldefrag()* functions are subclasses of the *tuple* type. These subclasses add the attributes listed in the documentation for those functions, the encoding and decoding support described in the previous section, as well as an additional method:

urllib.parse.SplitResult.geturl()

Return the re-combined version of the original URL as a string. This may differ from the original URL in that the scheme may be normalized to lower case and empty components may be dropped. Specifically, empty parameters, queries, and fragment identifiers will be removed.

For *urldefrag()* results, only empty fragment identifiers will be removed. For *urlsplit()* and *urlparse()* results, all noted changes will be made to the URL returned by this method.

The result of this method remains unchanged if passed back through the original parsing function:

```
>>> from urllib.parse import urlsplit
>>> url = 'HTTP://www.Python.org/doc/#'
>>> r1 = urlsplit(url)
>>> r1.geturl()
'http://www.Python.org/doc/'
>>> r2 = urlsplit(r1.geturl())
>>> r2.geturl()
'http://www.Python.org/doc/'
```

The following classes provide the implementations of the structured parse results when operating on *str* objects:

class urllib.parse.DefragResult(*url*, *fragment*)

Concrete class for *urldefrag()* results containing *str* data. The `encode()` method returns a *DefragResultBytes* instance.

New in version 3.2.

class urllib.parse.ParseResult(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)

Concrete class for *urlparse()* results containing *str* data. The `encode()` method returns a *ParseResultBytes* instance.

class urllib.parse.SplitResult(*scheme*, *netloc*, *path*, *query*, *fragment*)

Concrete class for *urlsplit()* results containing *str* data. The `encode()` method returns a *SplitResultBytes* instance.

The following classes provide the implementations of the parse results when operating on *bytes* or *bytearray* objects:

class urllib.parse.DefragResultBytes(*url*, *fragment*)
Concrete class for *urldefrag()* results containing *bytes* data. The decode() method returns a *DefragResult* instance.

New in version 3.2.

class urllib.parse.ParseResultBytes(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)
Concrete class for *urlparse()* results containing *bytes* data. The decode() method returns a *ParseResult* instance.

New in version 3.2.

class urllib.parse.SplitResultBytes(*scheme*, *netloc*, *path*, *query*, *fragment*)
Concrete class for *urlsplit()* results containing *bytes* data. The decode() method returns a *SplitResult* instance.

New in version 3.2.

22.8.4 URL Quoting

The URL quoting functions focus on taking program data and making it safe for use as URL components by quoting special characters and appropriately encoding non-ASCII text. They also support reversing these operations to recreate the original data from the contents of a URL component if that task isn't already covered by the URL parsing functions above.

urllib.parse.quote(*string*, *safe='/'*, *encoding=None*, *errors=None*)
Replace special characters in *string* using the %xx escape. Letters, digits, and the characters '_.-~' are never quoted. By default, this function is intended for quoting the path section of URL. The optional *safe* parameter specifies additional ASCII characters that should not be quoted — its default value is '/'.

string may be either a *str* or a *bytes*.

Changed in version 3.7: Moved from **RFC 2396** to **RFC 3986** for quoting URL strings. "~" is now included in the set of reserved characters.

The optional *encoding* and *errors* parameters specify how to deal with non-ASCII characters, as accepted by the *str.encode()* method. *encoding* defaults to 'utf-8'. *errors* defaults to 'strict', meaning unsupported characters raise a *UnicodeEncodeError*. *encoding* and *errors* must not be supplied if *string* is a *bytes*, or a *TypeError* is raised.

Note that quote(string, safe, encoding, errors) is equivalent to quote_from_bytes(string.encode(encoding, errors), safe).

Example: quote('/El Niño/') yields '/El%20Ni%C3%B1o/'.

urllib.parse.quote_plus(*string*, *safe=''*, *encoding=None*, *errors=None*)
Like *quote()*, but also replace spaces by plus signs, as required for quoting HTML form values when building up a query string to go into a URL. Plus signs in the original string are escaped unless they are included in *safe*. It also does not have *safe* default to '/'.

Example: quote_plus('/El Niño/') yields '%2FEl+Ni%C3%B1o%2F'.

urllib.parse.quote_from_bytes(*bytes*, *safe='/'*)
Like *quote()*, but accepts a *bytes* object rather than a *str*, and does not perform string-to-bytes encoding.

Example: quote_from_bytes(b'a&\xef') yields 'a%26%EF'.

urllib.parse.unquote(*string*, *encoding='utf-8'*, *errors='replace'*)
Replace %xx escapes by their single-character equivalent. The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the *bytes.decode()* method.

string must be a *str*.

encoding defaults to `'utf-8'`. *errors* defaults to `'replace'`, meaning invalid sequences are replaced by a placeholder character.

Example: `unquote('/El%20Ni%C3%B1o/')` yields `'/El Niño/'`.

`urllib.parse.unquote_plus(`*string*, *encoding='utf-8'*, *errors='replace'*`)`
Like *unquote()*, but also replace plus signs by spaces, as required for unquoting HTML form values.

string must be a *str*.

Example: `unquote_plus('/El+Ni%C3%B1o/')` yields `'/El Niño/'`.

`urllib.parse.unquote_to_bytes(`*string*`)`
Replace `%xx` escapes by their single-octet equivalent, and return a *bytes* object.

string may be either a *str* or a *bytes*.

If it is a *str*, unescaped non-ASCII characters in *string* are encoded into UTF-8 bytes.

Example: `unquote_to_bytes('a%26%EF')` yields `b'a&\xef'`.

`urllib.parse.urlencode(`*query*, *doseq=False*, *safe=''*, *encoding=None*, *errors=None*, *quote_via=quote_plus*`)`
Convert a mapping object or a sequence of two-element tuples, which may contain *str* or *bytes* objects, to a percent-encoded ASCII text string. If the resultant string is to be used as a *data* for POST operation with the *urlopen()* function, then it should be encoded to bytes, otherwise it would result in a *TypeError*.

The resulting string is a series of `key=value` pairs separated by `'&'` characters, where both *key* and *value* are quoted using the *quote_via* function. By default, *quote_plus()* is used to quote the values, which means spaces are quoted as a `'+'` character and '/' characters are encoded as `%2F`, which follows the standard for GET requests (`application/x-www-form-urlencoded`). An alternate function that can be passed as *quote_via* is *quote()*, which will encode spaces as `%20` and not encode '/' characters. For maximum control of what is quoted, use `quote` and specify a value for *safe*.

When a sequence of two-element tuples is used as the *query* argument, the first element of each tuple is a key and the second is a value. The value element in itself can be a sequence and in that case, if the optional parameter *doseq* is evaluates to `True`, individual `key=value` pairs separated by `'&'` are generated for each element of the value sequence for the key. The order of parameters in the encoded string will match the order of parameter tuples in the sequence.

The *safe*, *encoding*, and *errors* parameters are passed down to *quote_via* (the *encoding* and *errors* parameters are only passed when a query element is a *str*).

To reverse this encoding process, *parse_qs()* and *parse_qsl()* are provided in this module to parse query strings into Python data structures.

Refer to *urllib examples* to find out how urlencode method can be used for generating query string for a URL or data for POST.

Changed in version 3.2: Query parameter supports bytes and string objects.

New in version 3.5: *quote_via* parameter.

See also:

RFC 3986 - Uniform Resource Identifiers This is the current standard (STD66). Any changes to urllib.parse module should conform to this. Certain deviations could be observed, which are mostly for backward compatibility purposes and for certain de-facto parsing requirements as commonly observed in major browsers.

RFC 2732 - Format for Literal IPv6 Addresses in URL's. This specifies the parsing requirements of IPv6 URLs.

RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax Document describing the generic syntactic requirements for both Uniform Resource Names (URNs) and Uniform Resource Locators (URLs).

RFC 2368 - The mailto URL scheme. Parsing requirements for mailto URL schemes.

RFC 1808 - Relative Uniform Resource Locators This Request For Comments includes the rules for joining an absolute and a relative URL, including a fair number of "Abnormal Examples" which govern the treatment of border cases.

RFC 1738 - Uniform Resource Locators (URL) This specifies the formal syntax and semantics of absolute URLs.

22.9 urllib.error — Exception classes raised by urllib.request

Source code: Lib/urllib/error.py

The *urllib.error* module defines the exception classes for exceptions raised by *urllib.request*. The base exception class is *URLError*.

The following exceptions are raised by *urllib.error* as appropriate:

exception urllib.error.URLError
The handlers raise this exception (or derived exceptions) when they run into a problem. It is a subclass of *OSError*.

reason
The reason for this error. It can be a message string or another exception instance.

Changed in version 3.3: *URLError* has been made a subclass of *OSError* instead of *IOError*.

exception urllib.error.HTTPError
Though being an exception (a subclass of *URLError*), an *HTTPError* can also function as a non-exceptional file-like return value (the same thing that *urlopen()* returns). This is useful when handling exotic HTTP errors, such as requests for authentication.

code
An HTTP status code as defined in **RFC 2616**. This numeric value corresponds to a value found in the dictionary of codes as found in *http.server.BaseHTTPRequestHandler.responses*.

reason
This is usually a string explaining the reason for this error.

headers
The HTTP response headers for the HTTP request that caused the *HTTPError*.

New in version 3.4.

exception urllib.error.ContentTooShortError(*msg*, *content*)
This exception is raised when the *urlretrieve()* function detects that the amount of the downloaded data is less than the expected amount (given by the *Content-Length* header). The `content` attribute stores the downloaded (and supposedly truncated) data.

22.10 urllib.robotparser — Parser for robots.txt

Source code: Lib/urllib/robotparser.py

This module provides a single class, *RobotFileParser*, which answers questions about whether or not a particular user agent can fetch a URL on the Web site that published the `robots.txt` file. For more details on the structure of `robots.txt` files, see http://www.robotstxt.org/orig.html.

class urllib.robotparser.RobotFileParser(*url=''*)

This class provides methods to read, parse and answer questions about the `robots.txt` file at *url*.

set_url(*url*)

Sets the URL referring to a `robots.txt` file.

read()

Reads the `robots.txt` URL and feeds it to the parser.

parse(*lines*)

Parses the lines argument.

can_fetch(*useragent*, *url*)

Returns `True` if the *useragent* is allowed to fetch the *url* according to the rules contained in the parsed `robots.txt` file.

mtime()

Returns the time the `robots.txt` file was last fetched. This is useful for long-running web spiders that need to check for new `robots.txt` files periodically.

modified()

Sets the time the `robots.txt` file was last fetched to the current time.

crawl_delay(*useragent*)

Returns the value of the `Crawl-delay` parameter from `robots.txt` for the *useragent* in question. If there is no such parameter or it doesn't apply to the *useragent* specified or the `robots.txt` entry for this parameter has invalid syntax, return `None`.

New in version 3.6.

request_rate(*useragent*)

Returns the contents of the `Request-rate` parameter from `robots.txt` as a *named tuple* `RequestRate(requests, seconds)`. If there is no such parameter or it doesn't apply to the *useragent* specified or the `robots.txt` entry for this parameter has invalid syntax, return `None`.

New in version 3.6.

The following example demonstrates basic use of the *RobotFileParser* class:

```
>>> import urllib.robotparser
>>> rp = urllib.robotparser.RobotFileParser()
>>> rp.set_url("http://www.musi-cal.com/robots.txt")
>>> rp.read()
>>> rrate = rp.request_rate("*")
>>> rrate.requests
3
>>> rrate.seconds
20
>>> rp.crawl_delay("*")
6
>>> rp.can_fetch("*", "http://www.musi-cal.com/cgi-bin/search?city=San+Francisco")
False
>>> rp.can_fetch("*", "http://www.musi-cal.com/")
True
```

22.11 http — HTTP modules

Source code: Lib/http/__init__.py

---

http is a package that collects several modules for working with the HyperText Transfer Protocol:

- *http.client* is a low-level HTTP protocol client; for high-level URL opening use *urllib.request*
- *http.server* contains basic HTTP server classes based on *socketserver*
- *http.cookies* has utilities for implementing state management with cookies
- *http.cookiejar* provides persistence of cookies

http is also a module that defines a number of HTTP status codes and associated messages through the *http.HTTPStatus* enum:

class http.HTTPStatus

New in version 3.5.

A subclass of *enum.IntEnum* that defines a set of HTTP status codes, reason phrases and long descriptions written in English.

Usage:

```
>>> from http import HTTPStatus
>>> HTTPStatus.OK
<HTTPStatus.OK: 200>
>>> HTTPStatus.OK == 200
True
>>> http.HTTPStatus.OK.value
200
>>> HTTPStatus.OK.phrase
'OK'
>>> HTTPStatus.OK.description
'Request fulfilled, document follows'
>>> list(HTTPStatus)
[<HTTPStatus.CONTINUE: 100>, <HTTPStatus.SWITCHING_PROTOCOLS: 101>, ...]
```

22.11.1 HTTP status codes

Supported, IANA-registered status codes available in *http.HTTPStatus* are:

Code	Enum Name	Details
100	CONTINUE	HTTP/1.1 **RFC 7231**, Section 6.2.1
101	SWITCHING_PROTOCOLS	HTTP/1.1 **RFC 7231**, Section 6.2.2
102	PROCESSING	WebDAV **RFC 2518**, Section 10.1
200	OK	HTTP/1.1 **RFC 7231**, Section 6.3.1
201	CREATED	HTTP/1.1 **RFC 7231**, Section 6.3.2
202	ACCEPTED	HTTP/1.1 **RFC 7231**, Section 6.3.3
203	NON_AUTHORITATIVE_INFORMATION	HTTP/1.1 **RFC 7231**, Section 6.3.4
204	NO_CONTENT	HTTP/1.1 **RFC 7231**, Section 6.3.5
205	RESET_CONTENT	HTTP/1.1 **RFC 7231**, Section 6.3.6
206	PARTIAL_CONTENT	HTTP/1.1 **RFC 7233**, Section 4.1

Continued on

Table 1 – continued from previous page

Code	Enum Name	Details
207	MULTI_STATUS	WebDAV **RFC 4918**, Section 11.1
208	ALREADY_REPORTED	WebDAV Binding Extensions **RFC 5842**, Section 7.1 (Experimental)
226	IM_USED	Delta Encoding in HTTP **RFC 3229**, Section 10.4.1
300	MULTIPLE_CHOICES	HTTP/1.1 **RFC 7231**, Section 6.4.1
301	MOVED_PERMANENTLY	HTTP/1.1 **RFC 7231**, Section 6.4.2
302	FOUND	HTTP/1.1 **RFC 7231**, Section 6.4.3
303	SEE_OTHER	HTTP/1.1 **RFC 7231**, Section 6.4.4
304	NOT_MODIFIED	HTTP/1.1 **RFC 7232**, Section 4.1
305	USE_PROXY	HTTP/1.1 **RFC 7231**, Section 6.4.5
307	TEMPORARY_REDIRECT	HTTP/1.1 **RFC 7231**, Section 6.4.7
308	PERMANENT_REDIRECT	Permanent Redirect **RFC 7238**, Section 3 (Experimental)
400	BAD_REQUEST	HTTP/1.1 **RFC 7231**, Section 6.5.1
401	UNAUTHORIZED	HTTP/1.1 Authentication **RFC 7235**, Section 3.1
402	PAYMENT_REQUIRED	HTTP/1.1 **RFC 7231**, Section 6.5.2
403	FORBIDDEN	HTTP/1.1 **RFC 7231**, Section 6.5.3
404	NOT_FOUND	HTTP/1.1 **RFC 7231**, Section 6.5.4
405	METHOD_NOT_ALLOWED	HTTP/1.1 **RFC 7231**, Section 6.5.5
406	NOT_ACCEPTABLE	HTTP/1.1 **RFC 7231**, Section 6.5.6
407	PROXY_AUTHENTICATION_REQUIRED	HTTP/1.1 Authentication **RFC 7235**, Section 3.2
408	REQUEST_TIMEOUT	HTTP/1.1 **RFC 7231**, Section 6.5.7
409	CONFLICT	HTTP/1.1 **RFC 7231**, Section 6.5.8
410	GONE	HTTP/1.1 **RFC 7231**, Section 6.5.9
411	LENGTH_REQUIRED	HTTP/1.1 **RFC 7231**, Section 6.5.10
412	PRECONDITION_FAILED	HTTP/1.1 **RFC 7232**, Section 4.2
413	REQUEST_ENTITY_TOO_LARGE	HTTP/1.1 **RFC 7231**, Section 6.5.11
414	REQUEST_URI_TOO_LONG	HTTP/1.1 **RFC 7231**, Section 6.5.12
415	UNSUPPORTED_MEDIA_TYPE	HTTP/1.1 **RFC 7231**, Section 6.5.13
416	REQUEST_RANGE_NOT_SATISFIABLE	HTTP/1.1 Range Requests **RFC 7233**, Section 4.4
417	EXPECTATION_FAILED	HTTP/1.1 **RFC 7231**, Section 6.5.14
421	MISDIRECTED_REQUEST	HTTP/2 **RFC 7540**, Section 9.1.2
422	UNPROCESSABLE_ENTITY	WebDAV **RFC 4918**, Section 11.2
423	LOCKED	WebDAV **RFC 4918**, Section 11.3
424	FAILED_DEPENDENCY	WebDAV **RFC 4918**, Section 11.4
426	UPGRADE_REQUIRED	HTTP/1.1 **RFC 7231**, Section 6.5.15
428	PRECONDITION_REQUIRED	Additional HTTP Status Codes **RFC 6585**
429	TOO_MANY_REQUESTS	Additional HTTP Status Codes **RFC 6585**
431	REQUEST_HEADER_FIELDS_TOO_LARGE	Additional HTTP Status Codes **RFC 6585**
500	INTERNAL_SERVER_ERROR	HTTP/1.1 **RFC 7231**, Section 6.6.1
501	NOT_IMPLEMENTED	HTTP/1.1 **RFC 7231**, Section 6.6.2
502	BAD_GATEWAY	HTTP/1.1 **RFC 7231**, Section 6.6.3
503	SERVICE_UNAVAILABLE	HTTP/1.1 **RFC 7231**, Section 6.6.4
504	GATEWAY_TIMEOUT	HTTP/1.1 **RFC 7231**, Section 6.6.5
505	HTTP_VERSION_NOT_SUPPORTED	HTTP/1.1 **RFC 7231**, Section 6.6.6
506	VARIANT_ALSO_NEGOTIATES	Transparent Content Negotiation in HTTP **RFC 2295**, Section 8.1 (Expe
507	INSUFFICIENT_STORAGE	WebDAV **RFC 4918**, Section 11.5
508	LOOP_DETECTED	WebDAV Binding Extensions **RFC 5842**, Section 7.2 (Experimental)
510	NOT_EXTENDED	An HTTP Extension Framework **RFC 2774**, Section 7 (Experimental)
511	NETWORK_AUTHENTICATION_REQUIRED	Additional HTTP Status Codes **RFC 6585**, Section 6

In order to preserve backwards compatibility, enum values are also present in the *http.client* module in

the form of constants. The enum name is equal to the constant name (i.e. `http.HTTPStatus.OK` is also available as `http.client.OK`).

Changed in version 3.7: Added `421 MISDIRECTED_REQUEST` status code.

22.12 `http.client` — HTTP protocol client

Source code: Lib/http/client.py

This module defines classes which implement the client side of the HTTP and HTTPS protocols. It is normally not used directly — the module *`urllib.request`* uses it to handle URLs that use HTTP and HTTPS.

See also:

The Requests package is recommended for a higher-level HTTP client interface.

Note: HTTPS support is only available if Python was compiled with SSL support (through the *`ssl`* module).

The module provides the following classes:

class `http.client.HTTPConnection`(*host*, *port=None*[, *timeout*], *source_address=None*, *blocksize=8192*)

An *`HTTPConnection`* instance represents one transaction with an HTTP server. It should be instantiated passing it a host and optional port number. If no port number is passed, the port is extracted from the host string if it has the form `host:port`, else the default HTTP port (80) is used. If the optional *timeout* parameter is given, blocking operations (like connection attempts) will timeout after that many seconds (if it is not given, the global default timeout setting is used). The optional *source_address* parameter may be a tuple of a (host, port) to use as the source address the HTTP connection is made from. The optional *blocksize* parameter sets the buffer size in bytes for sending a file-like message body.

For example, the following calls all create instances that connect to the server at the same host and port:

```
>>> h1 = http.client.HTTPConnection('www.python.org')
>>> h2 = http.client.HTTPConnection('www.python.org:80')
>>> h3 = http.client.HTTPConnection('www.python.org', 80)
>>> h4 = http.client.HTTPConnection('www.python.org', 80, timeout=10)
```

Changed in version 3.2: *source_address* was added.

Changed in version 3.4: The *strict* parameter was removed. HTTP 0.9-style "Simple Responses" are not longer supported.

Changed in version 3.7: *blocksize* parameter was added.

class `http.client.HTTPSConnection`(*host*, *port=None*, *key_file=None*, *cert_file=None*[, *timeout*], *source_address=None*, ***, *context=None*, *check_hostname=None*, *blocksize=8192*)

A subclass of *`HTTPConnection`* that uses SSL for communication with secure servers. Default port is 443. If *context* is specified, it must be a *`ssl.SSLContext`* instance describing the various SSL options.

Please read *Security considerations* for more information on best practices.

Changed in version 3.2: *source_address*, *context* and *check_hostname* were added.

Changed in version 3.2: This class now supports HTTPS virtual hosts if possible (that is, if *ssl.HAS_SNI* is true).

Changed in version 3.4: The *strict* parameter was removed. HTTP 0.9-style "Simple Responses" are no longer supported.

Changed in version 3.4.3: This class now performs all the necessary certificate and hostname checks by default. To revert to the previous, unverified, behavior `ssl._create_unverified_context()` can be passed to the *context* parameter.

Deprecated since version 3.6: *key_file* and *cert_file* 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.

The *check_hostname* parameter is also deprecated; the *ssl.SSLContext.check_hostname* attribute of *context* should be used instead.

class `http.client.HTTPResponse`(*sock*, *debuglevel=0*, *method=None*, *url=None*)
Class whose instances are returned upon successful connection. Not instantiated directly by user.

Changed in version 3.4: The *strict* parameter was removed. HTTP 0.9 style "Simple Responses" are no longer supported.

The following exceptions are raised as appropriate:

exception `http.client.HTTPException`
The base class of the other exceptions in this module. It is a subclass of *Exception*.

exception `http.client.NotConnected`
A subclass of *HTTPException*.

exception `http.client.InvalidURL`
A subclass of *HTTPException*, raised if a port is given and is either non-numeric or empty.

exception `http.client.UnknownProtocol`
A subclass of *HTTPException*.

exception `http.client.UnknownTransferEncoding`
A subclass of *HTTPException*.

exception `http.client.UnimplementedFileMode`
A subclass of *HTTPException*.

exception `http.client.IncompleteRead`
A subclass of *HTTPException*.

exception `http.client.ImproperConnectionState`
A subclass of *HTTPException*.

exception `http.client.CannotSendRequest`
A subclass of *ImproperConnectionState*.

exception `http.client.CannotSendHeader`
A subclass of *ImproperConnectionState*.

exception `http.client.ResponseNotReady`
A subclass of *ImproperConnectionState*.

exception `http.client.BadStatusLine`
A subclass of *HTTPException*. Raised if a server responds with a HTTP status code that we don't understand.

exception http.client.LineTooLong
A subclass of *HTTPException*. Raised if an excessively long line is received in the HTTP protocol from the server.

exception http.client.RemoteDisconnected
A subclass of *ConnectionResetError* and *BadStatusLine*. Raised by *HTTPConnection.getresponse()* when the attempt to read the response results in no data read from the connection, indicating that the remote end has closed the connection.

New in version 3.5: Previously, *BadStatusLine*`('')` was raised.

The constants defined in this module are:

http.client.HTTP_PORT
The default port for the HTTP protocol (always `80`).

http.client.HTTPS_PORT
The default port for the HTTPS protocol (always `443`).

http.client.responses
This dictionary maps the HTTP 1.1 status codes to the W3C names.

Example: `http.client.responses[http.client.NOT_FOUND]` is `'Not Found'`.

See *HTTP status codes* for a list of HTTP status codes that are available in this module as constants.

22.12.1 HTTPConnection Objects

HTTPConnection instances have the following methods:

HTTPConnection.request(*method*, *url*, *body=None*, *headers={}*, ***, *encode_chunked=False*)
This will send a request to the server using the HTTP request method *method* and the selector *url*.

If *body* is specified, the specified data is sent after the headers are finished. It may be a *str*, a *bytes-like object*, an open *file object*, or an iterable of *bytes*. If *body* is a string, it is encoded as ISO-8859-1, the default for HTTP. If it is a bytes-like object, the bytes are sent as is. If it is a *file object*, the contents of the file is sent; this file object should support at least the `read()` method. If the file object is an instance of *io.TextIOBase*, the data returned by the `read()` method will be encoded as ISO-8859-1, otherwise the data returned by `read()` is sent as is. If *body* is an iterable, the elements of the iterable are sent as is until the iterable is exhausted.

The *headers* argument should be a mapping of extra HTTP headers to send with the request.

If *headers* contains neither Content-Length nor Transfer-Encoding, but there is a request body, one of those header fields will be added automatically. If *body* is `None`, the Content-Length header is set to `0` for methods that expect a body (`PUT`, `POST`, and `PATCH`). If *body* is a string or a bytes-like object that is not also a *file*, the Content-Length header is set to its length. Any other type of *body* (files and iterables in general) will be chunk-encoded, and the Transfer-Encoding header will automatically be set instead of Content-Length.

The *encode_chunked* argument is only relevant if Transfer-Encoding is specified in *headers*. If *encode_chunked* is `False`, the HTTPConnection object assumes that all encoding is handled by the calling code. If it is `True`, the body will be chunk-encoded.

Note: Chunked transfer encoding has been added to the HTTP protocol version 1.1. Unless the HTTP server is known to handle HTTP 1.1, the caller must either specify the Content-Length, or must pass a *str* or bytes-like object that is not also a file as the body representation.

New in version 3.2: *body* can now be an iterable.

Changed in version 3.6: If neither Content-Length nor Transfer-Encoding are set in *headers*, file and iterable *body* objects are now chunk-encoded. The *encode_chunked* argument was added. No attempt is made to determine the Content-Length for file objects.

HTTPConnection.getresponse()
Should be called after a request is sent to get the response from the server. Returns an *HTTPResponse* instance.

Note: Note that you must have read the whole response before you can send a new request to the server.

Changed in version 3.5: If a *ConnectionError* or subclass is raised, the *HTTPConnection* object will be ready to reconnect when a new request is sent.

HTTPConnection.set_debuglevel(*level*)
Set the debugging level. The default debug level is `0`, meaning no debugging output is printed. Any value greater than `0` will cause all currently defined debug output to be printed to stdout. The `debuglevel` is passed to any new *HTTPResponse* objects that are created.

New in version 3.1.

HTTPConnection.set_tunnel(*host*, *port=None*, *headers=None*)
Set the host and the port for HTTP Connect Tunnelling. This allows running the connection through a proxy server.

The host and port arguments specify the endpoint of the tunneled connection (i.e. the address included in the CONNECT request, *not* the address of the proxy server).

The headers argument should be a mapping of extra HTTP headers to send with the CONNECT request.

For example, to tunnel through a HTTPS proxy server running locally on port 8080, we would pass the address of the proxy to the *HTTPSConnection* constructor, and the address of the host that we eventually want to reach to the *set_tunnel()* method:

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("localhost", 8080)
>>> conn.set_tunnel("www.python.org")
>>> conn.request("HEAD","/index.html")
```

New in version 3.2.

HTTPConnection.connect()
Connect to the server specified when the object was created. By default, this is called automatically when making a request if the client does not already have a connection.

HTTPConnection.close()
Close the connection to the server.

HTTPConnection.blocksize
Buffer size in bytes for sending a file-like message body.

New in version 3.7.

As an alternative to using the `request()` method described above, you can also send your request step by step, by using the four functions below.

HTTPConnection.putrequest(*method*, *url*, *skip_host=False*, *skip_accept_encoding=False*)
This should be the first call after the connection to the server has been made. It sends a line to the server consisting of the *method* string, the *url* string, and the HTTP version (`HTTP/1.1`). To disable

automatic sending of `Host:` or `Accept-Encoding:` headers (for example to accept additional content encodings), specify *skip_host* or *skip_accept_encoding* with non-False values.

`HTTPConnection.putheader`(*header*, *argument*[, ...])
Send an **RFC 822**-style header to the server. It sends a line to the server consisting of the header, a colon and a space, and the first argument. If more arguments are given, continuation lines are sent, each consisting of a tab and an argument.

`HTTPConnection.endheaders`(*message_body=None*, *, *encode_chunked=False*)
Send a blank line to the server, signalling the end of the headers. The optional *message_body* argument can be used to pass a message body associated with the request.

If *encode_chunked* is `True`, the result of each iteration of *message_body* will be chunk-encoded as specified in **RFC 7230**, Section 3.3.1. How the data is encoded is dependent on the type of *message_body*. If *message_body* implements the buffer interface the encoding will result in a single chunk. If *message_body* is a *collections.abc.Iterable*, each iteration of *message_body* will result in a chunk. If *message_body* is a *file object*, each call to `.read()` will result in a chunk. The method automatically signals the end of the chunk-encoded data immediately after *message_body*.

Note: Due to the chunked encoding specification, empty chunks yielded by an iterator body will be ignored by the chunk-encoder. This is to avoid premature termination of the read of the request by the target server due to malformed encoding.

New in version 3.6: Chunked encoding support. The *encode_chunked* parameter was added.

`HTTPConnection.send`(*data*)
Send data to the server. This should be used directly only after the *endheaders()* method has been called and before *getresponse()* is called.

22.12.2 HTTPResponse Objects

An *HTTPResponse* instance wraps the HTTP response from the server. It provides access to the request headers and the entity body. The response is an iterable object and can be used in a with statement.

Changed in version 3.5: The *io.BufferedIOBase* interface is now implemented and all of its reader operations are supported.

`HTTPResponse.read`([*amt*])
Reads and returns the response body, or up to the next *amt* bytes.

`HTTPResponse.readinto`(*b*)
Reads up to the next len(b) bytes of the response body into the buffer *b*. Returns the number of bytes read.

New in version 3.3.

`HTTPResponse.getheader`(*name*, *default=None*)
Return the value of the header *name*, or *default* if there is no header matching *name*. If there is more than one header with the name *name*, return all of the values joined by ', '. If 'default' is any iterable other than a single string, its elements are similarly returned joined by commas.

`HTTPResponse.getheaders`()
Return a list of (header, value) tuples.

`HTTPResponse.fileno`()
Return the `fileno` of the underlying socket.

HTTPResponse.**msg**
A `http.client.HTTPMessage` instance containing the response headers. `http.client.HTTPMessage` is a subclass of *email.message.Message*.

HTTPResponse.**version**
HTTP protocol version used by server. 10 for HTTP/1.0, 11 for HTTP/1.1.

HTTPResponse.**status**
Status code returned by server.

HTTPResponse.**reason**
Reason phrase returned by server.

HTTPResponse.**debuglevel**
A debugging hook. If *debuglevel* is greater than zero, messages will be printed to stdout as the response is read and parsed.

HTTPResponse.**closed**
Is `True` if the stream is closed.

22.12.3 Examples

Here is an example session that uses the `GET` method:

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> print(r1.status, r1.reason)
200 OK
>>> data1 = r1.read()  # This will return entire content.
>>> # The following example demonstrates reading data in chunks.
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> while not r1.closed:
...     print(r1.read(200))  # 200 bytes
b'<!doctype html>\n<!--[if"...
...
>>> # Example of an invalid request
>>> conn = http.client.HTTPSConnection("docs.python.org")
>>> conn.request("GET", "/parrot.spam")
>>> r2 = conn.getresponse()
>>> print(r2.status, r2.reason)
404 Not Found
>>> data2 = r2.read()
>>> conn.close()
```

Here is an example session that uses the `HEAD` method. Note that the `HEAD` method never returns any data.

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("HEAD", "/")
>>> res = conn.getresponse()
>>> print(res.status, res.reason)
200 OK
>>> data = res.read()
```

(continues on next page)

(continued from previous page)

```
>>> print(len(data))
0
>>> data == b''
True
```

Here is an example session that shows how to POST requests:

```
>>> import http.client, urllib.parse
>>> params = urllib.parse.urlencode({'@number': 12524, '@type': 'issue', '@action': 'show
↪'})
>>> headers = {"Content-type": "application/x-www-form-urlencoded",
...            "Accept": "text/plain"}
>>> conn = http.client.HTTPConnection("bugs.python.org")
>>> conn.request("POST", "", params, headers)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
302 Found
>>> data = response.read()
>>> data
b'Redirecting to <a href="http://bugs.python.org/issue12524">http://bugs.python.org/
↪issue12524</a>'
>>> conn.close()
```

Client side HTTP PUT requests are very similar to POST requests. The difference lies only the server side where HTTP server will allow resources to be created via PUT request. It should be noted that custom HTTP methods +are also handled in *urllib.request.Request* by sending the appropriate +method attribute.Here is an example session that shows how to do PUT request using http.client:

```
>>> # This creates an HTTP message
>>> # with the content of BODY as the enclosed representation
>>> # for the resource http://localhost:8080/file
...
>>> import http.client
>>> BODY = "***filecontents***"
>>> conn = http.client.HTTPConnection("localhost", 8080)
>>> conn.request("PUT", "/file", BODY)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
200, OK
```

22.12.4 HTTPMessage Objects

An http.client.HTTPMessage instance holds the headers from an HTTP response. It is implemented using the *email.message.Message* class.

22.13 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()
... # doctest: +SKIP
'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', 'clockspeed', 'djbdns-jedi',
↪ 'docs', 'eaccelerator-jedi', 'favicon.ico', 'francotone', 'fugu', 'ignore',
↪'libpuzzle', 'metalog', 'minidentd', 'misc', 'mysql-udf-global-user-variables',
↪'php-jenkins-hash', 'php-skein-hash', 'php-webdav', 'phpaudit', 'phpbench',
↪'pincaster', 'ping', 'posto', 'pub', 'public', 'public_keys', 'pure-ftpd', 'qscan
↪', 'qtc', 'sharedance', 'skycache', 'sound', 'tmp', 'ucarp']
```

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.

22.13.1 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).

22.13.2 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.

22.14 poplib — POP3 protocol client

Source code: Lib/poplib.py

This module defines a class, *POP3*, which encapsulates a connection to a POP3 server and implements the protocol as defined in **RFC 1939**. The *POP3* class supports both the minimal and optional command sets from **RFC 1939**. The *POP3* class also supports the `STLS` command introduced in **RFC 2595** to enable encrypted communication on an already established connection.

Additionally, this module provides a class *POP3_SSL*, which provides support for connecting to POP3 servers that use SSL as an underlying protocol layer.

Note that POP3, though widely supported, is obsolescent. The implementation quality of POP3 servers varies widely, and too many are quite poor. If your mailserver supports IMAP, you would be better off using the *imaplib.IMAP4* class, as IMAP servers tend to be better implemented.

The *poplib* module provides two classes:

class poplib.POP3(*host*, *port=POP3_PORT*[, *timeout*])
This class implements the actual POP3 protocol. The connection is created when the instance is initialized. If *port* is omitted, the standard POP3 port (110) is used. The optional *timeout* parameter specifies a timeout in seconds for the connection attempt (if not specified, the global default timeout setting will be used).

class poplib.POP3_SSL(*host*, *port=POP3_SSL_PORT*, *keyfile=None*, *certfile=None*, *timeout=None*, *context=None*)
This is a subclass of *POP3* that connects to the server over an SSL encrypted socket. If *port* is not specified, 995, the standard POP3-over-SSL port is used. *timeout* works as in the *POP3* constructor. *context* is an optional *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.

Changed in version 3.2: *context* parameter 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.

One exception is defined as an attribute of the *poplib* module:

exception poplib.error_proto
Exception raised on any errors from this module (errors from *socket* module are not caught). The reason for the exception is passed to the constructor as a string.

See also:

Module *imaplib* The standard Python IMAP module.

Frequently Asked Questions About Fetchmail The FAQ for the `fetchmail` POP/IMAP client collects information on POP3 server variations and RFC noncompliance that may be useful if you need to write an application based on the POP protocol.

22.14.1 POP3 Objects

All POP3 commands are represented by methods of the same name, in lower-case; most return the response text sent by the server.

An *POP3* instance has the following methods:

POP3.**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.

POP3.**getwelcome**()
Returns the greeting string sent by the POP3 server.

POP3.**capa**()
Query the server's capabilities as specified in **RFC 2449**. Returns a dictionary in the form `{'name': ['param'...]}`.

New in version 3.4.

POP3.**user**(*username*)
Send user command, response should indicate that a password is required.

POP3.**pass_**(*password*)
Send password, response includes message count and mailbox size. Note: the mailbox on the server is locked until `quit()` is called.

POP3.**apop**(*user*, *secret*)
Use the more secure APOP authentication to log into the POP3 server.

POP3.**rpop**(*user*)
Use RPOP authentication (similar to UNIX r-commands) to log into POP3 server.

POP3.**stat**()
Get mailbox status. The result is a tuple of 2 integers: `(message count, mailbox size)`.

POP3.**list**([*which*])
Request message list, result is in the form `(response, ['mesg_num octets', ...], octets)`. If *which* is set, it is the message to list.

POP3.**retr**(*which*)
Retrieve whole message number *which*, and set its seen flag. Result is in form `(response, ['line', ...], octets)`.

POP3.**dele**(*which*)
Flag message number *which* for deletion. On most servers deletions are not actually performed until QUIT (the major exception is Eudora QPOP, which deliberately violates the RFCs by doing pending deletes on any disconnect).

POP3.**rset**()
Remove any deletion marks for the mailbox.

POP3.**noop**()
Do nothing. Might be used as a keep-alive.

POP3.**quit**()
Signoff: commit changes, unlock mailbox, drop connection.

POP3.**top**(*which*, *howmuch*)
Retrieves the message header plus *howmuch* lines of the message after the header of message number *which*. Result is in form `(response, ['line', ...], octets)`.

The POP3 TOP command this method uses, unlike the RETR command, doesn't set the message's seen flag; unfortunately, TOP is poorly specified in the RFCs and is frequently broken in off-brand servers. Test this method by hand against the POP3 servers you will use before trusting it.

`POP3.uidl(`*which=None*`)`
Return message digest (unique id) list. If *which* is specified, result contains the unique id for that message in the form `'response mesgnum uid`, otherwise result is list `(response, ['mesgnum uid', ...], octets)`.

`POP3.utf8()`
Try to switch to UTF-8 mode. Returns the server response if successful, raises *error_proto* if not. Specified in **RFC 6856**.

New in version 3.5.

`POP3.stls(`*context=None*`)`
Start a TLS session on the active connection as specified in **RFC 2595**. This is only allowed before user authentication

context parameter 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.

This method supports hostname checking via *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

New in version 3.4.

Instances of *POP3_SSL* have no additional methods. The interface of this subclass is identical to its parent.

22.14.2 POP3 Example

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

```
import getpass, poplib

M = poplib.POP3('localhost')
M.user(getpass.getuser())
M.pass_(getpass.getpass())
numMessages = len(M.list()[1])
for i in range(numMessages):
    for j in M.retr(i+1)[1]:
        print(j)
```

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

22.15 `imaplib` — IMAP4 protocol client

Source code: Lib/imaplib.py

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 **RFC 2060**. It is backward compatible with IMAP4 (**RFC 1730**) servers, but note that the `STATUS` command is not supported in IMAP4.

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

class imaplib.IMAP4(*host=''*, *port=IMAP4_PORT*)

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 *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'])
```

Changed in version 3.5: Support for the `with` statement was added.

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

exception IMAP4.error

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

exception 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.

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:

class imaplib.IMAP4_SSL(*host=''*, *port=IMAP4_SSL_PORT*, *keyfile=None*, *certfile=None*, *ssl_context=None*)

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 *Security considerations* for best practices.

keyfile and *certfile* are a legacy alternative to *ssl_context* - they can point to PEM-formatted private key and certificate chain files for the SSL connection. Note that the *keyfile*/*certfile* parameters are mutually exclusive with *ssl_context*, a *ValueError* is raised if *keyfile*/*certfile* is provided along with *ssl_context*.

Changed in version 3.3: *ssl_context* parameter 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 *ssl_context*. Please use *ssl.SSLContext.load_cert_chain()* instead, or let *ssl.create_default_context()* select the system's trusted CA certificates for you.

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

class imaplib.IMAP4_stream(*command*)

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:

imaplib.**Internaldate2tuple**(*datestr*)
Parse an IMAP4 INTERNALDATE string and return corresponding local time. The return value is a *time.struct_time* tuple or None if the string has wrong format.

imaplib.**Int2AP**(*num*)
Converts an integer into a string representation using characters from the set [A .. P].

imaplib.**ParseFlags**(*flagstr*)
Converts an IMAP4 FLAGS response to a tuple of individual flags.

imaplib.**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.

See also:

Documents describing the protocol, and sources and binaries for servers implementing it, can all be found at the University of Washington's *IMAP Information Center* (https://www.washington.edu/imap/).

22.15.1 IMAP4 Objects

All IMAP4rev1 commands are represented by methods of the same name, either upper-case or lower-case.

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)').

Each command returns 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 string, 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 commands below is 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:*').

An *IMAP4* instance has the following methods:

IMAP4.**append**(*mailbox*, *flags*, *date_time*, *message*)
Append *message* to named mailbox.

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.

Changed in version 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*`)`
Copy *message_set* messages onto end of *new_mailbox*.

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

New in version 3.5: The *enable()* method itself, and **RFC 6855** support.

`IMAP4.expunge()`
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.

`IMAP4.fetch(`*message_set*, *message_parts*`)`
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.

`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.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. Will only work if the server CAPABILITY response includes the phrase AUTH=CRAM-MD5.

IMAP4.**logout**()
Shutdown connection to server. Returns server BYE response.

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.**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 **RFC 2342**.

IMAP4.**noop**()
Send NOOP to server.

IMAP4.**open**(*host*, *port*)
Opens socket to *port* at *host*. 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.

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*[, ...])
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.

Example:

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

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

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.

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 ANNOTATIONs 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*[, ...])
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.

This is an IMAP4rev1 extension command.

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 *Security considerations* for best practices.

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*).

IMAP4.**status**(*mailbox*, *names*)
Request named status conditions for *mailbox*.

IMAP4.**store**(*message_set*, *command*, *flag_list*)
Alters flag dispositions for messages in mailbox. *command* is specified by section 6.4.6 of **RFC 2060** as being one of "FLAGS", "+FLAGS", or "-FLAGS", optionally with a suffix of ".SILENT".

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', '\\Deleted')
M.expunge()
```

Note: Creating flags containing ']' (for example: "[test]") violates **RFC 3501** (the IMAP protocol). However, imaplib has historically allowed creation of such tags, and popular IMAP servers, such as Gmail, accept and produce such flags. There are non-Python programs which also create such tags. Although it is an RFC violation and IMAP clients and servers are supposed to be strict, imaplib nonetheless continues to allow such tags 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.

`IMAP4.subscribe(`*mailbox*`)`
Subscribe to new mailbox.

`IMAP4.thread(`*threading_algorithm*, *charset*, *search_criterion*[, ...]`)`
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.

This is an `IMAP4rev1` extension command.

`IMAP4.uid(`*command*, *arg*[, ...]`)`
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.

`IMAP4.unsubscribe(`*mailbox*`)`
Unsubscribe from old mailbox.

`IMAP4.xatom(`*name*[, ...]`)`
Allow simple extension commands notified by server in `CAPABILITY` response.

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

`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.

New in version 3.5.

22.15.2 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()
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()
```

22.16 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.committers')
>>> print('Group', name, 'has', count, 'articles, range', first, 'to', last)
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)
```

(continues on next page)

(continued from previous page)

```
'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')
... # doctest: +SKIP
('211 1755 1 1755 gmane.comp.python.committers', 1755, 1, 1755, 'gmane.comp.python.
↪committers')
>>>
```

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.

22.16.1 NNTP Objects

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

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.

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**(*context=None*)
Send a `STARTTLS` command. This will enable encryption on the NNTP connection. The *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) # doctest: +SKIP
85
>>> groups[0] # doctest: +SKIP
GroupInfo(group='gmane.network.tor.devel', last='4', first='1', flag='m')
```

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) # doctest: +SKIP
295
>>> descs.popitem() # doctest: +SKIP
('gmane.comp.python.bio.general', 'BioPython discussion list (Moderated)')
```

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
```

(continues on next page)

(continued from previous page)

```
>>> list(over.keys())
['xref', 'from', ':lines', ':bytes', 'references', 'date', 'message-id', 'subject']
>>> 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.metaslash.com>')
>>> 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.

22.16.2 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'
```

22.17 smtplib — SMTP protocol client

Source code: Lib/smtplib.py

The *smtplib* module defines an SMTP client session object that can be used to send mail to any Internet machine with an SMTP or ESMTP listener daemon. For details of SMTP and ESMTP operation, consult **RFC 821** (Simple Mail Transfer Protocol) and **RFC 1869** (SMTP Service Extensions).

class smtplib.SMTP(*host=''*, *port=0*, *local_hostname=None*[, *timeout*], *source_address=None*)

An *SMTP* instance encapsulates an SMTP connection. It has methods that support a full repertoire of SMTP and ESMTP operations. If the optional host and port parameters are given, the SMTP *connect()* method is called with those parameters during initialization. If specified, *local_hostname* is used as the FQDN of the local host in the HELO/EHLO command. Otherwise, the local hostname is found using *socket.getfqdn()*. If the *connect()* call returns anything other than a success code, an *SMTPConnectError* is raised. The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). If the timeout expires, *socket.timeout* is raised. The optional source_address parameter allows binding to some specific source address in a machine with multiple network interfaces, and/or to some specific source TCP port. It takes a 2-tuple (host, port), for the socket to bind to as its source address before connecting. If omitted (or if host or port are `''` and/or 0 respectively) the OS default behavior will be used.

For normal use, you should only require the initialization/connect, *sendmail()*, and *SMTP.quit()* methods. An example is included below.

The *SMTP* class supports the `with` statement. When used like this, the SMTP `QUIT` command is issued automatically when the `with` statement exits. E.g.:

```
>>> from smtplib import SMTP
>>> with SMTP("domain.org") as smtp:
...     smtp.noop()
...
(250, b'Ok')
>>>
```

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

Changed in version 3.3: source_address argument was added.

New in version 3.5: The SMTPUTF8 extension (**RFC 6531**) is now supported.

class smtplib.SMTP_SSL(*host=''*, *port=0*, *local_hostname=None*, *keyfile=None*, *certfile=None*[, *timeout*], *context=None*, *source_address=None*)

An *SMTP_SSL* instance behaves exactly the same as instances of *SMTP*. *SMTP_SSL* should be used for situations where SSL is required from the beginning of the connection and using `starttls()` is not appropriate. If *host* is not specified, the local host is used. If *port* is zero, the standard SMTP-over-SSL port (465) is used. The optional arguments *local_hostname*, *timeout* and *source_address* have

the same meaning as they do in the *SMTP* class. *context*, also optional, can contain a *SSLContext* and allows configuring various aspects of the secure connection. Please read *Security considerations* for best practices.

keyfile and *certfile* are a legacy alternative to *context*, and can point to a PEM formatted private key and certificate chain file for the SSL connection.

Changed in version 3.3: *context* was added.

Changed in version 3.3: source_address argument 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.

class smtplib.LMTP(*host=''*, *port=LMTP_PORT*, *local_hostname=None*, *source_address=None*)

The LMTP protocol, which is very similar to ESMTP, is heavily based on the standard SMTP client. It's common to use Unix sockets for LMTP, so our `connect()` method must support that as well as a regular host:port server. The optional arguments local_hostname and source_address have the same meaning as they do in the *SMTP* class. To specify a Unix socket, you must use an absolute path for *host*, starting with a '/'.

Authentication is supported, using the regular SMTP mechanism. When using a Unix socket, LMTP generally don't support or require any authentication, but your mileage might vary.

A nice selection of exceptions is defined as well:

exception smtplib.SMTPException

Subclass of *OSError* that is the base exception class for all the other exceptions provided by this module.

Changed in version 3.4: SMTPException became subclass of *OSError*

exception smtplib.SMTPServerDisconnected

This exception is raised when the server unexpectedly disconnects, or when an attempt is made to use the *SMTP* instance before connecting it to a server.

exception smtplib.SMTPResponseException

Base class for all exceptions that include an SMTP error code. These exceptions are generated in some instances when the SMTP server returns an error code. The error code is stored in the `smtp_code` attribute of the error, and the `smtp_error` attribute is set to the error message.

exception smtplib.SMTPSenderRefused

Sender address refused. In addition to the attributes set by on all *SMTPResponseException* exceptions, this sets 'sender' to the string that the SMTP server refused.

exception smtplib.SMTPRecipientsRefused

All recipient addresses refused. The errors for each recipient are accessible through the attribute `recipients`, which is a dictionary of exactly the same sort as *SMTP.sendmail()* returns.

exception smtplib.SMTPDataError

The SMTP server refused to accept the message data.

exception smtplib.SMTPConnectError

Error occurred during establishment of a connection with the server.

exception smtplib.SMTPHeloError

The server refused our `HELO` message.

exception smtplib.SMTPNotSupportedError

The command or option attempted is not supported by the server.

New in version 3.5.

exception smtplib.SMTPAuthenticationError
SMTP authentication went wrong. Most probably the server didn't accept the username/password combination provided.

See also:

RFC 821 - Simple Mail Transfer Protocol Protocol definition for SMTP. This document covers the model, operating procedure, and protocol details for SMTP.

RFC 1869 - SMTP Service Extensions Definition of the ESMTP extensions for SMTP. This describes a framework for extending SMTP with new commands, supporting dynamic discovery of the commands provided by the server, and defines a few additional commands.

22.17.1 SMTP Objects

An *SMTP* instance has the following methods:

SMTP.set_debuglevel(*level*)
Set the debug output level. A value of 1 or `True` for *level* results in debug messages for connection and for all messages sent to and received from the server. A value of 2 for *level* results in these messages being timestamped.

Changed in version 3.5: Added debuglevel 2.

SMTP.docmd(*cmd*, *args=''*)
Send a command *cmd* to the server. The optional argument *args* is simply concatenated to the command, separated by a space.

This returns a 2-tuple composed of a numeric response code and the actual response line (multiline responses are joined into one long line.)

In normal operation it should not be necessary to call this method explicitly. It is used to implement other methods and may be useful for testing private extensions.

If the connection to the server is lost while waiting for the reply, *SMTPServerDisconnected* will be raised.

SMTP.connect(*host='localhost'*, *port=0*)
Connect to a host on a given port. The defaults are to connect to the local host at the standard SMTP port (25). If the hostname ends with a colon (`':'`) followed by a number, that suffix will be stripped off and the number interpreted as the port number to use. This method is automatically invoked by the constructor if a host is specified during instantiation. Returns a 2-tuple of the response code and message sent by the server in its connection response.

SMTP.helo(*name=''*)
Identify yourself to the SMTP server using `HELO`. The hostname argument defaults to the fully qualified domain name of the local host. The message returned by the server is stored as the `helo_resp` attribute of the object.

In normal operation it should not be necessary to call this method explicitly. It will be implicitly called by the *sendmail()* when necessary.

SMTP.ehlo(*name=''*)
Identify yourself to an ESMTP server using `EHLO`. The hostname argument defaults to the fully qualified domain name of the local host. Examine the response for ESMTP option and store them for use by *has_extn()*. Also sets several informational attributes: the message returned by the server is stored as the `ehlo_resp` attribute, `does_esmtp` is set to true or false depending on whether the server supports ESMTP, and `esmtp_features` will be a dictionary containing the names of the SMTP service extensions this server supports, and their parameters (if any).

Unless you wish to use *has_extn()* before sending mail, it should not be necessary to call this method explicitly. It will be implicitly called by *sendmail()* when necessary.

SMTP.**ehlo_or_helo_if_needed**()
This method calls *ehlo()* and/or *helo()* if there has been no previous EHLO or HELO command this session. It tries ESMTP EHLO first.

SMTPHeloError The server didn't reply properly to the HELO greeting.

SMTP.**has_extn**(*name*)
Return *True* if *name* is in the set of SMTP service extensions returned by the server, *False* otherwise. Case is ignored.

SMTP.**verify**(*address*)
Check the validity of an address on this server using SMTP VRFY. Returns a tuple consisting of code 250 and a full **RFC 822** address (including human name) if the user address is valid. Otherwise returns an SMTP error code of 400 or greater and an error string.

Note: Many sites disable SMTP VRFY in order to foil spammers.

SMTP.**login**(*user*, *password*, *, *initial_response_ok=True*)
Log in on an SMTP server that requires authentication. The arguments are the username and the password to authenticate with. If there has been no previous EHLO or HELO command this session, this method tries ESMTP EHLO first. This method will return normally if the authentication was successful, or may raise the following exceptions:

SMTPHeloError The server didn't reply properly to the HELO greeting.

SMTPAuthenticationError The server didn't accept the username/password combination.

SMTPNotSupportedError The AUTH command is not supported by the server.

SMTPException No suitable authentication method was found.

Each of the authentication methods supported by *smtplib* are tried in turn if they are advertised as supported by the server. See *auth()* for a list of supported authentication methods. *initial_response_ok* is passed through to *auth()*.

Optional keyword argument *initial_response_ok* specifies whether, for authentication methods that support it, an "initial response" as specified in **RFC 4954** can be sent along with the AUTH command, rather than requiring a challenge/response.

Changed in version 3.5: *SMTPNotSupportedError* may be raised, and the *initial_response_ok* parameter was added.

SMTP.**auth**(*mechanism*, *authobject*, *, *initial_response_ok=True*)
Issue an SMTP AUTH command for the specified authentication *mechanism*, and handle the challenge response via *authobject*.

mechanism specifies which authentication mechanism is to be used as argument to the AUTH command; the valid values are those listed in the auth element of esmtp_features.

authobject must be a callable object taking an optional single argument:

data = authobject(challenge=None)

If optional keyword argument *initial_response_ok* is true, authobject() will be called first with no argument. It can return the **RFC 4954** "initial response" ASCII str which will be encoded and sent with the AUTH command as below. If the authobject() does not support an initial response (e.g. because it requires a challenge), it should return None when called with challenge=None. If *initial_response_ok* is false, then authobject() will not be called first with None.

If the initial response check returns `None`, or if *initial_response_ok* is false, `authobject()` will be called to process the server's challenge response; the *challenge* argument it is passed will be a `bytes`. It should return ASCII `str` *data* that will be base64 encoded and sent to the server.

The `SMTP` class provides `authobjects` for the `CRAM-MD5`, `PLAIN`, and `LOGIN` mechanisms; they are named `SMTP.auth_cram_md5`, `SMTP.auth_plain`, and `SMTP.auth_login` respectively. They all require that the `user` and `password` properties of the `SMTP` instance are set to appropriate values.

User code does not normally need to call `auth` directly, but can instead call the *login()* method, which will try each of the above mechanisms in turn, in the order listed. `auth` is exposed to facilitate the implementation of authentication methods not (or not yet) supported directly by *smtplib*.

New in version 3.5.

`SMTP.starttls`(*keyfile=None*, *certfile=None*, *context=None*)
Put the SMTP connection in TLS (Transport Layer Security) mode. All SMTP commands that follow will be encrypted. You should then call *ehlo()* again.

If *keyfile* and *certfile* are provided, they are used to create an *ssl.SSLContext*.

Optional *context* parameter is an *ssl.SSLContext* object; This is an alternative to using a keyfile and a certfile and if specified both *keyfile* and *certfile* should be `None`.

If there has been no previous `EHLO` or `HELO` command this session, this method tries ESMTP `EHLO` first.

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.

SMTPHeloError The server didn't reply properly to the `HELO` greeting.

SMTPNotSupportedError The server does not support the STARTTLS extension.

RuntimeError SSL/TLS support is not available to your Python interpreter.

Changed in version 3.3: *context* was added.

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

Changed in version 3.5: The error raised for lack of STARTTLS support is now the *SMTPNotSupportedError* subclass instead of the base *SMTPException*.

`SMTP.sendmail`(*from_addr*, *to_addrs*, *msg*, *mail_options=()*, *rcpt_options=()*)
Send mail. The required arguments are an **RFC 822** from-address string, a list of **RFC 822** to-address strings (a bare string will be treated as a list with 1 address), and a message string. The caller may pass a list of ESMTP options (such as `8bitmime`) to be used in `MAIL FROM` commands as *mail_options*. ESMTP options (such as `DSN` commands) that should be used with all `RCPT` commands can be passed as *rcpt_options*. (If you need to use different ESMTP options to different recipients you have to use the low-level methods such as `mail()`, `rcpt()` and `data()` to send the message.)

Note: The *from_addr* and *to_addrs* parameters are used to construct the message envelope used by the transport agents. `sendmail` does not modify the message headers in any way.

---

msg may be a string containing characters in the ASCII range, or a byte string. A string is encoded to bytes using the ascii codec, and lone `\r` and `\n` characters are converted to `\r\n` characters. A byte string is not modified.

If there has been no previous `EHLO` or `HELO` command this session, this method tries ESMTP `EHLO` first. If the server does ESMTP, message size and each of the specified options will be passed to it (if the option is in the feature set the server advertises). If `EHLO` fails, `HELO` will be tried and ESMTP options suppressed.

This method will return normally if the mail is accepted for at least one recipient. Otherwise it will raise an exception. That is, if this method does not raise an exception, then someone should get your mail. If this method does not raise an exception, it returns a dictionary, with one entry for each recipient that was refused. Each entry contains a tuple of the SMTP error code and the accompanying error message sent by the server.

If `SMTPUTF8` is included in *mail_options*, and the server supports it, *from_addr* and *to_addrs* may contain non-ASCII characters.

This method may raise the following exceptions:

SMTPRecipientsRefused All recipients were refused. Nobody got the mail. The `recipients` attribute of the exception object is a dictionary with information about the refused recipients (like the one returned when at least one recipient was accepted).

SMTPHeloError The server didn't reply properly to the `HELO` greeting.

SMTPSenderRefused The server didn't accept the *from_addr*.

SMTPDataError The server replied with an unexpected error code (other than a refusal of a recipient).

SMTPNotSupportedError `SMTPUTF8` was given in the *mail_options* but is not supported by the server.

Unless otherwise noted, the connection will be open even after an exception is raised.

Changed in version 3.2: *msg* may be a byte string.

Changed in version 3.5: `SMTPUTF8` support added, and *SMTPNotSupportedError* may be raised if `SMTPUTF8` is specified but the server does not support it.

`SMTP.send_message`(*msg*, *from_addr=None*, *to_addrs=None*, *mail_options=()*, *rcpt_options=()*)
This is a convenience method for calling *sendmail()* with the message represented by an *email.message.Message* object. The arguments have the same meaning as for *sendmail()*, except that *msg* is a `Message` object.

If *from_addr* is `None` or *to_addrs* is `None`, `send_message` fills those arguments with addresses extracted from the headers of *msg* as specified in **RFC 5322**: *from_addr* is set to the *Sender* field if it is present, and otherwise to the *From* field. *to_addrs* combines the values (if any) of the *To*, *Cc*, and *Bcc* fields from *msg*. If exactly one set of *Resent-** headers appear in the message, the regular headers are ignored and the *Resent-** headers are used instead. If the message contains more than one set of *Resent-** headers, a *ValueError* is raised, since there is no way to unambiguously detect the most recent set of *Resent-* headers.

`send_message` serializes *msg* using *BytesGenerator* with `\r\n` as the *linesep*, and calls *sendmail()* to transmit the resulting message. Regardless of the values of *from_addr* and *to_addrs*, `send_message` does not transmit any *Bcc* or *Resent-Bcc* headers that may appear in *msg*. If any of the addresses in *from_addr* and *to_addrs* contain non-ASCII characters and the server does not advertise `SMTPUTF8` support, an `SMTPNotSupported` error is raised. Otherwise the `Message` is serialized with a clone of its *policy* with the *utf8* attribute set to `True`, and `SMTPUTF8` and `BODY=8BITMIME` are added to *mail_options*.

New in version 3.2.

New in version 3.5: Support for internationalized addresses (`SMTPUTF8`).

`SMTP.quit`()
Terminate the SMTP session and close the connection. Return the result of the SMTP `QUIT` command.

Low-level methods corresponding to the standard SMTP/ESMTP commands `HELP`, `RSET`, `NOOP`, `MAIL`, `RCPT`, and `DATA` are also supported. Normally these do not need to be called directly, so they are not documented here. For details, consult the module code.

22.17.2 SMTP Example

This example prompts the user for addresses needed in the message envelope ('To' and 'From' addresses), and the message to be delivered. Note that the headers to be included with the message must be included in the message as entered; this example doesn't do any processing of the **RFC 822** headers. In particular, the 'To' and 'From' addresses must be included in the message headers explicitly.

```
import smtplib

def prompt(prompt):
    return input(prompt).strip()

fromaddr = prompt("From: ")
toaddrs  = prompt("To: ").split()
print("Enter message, end with ^D (Unix) or ^Z (Windows):")

# Add the From: and To: headers at the start!
msg = ("From: %s\r\nTo: %s\r\n\r\n"
       % (fromaddr, ", ".join(toaddrs)))
while True:
    try:
        line = input()
    except EOFError:
        break
    if not line:
        break
    msg = msg + line

print("Message length is", len(msg))

server = smtplib.SMTP('localhost')
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

Note: In general, you will want to use the *email* package's features to construct an email message, which you can then send via *send_message()*; see *email: Examples*.

22.18 smtpd — SMTP Server

Source code: Lib/smtpd.py

This module offers several classes to implement SMTP (email) servers.

See also:

The aiosmtpd package is a recommended replacement for this module. It is based on *asyncio* and provides a more straightforward API. *smtpd* should be considered deprecated.

Several server implementations are present; one is a generic do-nothing implementation, which can be overridden, while the other two offer specific mail-sending strategies.

Additionally the SMTPChannel may be extended to implement very specific interaction behaviour with SMTP clients.

The code supports **RFC 5321**, plus the **RFC 1870** SIZE and **RFC 6531** SMTPUTF8 extensions.

22.18.1 SMTPServer Objects

class smtpd.SMTPServer(*localaddr*, *remoteaddr*, *data_size_limit=33554432*, *map=None*, *enable_SMTPUTF8=False*, *decode_data=False*)

Create a new *SMTPServer* object, which binds to local address *localaddr*. It will treat *remoteaddr* as an upstream SMTP relayer. Both *localaddr* and *remoteaddr* should be a *(host, port)* tuple. The object inherits from *asyncore.dispatcher*, and so will insert itself into *asyncore*'s event loop on instantiation.

data_size_limit specifies the maximum number of bytes that will be accepted in a `DATA` command. A value of `None` or `0` means no limit.

map is the socket map to use for connections (an initially empty dictionary is a suitable value). If not specified the *asyncore* global socket map is used.

enable_SMTPUTF8 determines whether the `SMTPUTF8` extension (as defined in **RFC 6531**) should be enabled. The default is `False`. When `True`, `SMTPUTF8` is accepted as a parameter to the `MAIL` command and when present is passed to *process_message()* in the `kwargs['mail_options']` list. *decode_data* and *enable_SMTPUTF8* cannot be set to `True` at the same time.

decode_data specifies whether the data portion of the SMTP transaction should be decoded using UTF-8. When *decode_data* is `False` (the default), the server advertises the `8BITMIME` extension (**RFC 6152**), accepts the `BODY=8BITMIME` parameter to the `MAIL` command, and when present passes it to *process_message()* in the `kwargs['mail_options']` list. *decode_data* and *enable_SMTPUTF8* cannot be set to `True` at the same time.

process_message(*peer*, *mailfrom*, *rcpttos*, *data*, ***kwargs*)

Raise a *NotImplementedError* exception. Override this in subclasses to do something useful with this message. Whatever was passed in the constructor as *remoteaddr* will be available as the `_remoteaddr` attribute. *peer* is the remote host's address, *mailfrom* is the envelope originator, *rcpttos* are the envelope recipients and *data* is a string containing the contents of the e-mail (which should be in **RFC 5321** format).

If the *decode_data* constructor keyword is set to `True`, the *data* argument will be a unicode string. If it is set to `False`, it will be a bytes object.

kwargs is a dictionary containing additional information. It is empty if `decode_data=True` was given as an init argument, otherwise it contains the following keys:

mail_options: a list of all received parameters to the `MAIL` command (the elements are uppercase strings; example: `['BODY=8BITMIME', 'SMTPUTF8']`).

rcpt_options: same as *mail_options* but for the `RCPT` command. Currently no `RCPT TO` options are supported, so for now this will always be an empty list.

Implementations of `process_message` should use the `**kwargs` signature to accept arbitrary keyword arguments, since future feature enhancements may add keys to the kwargs dictionary.

Return `None` to request a normal `250 Ok` response; otherwise return the desired response string in **RFC 5321** format.

channel_class

Override this in subclasses to use a custom *SMTPChannel* for managing SMTP clients.

New in version 3.4: The *map* constructor argument.

Changed in version 3.5: *localaddr* and *remoteaddr* may now contain IPv6 addresses.

New in version 3.5: The *decode_data* and *enable_SMTPUTF8* constructor parameters, and the *kwargs* parameter to *process_message()* when *decode_data* is `False`.

Changed in version 3.6: *decode_data* is now `False` by default.

22.18.2 DebuggingServer Objects

`class smtpd.DebuggingServer(`*localaddr*, *remoteaddr*`)`
Create a new debugging server. Arguments are as per *SMTPServer*. Messages will be discarded, and printed on stdout.

22.18.3 PureProxy Objects

`class smtpd.PureProxy(`*localaddr*, *remoteaddr*`)`
Create a new pure proxy server. Arguments are as per *SMTPServer*. Everything will be relayed to *remoteaddr*. Note that running this has a good chance to make you into an open relay, so please be careful.

22.18.4 MailmanProxy Objects

`class smtpd.MailmanProxy(`*localaddr*, *remoteaddr*`)`
Create a new pure proxy server. Arguments are as per *SMTPServer*. Everything will be relayed to *remoteaddr*, unless local mailman configurations knows about an address, in which case it will be handled via mailman. Note that running this has a good chance to make you into an open relay, so please be careful.

22.18.5 SMTPChannel Objects

`class smtpd.SMTPChannel(`*server*, *conn*, *addr*, *data_size_limit=33554432*, *map=None*, *enable_SMTPUTF8=False*, *decode_data=False*`)`
Create a new *SMTPChannel* object which manages the communication between the server and a single SMTP client.

conn and *addr* are as per the instance variables described below.

data_size_limit specifies the maximum number of bytes that will be accepted in a `DATA` command. A value of `None` or `0` means no limit.

enable_SMTPUTF8 determines whether the `SMTPUTF8` extension (as defined in **RFC 6531**) should be enabled. The default is `False`. *decode_data* and *enable_SMTPUTF8* cannot be set to `True` at the same time.

A dictionary can be specified in *map* to avoid using a global socket map.

decode_data specifies whether the data portion of the SMTP transaction should be decoded using UTF-8. The default is `False`. *decode_data* and *enable_SMTPUTF8* cannot be set to `True` at the same time.

To use a custom SMTPChannel implementation you need to override the *SMTPServer.channel_class* of your *SMTPServer*.

Changed in version 3.5: The *decode_data* and *enable_SMTPUTF8* parameters were added.

Changed in version 3.6: *decode_data* is now `False` by default.

The *SMTPChannel* has the following instance variables:

smtp_server
Holds the *SMTPServer* that spawned this channel.

conn
Holds the socket object connecting to the client.

addr
Holds the address of the client, the second value returned by *socket.accept*

received_lines
Holds a list of the line strings (decoded using UTF-8) received from the client. The lines have their `"\r\n"` line ending translated to `"\n"`.

smtp_state
Holds the current state of the channel. This will be either `COMMAND` initially and then `DATA` after the client sends a "DATA" line.

seen_greeting
Holds a string containing the greeting sent by the client in its "HELO".

mailfrom
Holds a string containing the address identified in the "MAIL FROM:" line from the client.

rcpttos
Holds a list of strings containing the addresses identified in the "RCPT TO:" lines from the client.

received_data
Holds a string containing all of the data sent by the client during the DATA state, up to but not including the terminating `"\r\n.\r\n"`.

fqdn
Holds the fully-qualified domain name of the server as returned by *socket.getfqdn()*.

peer
Holds the name of the client peer as returned by `conn.getpeername()` where `conn` is *conn*.

The *SMTPChannel* operates by invoking methods named `smtp_<command>` upon reception of a command line from the client. Built into the base *SMTPChannel* class are methods for handling the following commands (and responding to them appropriately):

Command	Action taken
HELO	Accepts the greeting from the client and stores it in *seen_greeting*. Sets server to base command mode.
EHLO	Accepts the greeting from the client and stores it in *seen_greeting*. Sets server to extended command mode.
NOOP	Takes no action.
QUIT	Closes the connection cleanly.
MAIL	Accepts the "MAIL FROM:" syntax and stores the supplied address as *mailfrom*. In extended command mode, accepts the **RFC 1870** SIZE attribute and responds appropriately based on the value of *data_size_limit*.
RCPT	Accepts the "RCPT TO:" syntax and stores the supplied addresses in the *rcpttos* list.
RSET	Resets the *mailfrom*, *rcpttos*, and *received_data*, but not the greeting.
DATA	Sets the internal state to `DATA` and stores remaining lines from the client in *received_data* until the terminator `"\r\n.\r\n"` is received.
HELP	Returns minimal information on command syntax
VRFY	Returns code 252 (the server doesn't know if the address is valid)
EXPN	Reports that the command is not implemented.

22.19 telnetlib — Telnet client

Source code: Lib/telnetlib.py

The *telnetlib* module provides a *Telnet* class that implements the Telnet protocol. See **RFC 854** for details about the protocol. In addition, it provides symbolic constants for the protocol characters (see below), and for the telnet options. The symbolic names of the telnet options follow the definitions in `arpa/telnet.h`, with the leading `TELOPT_` removed. For symbolic names of options which are traditionally not included in `arpa/telnet.h`, see the module source itself.

The symbolic constants for the telnet commands are: IAC, DONT, DO, WONT, WILL, SE (Subnegotiation End), NOP (No Operation), DM (Data Mark), BRK (Break), IP (Interrupt process), AO (Abort output), AYT (Are You There), EC (Erase Character), EL (Erase Line), GA (Go Ahead), SB (Subnegotiation Begin).

class telnetlib.Telnet(*host=None*, *port=0*[, *timeout*])

Telnet represents a connection to a Telnet server. The instance is initially not connected by default; the *open()* method must be used to establish a connection. Alternatively, the host name and optional port number can be passed to the constructor too, in which case the connection to the server will be established before the constructor returns. The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used).

Do not reopen an already connected instance.

This class has many `read_*()` methods. Note that some of them raise *EOFError* when the end of the connection is read, because they can return an empty string for other reasons. See the individual descriptions below.

A *Telnet* object is a context manager and can be used in a `with` statement. When the `with` block ends, the *close()* method is called:

```
>>> from telnetlib import Telnet
>>> with Telnet('localhost', 23) as tn:
...     tn.interact()
...
```

Changed in version 3.6: Context manager support added

See also:

RFC 854 - Telnet Protocol Specification Definition of the Telnet protocol.

22.19.1 Telnet Objects

Telnet instances have the following methods:

Telnet.read_until(*expected*, *timeout=None*)

Read until a given byte string, *expected*, is encountered or until *timeout* seconds have passed.

When no match is found, return whatever is available instead, possibly empty bytes. Raise *EOFError* if the connection is closed and no cooked data is available.

Telnet.read_all()

Read all data until EOF as bytes; block until connection closed.

Telnet.read_some()

Read at least one byte of cooked data unless EOF is hit. Return `b''` if EOF is hit. Block if no data is immediately available.

Telnet.**read_very_eager**()
Read everything that can be without blocking in I/O (eager).

Raise *EOFError* if connection closed and no cooked data available. Return `b''` if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

Telnet.**read_eager**()
Read readily available data.

Raise *EOFError* if connection closed and no cooked data available. Return `b''` if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

Telnet.**read_lazy**()
Process and return data already in the queues (lazy).

Raise *EOFError* if connection closed and no data available. Return `b''` if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

Telnet.**read_very_lazy**()
Return any data available in the cooked queue (very lazy).

Raise *EOFError* if connection closed and no data available. Return `b''` if no cooked data available otherwise. This method never blocks.

Telnet.**read_sb_data**()
Return the data collected between a SB/SE pair (suboption begin/end). The callback should access these data when it was invoked with a `SE` command. This method never blocks.

Telnet.**open**(*host*, *port=0*[, *timeout*])
Connect to a host. The optional second argument is the port number, which defaults to the standard Telnet port (23). The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used).

Do not try to reopen an already connected instance.

Telnet.**msg**(*msg*, **args*)
Print a debug message when the debug level is > 0. If extra arguments are present, they are substituted in the message using the standard string formatting operator.

Telnet.**set_debuglevel**(*debuglevel*)
Set the debug level. The higher the value of *debuglevel*, the more debug output you get (on `sys.stdout`).

Telnet.**close**()
Close the connection.

Telnet.**get_socket**()
Return the socket object used internally.

Telnet.**fileno**()
Return the file descriptor of the socket object used internally.

Telnet.**write**(*buffer*)
Write a byte string to the socket, doubling any IAC characters. This can block if the connection is blocked. May raise *OSError* if the connection is closed.

Changed in version 3.3: This method used to raise *socket.error*, which is now an alias of *OSError*.

Telnet.**interact**()
Interaction function, emulates a very dumb Telnet client.

Telnet.**mt_interact**()
Multithreaded version of *interact()*.

Telnet.**expect**(*list*, *timeout=None*)
Read until one from a list of a regular expressions matches.

The first argument is a list of regular expressions, either compiled (*regex objects*) or uncompiled (byte strings). The optional second argument is a timeout, in seconds; the default is to block indefinitely.

Return a tuple of three items: the index in the list of the first regular expression that matches; the match object returned; and the bytes read up till and including the match.

If end of file is found and no bytes were read, raise *EOFError*. Otherwise, when nothing matches, return `(-1, None, data)` where *data* is the bytes received so far (may be empty bytes if a timeout happened).

If a regular expression ends with a greedy match (such as `.*`) or if more than one expression can match the same input, the results are non-deterministic, and may depend on the I/O timing.

Telnet.**set_option_negotiation_callback**(*callback*)
Each time a telnet option is read on the input flow, this *callback* (if set) is called with the following parameters: callback(telnet socket, command (DO/DONT/WILL/WONT), option). No other action is done afterwards by telnetlib.

22.19.2 Telnet Example

A simple example illustrating typical use:

```
import getpass
import telnetlib

HOST = "localhost"
user = input("Enter your remote account: ")
password = getpass.getpass()

tn = telnetlib.Telnet(HOST)

tn.read_until(b"login: ")
tn.write(user.encode('ascii') + b"\n")
if password:
    tn.read_until(b"Password: ")
    tn.write(password.encode('ascii') + b"\n")

tn.write(b"ls\n")
tn.write(b"exit\n")

print(tn.read_all().decode('ascii'))
```

22.20 uuid — UUID objects according to RFC 4122

Source code: Lib/uuid.py

This module provides immutable *UUID* objects (the *UUID* class) and the functions *uuid1()*, *uuid3()*, *uuid4()*, *uuid5()* for generating version 1, 3, 4, and 5 UUIDs as specified in **RFC 4122**.

If all you want is a unique ID, you should probably call *uuid1()* or *uuid4()*. Note that *uuid1()* may compromise privacy since it creates a UUID containing the computer's network address. *uuid4()* creates a random UUID.

Depending on support from the underlying platform, *uuid1()* may or may not return a "safe" UUID. A safe UUID is one which is generated using synchronization methods that ensure no two processes can obtain the same UUID. All instances of *UUID* have an `is_safe` attribute which relays any information about the UUID's safety, using this enumeration:

class uuid.SafeUUID

New in version 3.7.

safe
The UUID was generated by the platform in a multiprocessing-safe way.

unsafe
The UUID was not generated in a multiprocessing-safe way.

unknown
The platform does not provide information on whether the UUID was generated safely or not.

class uuid.UUID(*hex=None*, *bytes=None*, *bytes_le=None*, *fields=None*, *int=None*, *version=None*, ***, *is_safe=SafeUUID.unknown*)

Create a UUID from either a string of 32 hexadecimal digits, a string of 16 bytes in big-endian order as the *bytes* argument, a string of 16 bytes in little-endian order as the *bytes_le* argument, a tuple of six integers (32-bit *time_low*, 16-bit *time_mid*, 16-bit *time_hi_version*, 8-bit *clock_seq_hi_variant*, 8-bit *clock_seq_low*, 48-bit *node*) as the *fields* argument, or a single 128-bit integer as the *int* argument. When a string of hex digits is given, curly braces, hyphens, and a URN prefix are all optional. For example, these expressions all yield the same UUID:

```
UUID('{12345678-1234-5678-1234-567812345678}')
UUID('12345678123456781234567812345678')
UUID('urn:uuid:12345678-1234-5678-1234-567812345678')
UUID(bytes=b'\x12\x34\x56\x78'*4)
UUID(bytes_le=b'\x78\x56\x34\x12\x34\x12\x78\x56' +
              b'\x12\x34\x56\x78\x12\x34\x56\x78')
UUID(fields=(0x12345678, 0x1234, 0x5678, 0x12, 0x34, 0x567812345678))
UUID(int=0x12345678123456781234567812345678)
```

Exactly one of *hex*, *bytes*, *bytes_le*, *fields*, or *int* must be given. The *version* argument is optional; if given, the resulting UUID will have its variant and version number set according to **RFC 4122**, overriding bits in the given *hex*, *bytes*, *bytes_le*, *fields*, or *int*.

Comparison of UUID objects are made by way of comparing their *UUID.int* attributes. Comparison with a non-UUID object raises a *TypeError*.

`str(uuid)` returns a string in the form `12345678-1234-5678-1234-567812345678` where the 32 hexadecimal digits represent the UUID.

UUID instances have these read-only attributes:

UUID.bytes
The UUID as a 16-byte string (containing the six integer fields in big-endian byte order).

UUID.bytes_le
The UUID as a 16-byte string (with *time_low*, *time_mid*, and *time_hi_version* in little-endian byte order).

UUID.fields
A tuple of the six integer fields of the UUID, which are also available as six individual attributes and two derived attributes:

Field	Meaning
`time_low`	the first 32 bits of the UUID
`time_mid`	the next 16 bits of the UUID
`time_hi_version`	the next 16 bits of the UUID
`clock_seq_hi_variant`	the next 8 bits of the UUID
`clock_seq_low`	the next 8 bits of the UUID
`node`	the last 48 bits of the UUID
time	the 60-bit timestamp
`clock_seq`	the 14-bit sequence number

UUID.hex
The UUID as a 32-character hexadecimal string.

UUID.int
The UUID as a 128-bit integer.

UUID.urn
The UUID as a URN as specified in **RFC 4122**.

UUID.variant
The UUID variant, which determines the internal layout of the UUID. This will be one of the constants *RESERVED_NCS*, *RFC_4122*, *RESERVED_MICROSOFT*, or *RESERVED_FUTURE*.

UUID.version
The UUID version number (1 through 5, meaningful only when the variant is *RFC_4122*).

UUID.is_safe
An enumeration of *SafeUUID* which indicates whether the platform generated the UUID in a multiprocessing-safe way.

New in version 3.7.

The *uuid* module defines the following functions:

uuid.getnode()
Get the hardware address as a 48-bit positive integer. The first time this runs, it may launch a separate program, which could be quite slow. If all attempts to obtain the hardware address fail, we choose a random 48-bit number with the multicast bit (least significant bit of the first octet) set to 1 as recommended in **RFC 4122**. "Hardware address" means the MAC address of a network interface. On a machine with multiple network interfaces, universally administered MAC addresses (i.e. where the second least significant bit of the first octet is *unset*) will be preferred over locally administered MAC addresses, but with no other ordering guarantees.

Changed in version 3.7: Universally administered MAC addresses are preferred over locally administered MAC addresses, since the former are guaranteed to be globally unique, while the latter are not.

uuid.uuid1(*node=None*, *clock_seq=None*)
Generate a UUID from a host ID, sequence number, and the current time. If *node* is not given, *getnode()* is used to obtain the hardware address. If *clock_seq* is given, it is used as the sequence number; otherwise a random 14-bit sequence number is chosen.

uuid.uuid3(*namespace*, *name*)
Generate a UUID based on the MD5 hash of a namespace identifier (which is a UUID) and a name (which is a string).

uuid.uuid4()
Generate a random UUID.

uuid.uuid5(*namespace*, *name*)
Generate a UUID based on the SHA-1 hash of a namespace identifier (which is a UUID) and a name (which is a string).

The *uuid* module defines the following namespace identifiers for use with *uuid3()* or *uuid5()*.

uuid.NAMESPACE_DNS
When this namespace is specified, the *name* string is a fully-qualified domain name.

uuid.NAMESPACE_URL
When this namespace is specified, the *name* string is a URL.

uuid.NAMESPACE_OID
When this namespace is specified, the *name* string is an ISO OID.

uuid.NAMESPACE_X500
When this namespace is specified, the *name* string is an X.500 DN in DER or a text output format.

The *uuid* module defines the following constants for the possible values of the `variant` attribute:

uuid.RESERVED_NCS
Reserved for NCS compatibility.

uuid.RFC_4122
Specifies the UUID layout given in **RFC 4122**.

uuid.RESERVED_MICROSOFT
Reserved for Microsoft compatibility.

uuid.RESERVED_FUTURE
Reserved for future definition.

See also:

RFC 4122 - A Universally Unique IDentifier (UUID) URN Namespace This specification defines a Uniform Resource Name namespace for UUIDs, the internal format of UUIDs, and methods of generating UUIDs.

22.20.1 Example

Here are some examples of typical usage of the *uuid* module:

```
>>> import uuid

>>> # make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

>>> # make a UUID using an MD5 hash of a namespace UUID and a name
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

>>> # make a random UUID
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

>>> # make a UUID using a SHA-1 hash of a namespace UUID and a name
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9aee199e5d')
```

(continues on next page)

(continued from previous page)

```
>>> # make a UUID from a string of hex digits (braces and hyphens ignored)
>>> x = uuid.UUID('{00010203-0405-0607-0809-0a0b0c0d0e0f}')

>>> # convert a UUID to a string of hex digits in standard form
>>> str(x)
'00010203-0405-0607-0809-0a0b0c0d0e0f'

>>> # get the raw 16 bytes of the UUID
>>> x.bytes
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'

>>> # make a UUID from a 16-byte string
>>> uuid.UUID(bytes=x.bytes)
UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')
```

22.21 socketserver — A framework for network servers

Source code: Lib/socketserver.py

The *socketserver* module simplifies the task of writing network servers.

There are four basic concrete server classes:

class socketserver.TCPServer(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
This uses the Internet TCP protocol, which provides for continuous streams of data between the client and server. If *bind_and_activate* is true, the constructor automatically attempts to invoke *server_bind()* and *server_activate()*. The other parameters are passed to the *BaseServer* base class.

class socketserver.UDPServer(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
This uses datagrams, which are discrete packets of information that may arrive out of order or be lost while in transit. The parameters are the same as for *TCPServer*.

class socketserver.UnixStreamServer(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
class socketserver.UnixDatagramServer(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
These more infrequently used classes are similar to the TCP and UDP classes, but use Unix domain sockets; they're not available on non-Unix platforms. The parameters are the same as for *TCPServer*.

These four classes process requests *synchronously*; each request must be completed before the next request can be started. This isn't suitable if each request takes a long time to complete, because it requires a lot of computation, or because it returns a lot of data which the client is slow to process. The solution is to create a separate process or thread to handle each request; the *ForkingMixIn* and *ThreadingMixIn* mix-in classes can be used to support asynchronous behaviour.

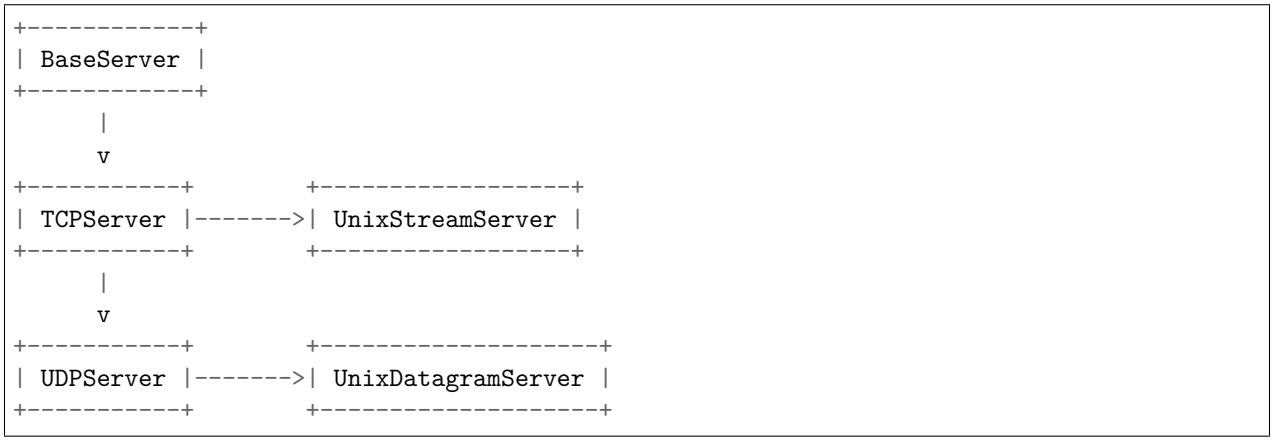
Creating a server requires several steps. First, you must create a request handler class by subclassing the *BaseRequestHandler* class and overriding its *handle()* method; this method will process incoming requests. Second, you must instantiate one of the server classes, passing it the server's address and the request handler class. It is recommended to use the server in a `with` statement. Then call the *handle_request()* or *serve_forever()* method of the server object to process one or many requests. Finally, call *server_close()* to close the socket (unless you used a `with` statement).

When inheriting from *ThreadingMixIn* for threaded connection behavior, you should explicitly declare how you want your threads to behave on an abrupt shutdown. The *ThreadingMixIn* class defines an attribute *daemon_threads*, which indicates whether or not the server should wait for thread termination. You should set the flag explicitly if you would like threads to behave autonomously; the default is *False*, meaning that Python will not exit until all threads created by *ThreadingMixIn* have exited.

Server classes have the same external methods and attributes, no matter what network protocol they use.

22.21.1 Server Creation Notes

There are five classes in an inheritance diagram, four of which represent synchronous servers of four types:



Note that *UnixDatagramServer* derives from *UDPServer*, not from *UnixStreamServer* — the only difference between an IP and a Unix stream server is the address family, which is simply repeated in both Unix server classes.

class socketserver.**ForkingMixIn**
class socketserver.**ThreadingMixIn**

Forking and threading versions of each type of server can be created using these mix-in classes. For instance, *ThreadingUDPServer* is created as follows:

```
class ThreadingUDPServer(ThreadingMixIn, UDPServer):
    pass
```

The mix-in class comes first, since it overrides a method defined in *UDPServer*. Setting the various attributes also changes the behavior of the underlying server mechanism.

ForkingMixIn and the Forking classes mentioned below are only available on POSIX platforms that support *fork()*.

`socketserver.ForkingMixIn.server_close()` waits until all child processes complete, except if `socketserver.ForkingMixIn.block_on_close` attribute is false.

`socketserver.ThreadingMixIn.server_close()` waits until all non-daemon threads complete, except if `socketserver.ThreadingMixIn.block_on_close` attribute is false. Use daemonic threads by setting `ThreadingMixIn.daemon_threads` to `True` to not wait until threads complete.

Changed in version 3.7: `socketserver.ForkingMixIn.server_close()` and `socketserver.ThreadingMixIn.server_close()` now waits until all child processes and non-daemonic threads complete. Add a new `socketserver.ForkingMixIn.block_on_close` class attribute to opt-in for the pre-3.7 behaviour.

class socketserver.**ForkingTCPServer**
class socketserver.**ForkingUDPServer**

class socketserver.ThreadingTCPServer
class socketserver.ThreadingUDPServer
These classes are pre-defined using the mix-in classes.

To implement a service, you must derive a class from *BaseRequestHandler* and redefine its *handle()* method. You can then run various versions of the service by combining one of the server classes with your request handler class. The request handler class must be different for datagram or stream services. This can be hidden by using the handler subclasses *StreamRequestHandler* or *DatagramRequestHandler*.

Of course, you still have to use your head! For instance, it makes no sense to use a forking server if the service contains state in memory that can be modified by different requests, since the modifications in the child process would never reach the initial state kept in the parent process and passed to each child. In this case, you can use a threading server, but you will probably have to use locks to protect the integrity of the shared data.

On the other hand, if you are building an HTTP server where all data is stored externally (for instance, in the file system), a synchronous class will essentially render the service "deaf" while one request is being handled – which may be for a very long time if a client is slow to receive all the data it has requested. Here a threading or forking server is appropriate.

In some cases, it may be appropriate to process part of a request synchronously, but to finish processing in a forked child depending on the request data. This can be implemented by using a synchronous server and doing an explicit fork in the request handler class *handle()* method.

Another approach to handling multiple simultaneous requests in an environment that supports neither threads nor *fork()* (or where these are too expensive or inappropriate for the service) is to maintain an explicit table of partially finished requests and to use *selectors* to decide which request to work on next (or whether to handle a new incoming request). This is particularly important for stream services where each client can potentially be connected for a long time (if threads or subprocesses cannot be used). See *asyncore* for another way to manage this.

22.21.2 Server Objects

class socketserver.BaseServer(*server_address*, *RequestHandlerClass*)
This is the superclass of all Server objects in the module. It defines the interface, given below, but does not implement most of the methods, which is done in subclasses. The two parameters are stored in the respective *server_address* and *RequestHandlerClass* attributes.

fileno()
Return an integer file descriptor for the socket on which the server is listening. This function is most commonly passed to *selectors*, to allow monitoring multiple servers in the same process.

handle_request()
Process a single request. This function calls the following methods in order: *get_request()*, *verify_request()*, and *process_request()*. If the user-provided *handle()* method of the handler class raises an exception, the server's *handle_error()* method will be called. If no request is received within *timeout* seconds, *handle_timeout()* will be called and *handle_request()* will return.

serve_forever(*poll_interval=0.5*)
Handle requests until an explicit *shutdown()* request. Poll for shutdown every *poll_interval* seconds. Ignores the *timeout* attribute. It also calls *service_actions()*, which may be used by a subclass or mixin to provide actions specific to a given service. For example, the *ForkingMixIn* class uses *service_actions()* to clean up zombie child processes.

Changed in version 3.3: Added `service_actions` call to the `serve_forever` method.

service_actions()
This is called in the *serve_forever()* loop. This method can be overridden by subclasses or mixin classes to perform actions specific to a given service, such as cleanup actions.

New in version 3.3.

shutdown()
Tell the *serve_forever()* loop to stop and wait until it does.

server_close()
Clean up the server. May be overridden.

address_family
The family of protocols to which the server's socket belongs. Common examples are *socket.AF_INET* and *socket.AF_UNIX*.

RequestHandlerClass
The user-provided request handler class; an instance of this class is created for each request.

server_address
The address on which the server is listening. The format of addresses varies depending on the protocol family; see the documentation for the *socket* module for details. For Internet protocols, this is a tuple containing a string giving the address, and an integer port number: `('127.0.0.1', 80)`, for example.

socket
The socket object on which the server will listen for incoming requests.

The server classes support the following class variables:

allow_reuse_address
Whether the server will allow the reuse of an address. This defaults to *False*, and can be set in subclasses to change the policy.

request_queue_size
The size of the request queue. If it takes a long time to process a single request, any requests that arrive while the server is busy are placed into a queue, up to *request_queue_size* requests. Once the queue is full, further requests from clients will get a "Connection denied" error. The default value is usually 5, but this can be overridden by subclasses.

socket_type
The type of socket used by the server; *socket.SOCK_STREAM* and *socket.SOCK_DGRAM* are two common values.

timeout
Timeout duration, measured in seconds, or *None* if no timeout is desired. If *handle_request()* receives no incoming requests within the timeout period, the *handle_timeout()* method is called.

There are various server methods that can be overridden by subclasses of base server classes like *TCPServer*; these methods aren't useful to external users of the server object.

finish_request(*request*, *client_address*)
Actually processes the request by instantiating *RequestHandlerClass* and calling its *handle()* method.

get_request()
Must accept a request from the socket, and return a 2-tuple containing the *new* socket object to be used to communicate with the client, and the client's address.

handle_error(*request*, *client_address*)
This function is called if the *handle()* method of a *RequestHandlerClass* instance raises an exception. The default action is to print the traceback to standard error and continue handling further requests.

Changed in version 3.6: Now only called for exceptions derived from the *Exception* class.

handle_timeout()
This function is called when the *timeout* attribute has been set to a value other than *None* and the timeout period has passed with no requests being received. The default action for forking servers is to collect the status of any child processes that have exited, while in threading servers this method does nothing.

process_request(*request*, *client_address*)
Calls *finish_request()* to create an instance of the *RequestHandlerClass*. If desired, this function can create a new process or thread to handle the request; the *ForkingMixIn* and *ThreadingMixIn* classes do this.

server_activate()
Called by the server's constructor to activate the server. The default behavior for a TCP server just invokes *listen()* on the server's socket. May be overridden.

server_bind()
Called by the server's constructor to bind the socket to the desired address. May be overridden.

verify_request(*request*, *client_address*)
Must return a Boolean value; if the value is *True*, the request will be processed, and if it's *False*, the request will be denied. This function can be overridden to implement access controls for a server. The default implementation always returns *True*.

Changed in version 3.6: Support for the *context manager* protocol was added. Exiting the context manager is equivalent to calling *server_close()*.

22.21.3 Request Handler Objects

class socketserver.BaseRequestHandler
This is the superclass of all request handler objects. It defines the interface, given below. A concrete request handler subclass must define a new *handle()* method, and can override any of the other methods. A new instance of the subclass is created for each request.

setup()
Called before the *handle()* method to perform any initialization actions required. The default implementation does nothing.

handle()
This function must do all the work required to service a request. The default implementation does nothing. Several instance attributes are available to it; the request is available as `self.request`; the client address as `self.client_address`; and the server instance as `self.server`, in case it needs access to per-server information.

The type of `self.request` is different for datagram or stream services. For stream services, `self.request` is a socket object; for datagram services, `self.request` is a pair of string and socket.

finish()
Called after the *handle()* method to perform any clean-up actions required. The default implementation does nothing. If *setup()* raises an exception, this function will not be called.

class socketserver.StreamRequestHandler
class socketserver.DatagramRequestHandler
These *BaseRequestHandler* subclasses override the *setup()* and *finish()* methods, and provide `self.rfile` and `self.wfile` attributes. The `self.rfile` and `self.wfile` attributes can be read or written, respectively, to get the request data or return data to the client.

The `rfile` attributes of both classes support the *io.BufferedIOBase* readable interface, and `DatagramRequestHandler.wfile` supports the *io.BufferedIOBase* writable interface.

Changed in version 3.6: `StreamRequestHandler.wfile` also supports the *io.BufferedIOBase* writable interface.

22.21.4 Examples

`socketserver.TCPServer` Example

This is the server side:

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    """
    The request handler class for our server.

    It is instantiated once per connection to the server, and must
    override the handle() method to implement communication to the
    client.
    """

    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(1024).strip()
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        # just send back the same data, but upper-cased
        self.request.sendall(self.data.upper())

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999

    # Create the server, binding to localhost on port 9999
    with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as server:
        # Activate the server; this will keep running until you
        # interrupt the program with Ctrl-C
        server.serve_forever()
```

An alternative request handler class that makes use of streams (file-like objects that simplify communication by providing the standard file interface):

```
class MyTCPHandler(socketserver.StreamRequestHandler):

    def handle(self):
        # self.rfile is a file-like object created by the handler;
        # we can now use e.g. readline() instead of raw recv() calls
        self.data = self.rfile.readline().strip()
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        # Likewise, self.wfile is a file-like object used to write back
        # to the client
        self.wfile.write(self.data.upper())
```

The difference is that the `readline()` call in the second handler will call `recv()` multiple times until it encounters a newline character, while the single `recv()` call in the first handler will just return what has been sent from the client in one `sendall()` call.

This is the client side:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# Create a socket (SOCK_STREAM means a TCP socket)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    # Connect to server and send data
    sock.connect((HOST, PORT))
    sock.sendall(bytes(data + "\n", "utf-8"))

    # Receive data from the server and shut down
    received = str(sock.recv(1024), "utf-8")

print("Sent:     {}".format(data))
print("Received: {}".format(received))
```

The output of the example should look something like this:

Server:

```
$ python TCPServer.py
127.0.0.1 wrote:
b'hello world with TCP'
127.0.0.1 wrote:
b'python is nice'
```

Client:

```
$ python TCPClient.py hello world with TCP
Sent:     hello world with TCP
Received: HELLO WORLD WITH TCP
$ python TCPClient.py python is nice
Sent:     python is nice
Received: PYTHON IS NICE
```

socketserver.UDPServer Example

This is the server side:

```
import socketserver

class MyUDPHandler(socketserver.BaseRequestHandler):
    """
    This class works similar to the TCP handler class, except that
    self.request consists of a pair of data and client socket, and since
    there is no connection the client address must be given explicitly
```

(continues on next page)

(continued from previous page)

```
    when sending data back via sendto().
    """

    def handle(self):
        data = self.request[0].strip()
        socket = self.request[1]
        print("{} wrote:".format(self.client_address[0]))
        print(data)
        socket.sendto(data.upper(), self.client_address)

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    with socketserver.UDPServer((HOST, PORT), MyUDPHandler) as server:
        server.serve_forever()
```

This is the client side:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# SOCK_DGRAM is the socket type to use for UDP sockets
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# As you can see, there is no connect() call; UDP has no connections.
# Instead, data is directly sent to the recipient via sendto().
sock.sendto(bytes(data + "\n", "utf-8"), (HOST, PORT))
received = str(sock.recv(1024), "utf-8")

print("Sent:     {}".format(data))
print("Received: {}".format(received))
```

The output of the example should look exactly like for the TCP server example.

Asynchronous Mixins

To build asynchronous handlers, use the *ThreadingMixIn* and *ForkingMixIn* classes.

An example for the *ThreadingMixIn* class:

```
import socket
import threading
import socketserver

class ThreadedTCPRequestHandler(socketserver.BaseRequestHandler):

    def handle(self):
        data = str(self.request.recv(1024), 'ascii')
        cur_thread = threading.current_thread()
        response = bytes("{}: {}".format(cur_thread.name, data), 'ascii')
```

(continues on next page)

(continued from previous page)

```
        self.request.sendall(response)

class ThreadedTCPServer(socketserver.ThreadingMixIn, socketserver.TCPServer):
    pass

def client(ip, port, message):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.connect((ip, port))
        sock.sendall(bytes(message, 'ascii'))
        response = str(sock.recv(1024), 'ascii')
        print("Received: {}".format(response))

if __name__ == "__main__":
    # Port 0 means to select an arbitrary unused port
    HOST, PORT = "localhost", 0

    server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)
    with server:
        ip, port = server.server_address

        # Start a thread with the server -- that thread will then start one
        # more thread for each request
        server_thread = threading.Thread(target=server.serve_forever)
        # Exit the server thread when the main thread terminates
        server_thread.daemon = True
        server_thread.start()
        print("Server loop running in thread:", server_thread.name)

        client(ip, port, "Hello World 1")
        client(ip, port, "Hello World 2")
        client(ip, port, "Hello World 3")

        server.shutdown()
```

The output of the example should look something like this:

```
$ python ThreadedTCPServer.py
Server loop running in thread: Thread-1
Received: Thread-2: Hello World 1
Received: Thread-3: Hello World 2
Received: Thread-4: Hello World 3
```

The *ForkingMixIn* class is used in the same way, except that the server will spawn a new process for each request. Available only on POSIX platforms that support *fork()*.

22.22 http.server — HTTP servers

Source code: Lib/http/server.py

This module defines classes for implementing HTTP servers (Web servers).

Warning: *http.server* is not recommended for production. It only implements basic security checks.

One class, *HTTPServer*, is a *socketserver.TCPServer* subclass. It creates and listens at the HTTP socket, dispatching the requests to a handler. Code to create and run the server looks like this:

```
def run(server_class=HTTPServer, handler_class=BaseHTTPRequestHandler):
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()
```

class http.server.**HTTPServer**(*server_address*, *RequestHandlerClass*)
This class builds on the *TCPServer* class by storing the server address as instance variables named `server_name` and `server_port`. The server is accessible by the handler, typically through the handler's `server` instance variable.

class http.server.**ThreadingHTTPServer**(*server_address*, *RequestHandlerClass*)
This class is identical to HTTPServer but uses threads to handle requests by using the *ThreadingMixIn*. This is useful to handle web browsers pre-opening sockets, on which *HTTPServer* would wait indefinitely.

New in version 3.7.

The *HTTPServer* and *ThreadingHTTPServer* must be given a *RequestHandlerClass* on instantiation, of which this module provides three different variants:

class http.server.**BaseHTTPRequestHandler**(*request*, *client_address*, *server*)
This class is used to handle the HTTP requests that arrive at the server. By itself, it cannot respond to any actual HTTP requests; it must be subclassed to handle each request method (e.g. GET or POST). *BaseHTTPRequestHandler* provides a number of class and instance variables, and methods for use by subclasses.

The handler will parse the request and the headers, then call a method specific to the request type. The method name is constructed from the request. For example, for the request method `SPAM`, the `do_SPAM()` method will be called with no arguments. All of the relevant information is stored in instance variables of the handler. Subclasses should not need to override or extend the `__init__()` method.

BaseHTTPRequestHandler has the following instance variables:

client_address
Contains a tuple of the form `(host, port)` referring to the client's address.

server
Contains the server instance.

close_connection
Boolean that should be set before *handle_one_request()* returns, indicating if another request may be expected, or if the connection should be shut down.

requestline
Contains the string representation of the HTTP request line. The terminating CRLF is stripped. This attribute should be set by *handle_one_request()*. If no valid request line was processed, it should be set to the empty string.

command
Contains the command (request type). For example, `'GET'`.

path
Contains the request path.

request_version
Contains the version string from the request. For example, `'HTTP/1.0'`.

headers
Holds an instance of the class specified by the *MessageClass* class variable. This instance parses and manages the headers in the HTTP request. The `parse_headers()` function from *http.client* is used to parse the headers and it requires that the HTTP request provide a valid **RFC 2822** style header.

rfile
An *io.BufferedIOBase* input stream, ready to read from the start of the optional input data.

wfile
Contains the output stream for writing a response back to the client. Proper adherence to the HTTP protocol must be used when writing to this stream in order to achieve successful interoperation with HTTP clients.

Changed in version 3.6: This is an *io.BufferedIOBase* stream.

BaseHTTPRequestHandler has the following attributes:

server_version
Specifies the server software version. You may want to override this. The format is multiple whitespace-separated strings, where each string is of the form name[/version]. For example, `'BaseHTTP/0.2'`.

sys_version
Contains the Python system version, in a form usable by the *version_string* method and the *server_version* class variable. For example, `'Python/1.4'`.

error_message_format
Specifies a format string that should be used by *send_error()* method for building an error response to the client. The string is filled by default with variables from *responses* based on the status code that passed to *send_error()*.

error_content_type
Specifies the Content-Type HTTP header of error responses sent to the client. The default value is `'text/html'`.

protocol_version
This specifies the HTTP protocol version used in responses. If set to `'HTTP/1.1'`, the server will permit HTTP persistent connections; however, your server *must* then include an accurate `Content-Length` header (using *send_header()*) in all of its responses to clients. For backwards compatibility, the setting defaults to `'HTTP/1.0'`.

MessageClass
Specifies an *email.message.Message*-like class to parse HTTP headers. Typically, this is not overridden, and it defaults to `http.client.HTTPMessage`.

responses
This attribute contains a mapping of error code integers to two-element tuples containing a short and long message. For example, `{code: (shortmessage, longmessage)}`. The *shortmessage* is usually used as the *message* key in an error response, and *longmessage* as the *explain* key. It is used by *send_response_only()* and *send_error()* methods.

A *BaseHTTPRequestHandler* instance has the following methods:

handle()
Calls *handle_one_request()* once (or, if persistent connections are enabled, multiple times) to handle incoming HTTP requests. You should never need to override it; instead, implement appropriate `do_*()` methods.

handle_one_request()
This method will parse and dispatch the request to the appropriate `do_*()` method. You should never need to override it.

handle_expect_100()
When a HTTP/1.1 compliant server receives an `Expect: 100-continue` request header it responds back with a `100 Continue` followed by `200 OK` headers. This method can be overridden to raise an error if the server does not want the client to continue. For e.g. server can chose to send `417 Expectation Failed` as a response header and `return False`.

New in version 3.2.

send_error(*code*, *message=None*, *explain=None*)
Sends and logs a complete error reply to the client. The numeric *code* specifies the HTTP error code, with *message* as an optional, short, human readable description of the error. The *explain* argument can be used to provide more detailed information about the error; it will be formatted using the *error_message_format* attribute and emitted, after a complete set of headers, as the response body. The *responses* attribute holds the default values for *message* and *explain* that will be used if no value is provided; for unknown codes the default value for both is the string `???`. The body will be empty if the method is HEAD or the response code is one of the following: `1xx`, `204 No Content`, `205 Reset Content`, `304 Not Modified`.

Changed in version 3.4: The error response includes a Content-Length header. Added the *explain* argument.

send_response(*code*, *message=None*)
Adds a response header to the headers buffer and logs the accepted request. The HTTP response line is written to the internal buffer, followed by *Server* and *Date* headers. The values for these two headers are picked up from the *version_string()* and *date_time_string()* methods, respectively. If the server does not intend to send any other headers using the *send_header()* method, then *send_response()* should be followed by an *end_headers()* call.

Changed in version 3.3: Headers are stored to an internal buffer and *end_headers()* needs to be called explicitly.

send_header(*keyword*, *value*)
Adds the HTTP header to an internal buffer which will be written to the output stream when either *end_headers()* or *flush_headers()* is invoked. *keyword* should specify the header keyword, with *value* specifying its value. Note that, after the send_header calls are done, *end_headers()* MUST BE called in order to complete the operation.

Changed in version 3.2: Headers are stored in an internal buffer.

send_response_only(*code*, *message=None*)
Sends the response header only, used for the purposes when `100 Continue` response is sent by the server to the client. The headers not buffered and sent directly the output stream.If the *message* is not specified, the HTTP message corresponding the response *code* is sent.

New in version 3.2.

end_headers()
Adds a blank line (indicating the end of the HTTP headers in the response) to the headers buffer and calls *flush_headers()*.

Changed in version 3.2: The buffered headers are written to the output stream.

flush_headers()
Finally send the headers to the output stream and flush the internal headers buffer.

New in version 3.3.

log_request(*code='-'*, *size='-'*)
Logs an accepted (successful) request. *code* should specify the numeric HTTP code associated with the response. If a size of the response is available, then it should be passed as the *size* parameter.

log_error(*...*)
Logs an error when a request cannot be fulfilled. By default, it passes the message to *log_message()*, so it takes the same arguments (*format* and additional values).

log_message(*format*, *...*)
Logs an arbitrary message to `sys.stderr`. This is typically overridden to create custom error logging mechanisms. The *format* argument is a standard printf-style format string, where the additional arguments to *log_message()* are applied as inputs to the formatting. The client ip address and current date and time are prefixed to every message logged.

version_string()
Returns the server software's version string. This is a combination of the *server_version* and *sys_version* attributes.

date_time_string(*timestamp=None*)
Returns the date and time given by *timestamp* (which must be `None` or in the format returned by *time.time()*), formatted for a message header. If *timestamp* is omitted, it uses the current date and time.

The result looks like `'Sun, 06 Nov 1994 08:49:37 GMT'`.

log_date_time_string()
Returns the current date and time, formatted for logging.

address_string()
Returns the client address.

Changed in version 3.3: Previously, a name lookup was performed. To avoid name resolution delays, it now always returns the IP address.

class http.server.SimpleHTTPRequestHandler(*request*, *client_address*, *server*, *directory=None*)
This class serves files from the current directory and below, directly mapping the directory structure to HTTP requests.

A lot of the work, such as parsing the request, is done by the base class *BaseHTTPRequestHandler*. This class implements the *do_GET()* and *do_HEAD()* functions.

The following are defined as class-level attributes of *SimpleHTTPRequestHandler*:

server_version
This will be `"SimpleHTTP/" + __version__`, where `__version__` is defined at the module level.

extensions_map
A dictionary mapping suffixes into MIME types. The default is signified by an empty string, and is considered to be `application/octet-stream`. The mapping is used case-insensitively, and so should contain only lower-cased keys.

directory
If not specified, the directory to serve is the current working directory.

The *SimpleHTTPRequestHandler* class defines the following methods:

do_HEAD()
This method serves the `'HEAD'` request type: it sends the headers it would send for the equivalent `GET` request. See the *do_GET()* method for a more complete explanation of the possible headers.

do_GET()
The request is mapped to a local file by interpreting the request as a path relative to the current working directory.

If the request was mapped to a directory, the directory is checked for a file named `index.html` or `index.htm` (in that order). If found, the file's contents are returned; otherwise a directory listing is generated by calling the `list_directory()` method. This method uses *os.listdir()* to scan the directory, and returns a `404` error response if the *listdir()* fails.

If the request was mapped to a file, it is opened. Any *OSError* exception in opening the requested file is mapped to a `404`, `'File not found'` error. If there was a `'If-Modified-Since'` header in the request, and the file was not modified after this time, a `304`, `'Not Modified'` response is sent. Otherwise, the content type is guessed by calling the `guess_type()` method, which in turn uses the *extensions_map* variable, and the file contents are returned.

A `'Content-type:'` header with the guessed content type is output, followed by a `'Content-Length:'` header with the file's size and a `'Last-Modified:'` header with the file's modification time.

Then follows a blank line signifying the end of the headers, and then the contents of the file are output. If the file's MIME type starts with `text/` the file is opened in text mode; otherwise binary mode is used.

For example usage, see the implementation of the *test()* function invocation in the *http.server* module.

Changed in version 3.7: Support of the `'If-Modified-Since'` header.

The *SimpleHTTPRequestHandler* class can be used in the following manner in order to create a very basic webserver serving files relative to the current directory:

```
import http.server
import socketserver

PORT = 8000

Handler = http.server.SimpleHTTPRequestHandler

with socketserver.TCPServer(("", PORT), Handler) as httpd:
    print("serving at port", PORT)
    httpd.serve_forever()
```

http.server can also be invoked directly using the `-m` switch of the interpreter with a `port number` argument. Similar to the previous example, this serves files relative to the current directory:

```
python -m http.server 8000
```

By default, server binds itself to all interfaces. The option `-b/--bind` specifies a specific address to which it should bind. For example, the following command causes the server to bind to localhost only:

```
python -m http.server 8000 --bind 127.0.0.1
```

New in version 3.4: `--bind` argument was introduced.

By default, server uses the current directory. The option `-d/--directory` specifies a directory to which it should serve the files. For example, the following command uses a specific directory:

```
python -m http.server --directory /tmp/
```

New in version 3.7: `--directory` specify alternate directory

class `http.server.CGIHTTPRequestHandler`(*request*, *client_address*, *server*)

This class is used to serve either files or output of CGI scripts from the current directory and below. Note that mapping HTTP hierarchic structure to local directory structure is exactly as in *SimpleHTTPRequestHandler*.

Note: CGI scripts run by the *CGIHTTPRequestHandler* class cannot execute redirects (HTTP code 302), because code 200 (script output follows) is sent prior to execution of the CGI script. This

pre-empts the status code.

The class will however, run the CGI script, instead of serving it as a file, if it guesses it to be a CGI script. Only directory-based CGI are used — the other common server configuration is to treat special extensions as denoting CGI scripts.

The `do_GET()` and `do_HEAD()` functions are modified to run CGI scripts and serve the output, instead of serving files, if the request leads to somewhere below the `cgi_directories` path.

The *CGIHTTPRequestHandler* defines the following data member:

`cgi_directories`
: This defaults to `['/cgi-bin', '/htbin']` and describes directories to treat as containing CGI scripts.

The *CGIHTTPRequestHandler* defines the following method:

`do_POST()`
: This method serves the `'POST'` request type, only allowed for CGI scripts. Error 501, "Can only POST to CGI scripts", is output when trying to POST to a non-CGI url.

Note that CGI scripts will be run with UID of user nobody, for security reasons. Problems with the CGI script will be translated to error 403.

CGIHTTPRequestHandler can be enabled in the command line by passing the `--cgi` option:

```
python -m http.server --cgi 8000
```

22.23 `http.cookies` — HTTP state management

Source code: Lib/http/cookies.py

The *http.cookies* module defines classes for abstracting the concept of cookies, an HTTP state management mechanism. It supports both simple string-only cookies, and provides an abstraction for having any serializable data-type as cookie value.

The module formerly strictly applied the parsing rules described in the **RFC 2109** and **RFC 2068** specifications. It has since been discovered that MSIE 3.0x doesn't follow the character rules outlined in those specs and also many current day browsers and servers have relaxed parsing rules when comes to Cookie handling. As a result, the parsing rules used are a bit less strict.

The character set, *string.ascii_letters*, *string.digits* and `!#$%&'*+-.^_`|~:` denote the set of valid characters allowed by this module in Cookie name (as *key*).

Changed in version 3.3: Allowed ':' as a valid Cookie name character.

Note: On encountering an invalid cookie, *CookieError* is raised, so if your cookie data comes from a browser you should always prepare for invalid data and catch *CookieError* on parsing.

`exception http.cookies.CookieError`
: Exception failing because of **RFC 2109** invalidity: incorrect attributes, incorrect *Set-Cookie* header, etc.

`class http.cookies.BaseCookie(`[*input*]`)`
: This class is a dictionary-like object whose keys are strings and whose values are *Morsel* instances.

Note that upon setting a key to a value, the value is first converted to a *Morsel* containing the key and the value.

If *input* is given, it is passed to the *load()* method.

class http.cookies.**SimpleCookie**([*input*])
This class derives from *BaseCookie* and overrides `value_decode()` and `value_encode()` to be the identity and *str()* respectively.

See also:

Module *http.cookiejar* HTTP cookie handling for web *clients*. The *http.cookiejar* and *http.cookies* modules do not depend on each other.

RFC 2109 - HTTP State Management Mechanism This is the state management specification implemented by this module.

22.23.1 Cookie Objects

BaseCookie.**value_decode**(*val*)
Return a decoded value from a string representation. Return value can be any type. This method does nothing in *BaseCookie* — it exists so it can be overridden.

BaseCookie.**value_encode**(*val*)
Return an encoded value. *val* can be any type, but return value must be a string. This method does nothing in *BaseCookie* — it exists so it can be overridden.

In general, it should be the case that *value_encode()* and *value_decode()* are inverses on the range of *value_decode*.

BaseCookie.**output**(*attrs=None*, *header='Set-Cookie:'*, *sep='\r\n'*)
Return a string representation suitable to be sent as HTTP headers. *attrs* and *header* are sent to each *Morsel*'s *output()* method. *sep* is used to join the headers together, and is by default the combination `'\r\n'` (CRLF).

BaseCookie.**js_output**(*attrs=None*)
Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP headers was sent.

The meaning for *attrs* is the same as in *output()*.

BaseCookie.**load**(*rawdata*)
If *rawdata* is a string, parse it as an `HTTP_COOKIE` and add the values found there as *Morsel*s. If it is a dictionary, it is equivalent to:

```
for k, v in rawdata.items():
    cookie[k] = v
```

22.23.2 Morsel Objects

class http.cookies.**Morsel**
Abstract a key/value pair, which has some **RFC 2109** attributes.

Morsels are dictionary-like objects, whose set of keys is constant — the valid **RFC 2109** attributes, which are

- `expires`
- `path`
- `comment`

- domain
- max-age
- secure
- version
- httponly

The attribute `httponly` specifies that the cookie is only transferred in HTTP requests, and is not accessible through JavaScript. This is intended to mitigate some forms of cross-site scripting.

The keys are case-insensitive and their default value is `''`.

Changed in version 3.5: `__eq__()` now takes *key* and *value* into account.

Changed in version 3.7: Attributes *key*, *value* and *coded_value* are read-only. Use *set()* for setting them.

Morsel.value
The value of the cookie.

Morsel.coded_value
The encoded value of the cookie — this is what should be sent.

Morsel.key
The name of the cookie.

Morsel.set(*key*, *value*, *coded_value*)
Set the *key*, *value* and *coded_value* attributes.

Morsel.isReservedKey(*K*)
Whether *K* is a member of the set of keys of a *Morsel*.

Morsel.output(*attrs=None*, *header='Set-Cookie:'*)
Return a string representation of the Morsel, suitable to be sent as an HTTP header. By default, all the attributes are included, unless *attrs* is given, in which case it should be a list of attributes to use. *header* is by default `"Set-Cookie:"`.

Morsel.js_output(*attrs=None*)
Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP header was sent.

The meaning for *attrs* is the same as in *output()*.

Morsel.OutputString(*attrs=None*)
Return a string representing the Morsel, without any surrounding HTTP or JavaScript.

The meaning for *attrs* is the same as in *output()*.

Morsel.update(*values*)
Update the values in the Morsel dictionary with the values in the dictionary *values*. Raise an error if any of the keys in the *values* dict is not a valid **RFC 2109** attribute.

Changed in version 3.5: an error is raised for invalid keys.

Morsel.copy(*value*)
Return a shallow copy of the Morsel object.

Changed in version 3.5: return a Morsel object instead of a dict.

Morsel.setdefault(*key*, *value=None*)
Raise an error if key is not a valid **RFC 2109** attribute, otherwise behave the same as *dict.setdefault()*.

22.23.3 Example

The following example demonstrates how to use the *http.cookies* module.

```
>>> from http import cookies
>>> C = cookies.SimpleCookie()
>>> C["fig"] = "newton"
>>> C["sugar"] = "wafer"
>>> print(C) # generate HTTP headers
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> print(C.output()) # same thing
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> C = cookies.SimpleCookie()
>>> C["rocky"] = "road"
>>> C["rocky"]["path"] = "/cookie"
>>> print(C.output(header="Cookie:"))
Cookie: rocky=road; Path=/cookie
>>> print(C.output(attrs=[], header="Cookie:"))
Cookie: rocky=road
>>> C = cookies.SimpleCookie()
>>> C.load("chips=ahoy; vienna=finger") # load from a string (HTTP header)
>>> print(C)
Set-Cookie: chips=ahoy
Set-Cookie: vienna=finger
>>> C = cookies.SimpleCookie()
>>> C.load('keebler="E=everybody; L=\\"Loves\\"; fudge=\\012;";')
>>> print(C)
Set-Cookie: keebler="E=everybody; L=\"Loves\"; fudge=\012;"
>>> C = cookies.SimpleCookie()
>>> C["oreo"] = "doublestuff"
>>> C["oreo"]["path"] = "/"
>>> print(C)
Set-Cookie: oreo=doublestuff; Path=/
>>> C = cookies.SimpleCookie()
>>> C["twix"] = "none for you"
>>> C["twix"].value
'none for you'
>>> C = cookies.SimpleCookie()
>>> C["number"] = 7 # equivalent to C["number"] = str(7)
>>> C["string"] = "seven"
>>> C["number"].value
'7'
>>> C["string"].value
'seven'
>>> print(C)
Set-Cookie: number=7
Set-Cookie: string=seven
```

22.24 http.cookiejar — Cookie handling for HTTP clients

Source code: Lib/http/cookiejar.py

The *http.cookiejar* module defines classes for automatic handling of HTTP cookies. It is useful for accessing web sites that require small pieces of data – *cookies* – to be set on the client machine by an HTTP response from a web server, and then returned to the server in later HTTP requests.

Both the regular Netscape cookie protocol and the protocol defined by **RFC 2965** are handled. RFC 2965 handling is switched off by default. **RFC 2109** cookies are parsed as Netscape cookies and subsequently treated either as Netscape or RFC 2965 cookies according to the 'policy' in effect. Note that the great majority of cookies on the Internet are Netscape cookies. *http.cookiejar* attempts to follow the de-facto Netscape cookie protocol (which differs substantially from that set out in the original Netscape specification), including taking note of the `max-age` and `port` cookie-attributes introduced with RFC 2965.

Note: The various named parameters found in *Set-Cookie* and *Set-Cookie2* headers (eg. `domain` and `expires`) are conventionally referred to as *attributes*. To distinguish them from Python attributes, the documentation for this module uses the term *cookie-attribute* instead.

The module defines the following exception:

exception http.cookiejar.LoadError
Instances of *FileCookieJar* raise this exception on failure to load cookies from a file. *LoadError* is a subclass of *OSError*.

Changed in version 3.3: LoadError was made a subclass of *OSError* instead of *IOError*.

The following classes are provided:

class http.cookiejar.CookieJar(*policy=None*)
policy is an object implementing the *CookiePolicy* interface.

The *CookieJar* class stores HTTP cookies. It extracts cookies from HTTP requests, and returns them in HTTP responses. *CookieJar* instances automatically expire contained cookies when necessary. Subclasses are also responsible for storing and retrieving cookies from a file or database.

class http.cookiejar.FileCookieJar(*filename*, *delayload=None*, *policy=None*)
policy is an object implementing the *CookiePolicy* interface. For the other arguments, see the documentation for the corresponding attributes.

A *CookieJar* which can load cookies from, and perhaps save cookies to, a file on disk. Cookies are **NOT** loaded from the named file until either the *load()* or *revert()* method is called. Subclasses of this class are documented in section *FileCookieJar subclasses and co-operation with web browsers*.

class http.cookiejar.CookiePolicy
This class is responsible for deciding whether each cookie should be accepted from / returned to the server.

class http.cookiejar.DefaultCookiePolicy(*blocked_domains=None*, *allowed_domains=None*, *netscape=True*, *rfc2965=False*, *rfc2109_as_netscape=None*, *hide_cookie2=False*, *strict_domain=False*, *strict_rfc2965_unverifiable=True*, *strict_ns_unverifiable=False*, *strict_ns_domain=DefaultCookiePolicy.DomainLiberal*, *strict_ns_set_initial_dollar=False*, *strict_ns_set_path=False*)
Constructor arguments should be passed as keyword arguments only. *blocked_domains* is a sequence

of domain names that we never accept cookies from, nor return cookies to. *allowed_domains* if not *None*, this is a sequence of the only domains for which we accept and return cookies. For all other arguments, see the documentation for *CookiePolicy* and *DefaultCookiePolicy* objects.

DefaultCookiePolicy implements the standard accept / reject rules for Netscape and **RFC 2965** cookies. By default, **RFC 2109** cookies (ie. cookies received in a *Set-Cookie* header with a version cookie-attribute of 1) are treated according to the RFC 2965 rules. However, if RFC 2965 handling is turned off or *rfc2109_as_netscape* is `True`, RFC 2109 cookies are 'downgraded' by the *CookieJar* instance to Netscape cookies, by setting the `version` attribute of the *Cookie* instance to 0. *DefaultCookiePolicy* also provides some parameters to allow some fine-tuning of policy.

`class http.cookiejar.Cookie`
This class represents Netscape, **RFC 2109** and **RFC 2965** cookies. It is not expected that users of *http.cookiejar* construct their own *Cookie* instances. Instead, if necessary, call `make_cookies()` on a *CookieJar* instance.

See also:

Module *urllib.request* URL opening with automatic cookie handling.

Module *http.cookies* HTTP cookie classes, principally useful for server-side code. The *http.cookiejar* and *http.cookies* modules do not depend on each other.

https://curl.haxx.se/rfc/cookie_spec.html The specification of the original Netscape cookie protocol. Though this is still the dominant protocol, the 'Netscape cookie protocol' implemented by all the major browsers (and *http.cookiejar*) only bears a passing resemblance to the one sketched out in `cookie_spec.html`.

RFC 2109 - HTTP State Management Mechanism Obsoleted by **RFC 2965**. Uses *Set-Cookie* with version=1.

RFC 2965 - HTTP State Management Mechanism The Netscape protocol with the bugs fixed. Uses *Set-Cookie2* in place of *Set-Cookie*. Not widely used.

http://kristol.org/cookie/errata.html Unfinished errata to **RFC 2965**.

RFC 2964 - Use of HTTP State Management

22.24.1 CookieJar and FileCookieJar Objects

CookieJar objects support the *iterator* protocol for iterating over contained *Cookie* objects.

CookieJar has the following methods:

`CookieJar.add_cookie_header(`*request*`)`
Add correct *Cookie* header to *request*.

If policy allows (ie. the `rfc2965` and `hide_cookie2` attributes of the *CookieJar*'s *CookiePolicy* instance are true and false respectively), the *Cookie2* header is also added when appropriate.

The *request* object (usually a `urllib.request..Request` instance) must support the methods `get_full_url()`, `get_host()`, `get_type()`, `unverifiable()`, `has_header()`, `get_header()`, `header_items()`, `add_unredirected_header()` and `origin_req_host` attribute as documented by *urllib.request*.

Changed in version 3.3: *request* object needs `origin_req_host` attribute. Dependency on a deprecated method `get_origin_req_host()` has been removed.

`CookieJar.extract_cookies(`*response*, *request*`)`
Extract cookies from HTTP *response* and store them in the *CookieJar*, where allowed by policy.

The *CookieJar* will look for allowable *Set-Cookie* and *Set-Cookie2* headers in the *response* argument, and store cookies as appropriate (subject to the *CookiePolicy.set_ok()* method's approval).

The *response* object (usually the result of a call to *urllib.request.urlopen()*, or similar) should support an `info()` method, which returns an *email.message.Message* instance.

The *request* object (usually a *urllib.request.Request* instance) must support the methods `get_full_url()`, `get_host()`, `unverifiable()`, and `origin_req_host` attribute, as documented by *urllib.request*. The request is used to set default values for cookie-attributes as well as for checking that the cookie is allowed to be set.

Changed in version 3.3: *request* object needs `origin_req_host` attribute. Dependency on a deprecated method `get_origin_req_host()` has been removed.

`CookieJar.set_policy`(*policy*)
Set the *CookiePolicy* instance to be used.

`CookieJar.make_cookies`(*response*, *request*)
Return sequence of *Cookie* objects extracted from *response* object.

See the documentation for *extract_cookies()* for the interfaces required of the *response* and *request* arguments.

`CookieJar.set_cookie_if_ok`(*cookie*, *request*)
Set a *Cookie* if policy says it's OK to do so.

`CookieJar.set_cookie`(*cookie*)
Set a *Cookie*, without checking with policy to see whether or not it should be set.

`CookieJar.clear`(*domain*[, *path*[, *name*]])
Clear some cookies.

If invoked without arguments, clear all cookies. If given a single argument, only cookies belonging to that *domain* will be removed. If given two arguments, cookies belonging to the specified *domain* and URL *path* are removed. If given three arguments, then the cookie with the specified *domain*, *path* and *name* is removed.

Raises *KeyError* if no matching cookie exists.

`CookieJar.clear_session_cookies`()
Discard all session cookies.

Discards all contained cookies that have a true `discard` attribute (usually because they had either no `max-age` or `expires` cookie-attribute, or an explicit `discard` cookie-attribute). For interactive browsers, the end of a session usually corresponds to closing the browser window.

Note that the `save()` method won't save session cookies anyway, unless you ask otherwise by passing a true *ignore_discard* argument.

FileCookieJar implements the following additional methods:

`FileCookieJar.save`(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)
Save cookies to a file.

This base class raises *NotImplementedError*. Subclasses may leave this method unimplemented.

filename is the name of file in which to save cookies. If *filename* is not specified, `self.filename` is used (whose default is the value passed to the constructor, if any); if `self.filename` is *None*, *ValueError* is raised.

ignore_discard: save even cookies set to be discarded. *ignore_expires*: save even cookies that have expired

The file is overwritten if it already exists, thus wiping all the cookies it contains. Saved cookies can be restored later using the *load()* or *revert()* methods.

`FileCookieJar.load`(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)
Load cookies from a file.

Old cookies are kept unless overwritten by newly loaded ones.

Arguments are as for *save()*.

The named file must be in the format understood by the class, or *LoadError* will be raised. Also, *OSError* may be raised, for example if the file does not exist.

Changed in version 3.3: *IOError* used to be raised, it is now an alias of *OSError*.

`FileCookieJar.revert`(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)
Clear all cookies and reload cookies from a saved file.

revert() can raise the same exceptions as *load()*. If there is a failure, the object's state will not be altered.

FileCookieJar instances have the following public attributes:

`FileCookieJar.filename`
Filename of default file in which to keep cookies. This attribute may be assigned to.

`FileCookieJar.delayload`
If true, load cookies lazily from disk. This attribute should not be assigned to. This is only a hint, since this only affects performance, not behaviour (unless the cookies on disk are changing). A *CookieJar* object may ignore it. None of the *FileCookieJar* classes included in the standard library lazily loads cookies.

22.24.2 FileCookieJar subclasses and co-operation with web browsers

The following *CookieJar* subclasses are provided for reading and writing.

`class http.cookiejar.MozillaCookieJar`(*filename*, *delayload=None*, *policy=None*)
A *FileCookieJar* that can load from and save cookies to disk in the Mozilla `cookies.txt` file format (which is also used by the Lynx and Netscape browsers).

Note: This loses information about **RFC 2965** cookies, and also about newer or non-standard cookie-attributes such as `port`.

Warning: Back up your cookies before saving if you have cookies whose loss / corruption would be inconvenient (there are some subtleties which may lead to slight changes in the file over a load / save round-trip).

Also note that cookies saved while Mozilla is running will get clobbered by Mozilla.

`class http.cookiejar.LWPCookieJar`(*filename*, *delayload=None*, *policy=None*)
A *FileCookieJar* that can load from and save cookies to disk in format compatible with the libwww-perl library's `Set-Cookie3` file format. This is convenient if you want to store cookies in a human-readable file.

22.24.3 CookiePolicy Objects

Objects implementing the *CookiePolicy* interface have the following methods:

`CookiePolicy.set_ok`(*cookie*, *request*)
Return boolean value indicating whether cookie should be accepted from server.

cookie is a *Cookie* instance. *request* is an object implementing the interface defined by the documentation for *CookieJar.extract_cookies()*.

CookiePolicy.**return_ok**(*cookie*, *request*)
Return boolean value indicating whether cookie should be returned to server.

cookie is a *Cookie* instance. *request* is an object implementing the interface defined by the documentation for *CookieJar.add_cookie_header()*.

CookiePolicy.**domain_return_ok**(*domain*, *request*)
Return false if cookies should not be returned, given cookie domain.

This method is an optimization. It removes the need for checking every cookie with a particular domain (which might involve reading many files). Returning true from *domain_return_ok()* and *path_return_ok()* leaves all the work to *return_ok()*.

If *domain_return_ok()* returns true for the cookie domain, *path_return_ok()* is called for the cookie path. Otherwise, *path_return_ok()* and *return_ok()* are never called for that cookie domain. If *path_return_ok()* returns true, *return_ok()* is called with the *Cookie* object itself for a full check. Otherwise, *return_ok()* is never called for that cookie path.

Note that *domain_return_ok()* is called for every *cookie* domain, not just for the *request* domain. For example, the function might be called with both `".example.com"` and `"www.example.com"` if the request domain is `"www.example.com"`. The same goes for *path_return_ok()*.

The *request* argument is as documented for *return_ok()*.

CookiePolicy.**path_return_ok**(*path*, *request*)
Return false if cookies should not be returned, given cookie path.

See the documentation for *domain_return_ok()*.

In addition to implementing the methods above, implementations of the *CookiePolicy* interface must also supply the following attributes, indicating which protocols should be used, and how. All of these attributes may be assigned to.

CookiePolicy.**netscape**
Implement Netscape protocol.

CookiePolicy.**rfc2965**
Implement **RFC 2965** protocol.

CookiePolicy.**hide_cookie2**
Don't add *Cookie2* header to requests (the presence of this header indicates to the server that we understand **RFC 2965** cookies).

The most useful way to define a *CookiePolicy* class is by subclassing from *DefaultCookiePolicy* and overriding some or all of the methods above. *CookiePolicy* itself may be used as a 'null policy' to allow setting and receiving any and all cookies (this is unlikely to be useful).

22.24.4 DefaultCookiePolicy Objects

Implements the standard rules for accepting and returning cookies.

Both **RFC 2965** and Netscape cookies are covered. RFC 2965 handling is switched off by default.

The easiest way to provide your own policy is to override this class and call its methods in your overridden implementations before adding your own additional checks:

```
import http.cookiejar
class MyCookiePolicy(http.cookiejar.DefaultCookiePolicy):
    def set_ok(self, cookie, request):
        if not http.cookiejar.DefaultCookiePolicy.set_ok(self, cookie, request):
            return False
```

(continues on next page)

(continued from previous page)

```
        if i_dont_want_to_store_this_cookie(cookie):
            return False
        return True
```

In addition to the features required to implement the *CookiePolicy* interface, this class allows you to block and allow domains from setting and receiving cookies. There are also some strictness switches that allow you to tighten up the rather loose Netscape protocol rules a little bit (at the cost of blocking some benign cookies).

A domain blacklist and whitelist is provided (both off by default). Only domains not in the blacklist and present in the whitelist (if the whitelist is active) participate in cookie setting and returning. Use the *blocked_domains* constructor argument, and `blocked_domains()` and `set_blocked_domains()` methods (and the corresponding argument and methods for *allowed_domains*). If you set a whitelist, you can turn it off again by setting it to *None*.

Domains in block or allow lists that do not start with a dot must equal the cookie domain to be matched. For example, `"example.com"` matches a blacklist entry of `"example.com"`, but `"www.example.com"` does not. Domains that do start with a dot are matched by more specific domains too. For example, both `"www.example.com"` and `"www.coyote.example.com"` match `".example.com"` (but `"example.com"` itself does not). IP addresses are an exception, and must match exactly. For example, if blocked_domains contains `"192.168.1.2"` and `".168.1.2"`, 192.168.1.2 is blocked, but 193.168.1.2 is not.

DefaultCookiePolicy implements the following additional methods:

`DefaultCookiePolicy.blocked_domains()`
Return the sequence of blocked domains (as a tuple).

`DefaultCookiePolicy.set_blocked_domains(`*blocked_domains*`)`
Set the sequence of blocked domains.

`DefaultCookiePolicy.is_blocked(`*domain*`)`
Return whether *domain* is on the blacklist for setting or receiving cookies.

`DefaultCookiePolicy.allowed_domains()`
Return *None*, or the sequence of allowed domains (as a tuple).

`DefaultCookiePolicy.set_allowed_domains(`*allowed_domains*`)`
Set the sequence of allowed domains, or *None*.

`DefaultCookiePolicy.is_not_allowed(`*domain*`)`
Return whether *domain* is not on the whitelist for setting or receiving cookies.

DefaultCookiePolicy instances have the following attributes, which are all initialised from the constructor arguments of the same name, and which may all be assigned to.

`DefaultCookiePolicy.rfc2109_as_netscape`
If true, request that the *CookieJar* instance downgrade **RFC 2109** cookies (ie. cookies received in a *Set-Cookie* header with a version cookie-attribute of 1) to Netscape cookies by setting the version attribute of the *Cookie* instance to 0. The default value is *None*, in which case RFC 2109 cookies are downgraded if and only if **RFC 2965** handling is turned off. Therefore, RFC 2109 cookies are downgraded by default.

General strictness switches:

`DefaultCookiePolicy.strict_domain`
Don't allow sites to set two-component domains with country-code top-level domains like `.co.uk`, `.gov.uk`, `.co.nz`.etc. This is far from perfect and isn't guaranteed to work!

RFC 2965 protocol strictness switches:

`DefaultCookiePolicy.strict_rfc2965_unverifiable`
: Follow **RFC 2965** rules on unverifiable transactions (usually, an unverifiable transaction is one resulting from a redirect or a request for an image hosted on another site). If this is false, cookies are *never* blocked on the basis of verifiability

Netscape protocol strictness switches:

`DefaultCookiePolicy.strict_ns_unverifiable`
: Apply **RFC 2965** rules on unverifiable transactions even to Netscape cookies.

`DefaultCookiePolicy.strict_ns_domain`
: Flags indicating how strict to be with domain-matching rules for Netscape cookies. See below for acceptable values.

`DefaultCookiePolicy.strict_ns_set_initial_dollar`
: Ignore cookies in Set-Cookie: headers that have names starting with `'$'`.

`DefaultCookiePolicy.strict_ns_set_path`
: Don't allow setting cookies whose path doesn't path-match request URI.

`strict_ns_domain` is a collection of flags. Its value is constructed by or-ing together (for example, `DomainStrictNoDots|DomainStrictNonDomain` means both flags are set).

`DefaultCookiePolicy.DomainStrictNoDots`
: When setting cookies, the 'host prefix' must not contain a dot (eg. `www.foo.bar.com` can't set a cookie for `.bar.com`, because `www.foo` contains a dot).

`DefaultCookiePolicy.DomainStrictNonDomain`
: Cookies that did not explicitly specify a `domain` cookie-attribute can only be returned to a domain equal to the domain that set the cookie (eg. `spam.example.com` won't be returned cookies from `example.com` that had no `domain` cookie-attribute).

`DefaultCookiePolicy.DomainRFC2965Match`
: When setting cookies, require a full **RFC 2965** domain-match.

The following attributes are provided for convenience, and are the most useful combinations of the above flags:

`DefaultCookiePolicy.DomainLiberal`
: Equivalent to 0 (ie. all of the above Netscape domain strictness flags switched off).

`DefaultCookiePolicy.DomainStrict`
: Equivalent to `DomainStrictNoDots|DomainStrictNonDomain`.

22.24.5 Cookie Objects

Cookie instances have Python attributes roughly corresponding to the standard cookie-attributes specified in the various cookie standards. The correspondence is not one-to-one, because there are complicated rules for assigning default values, because the `max-age` and `expires` cookie-attributes contain equivalent information, and because **RFC 2109** cookies may be 'downgraded' by *http.cookiejar* from version 1 to version 0 (Netscape) cookies.

Assignment to these attributes should not be necessary other than in rare circumstances in a *CookiePolicy* method. The class does not enforce internal consistency, so you should know what you're doing if you do that.

`Cookie.version`
: Integer or *None*. Netscape cookies have *version* 0. **RFC 2965** and **RFC 2109** cookies have a `version` cookie-attribute of 1. However, note that *http.cookiejar* may 'downgrade' RFC 2109 cookies to Netscape cookies, in which case *version* is 0.

Cookie.name
Cookie name (a string).

Cookie.value
Cookie value (a string), or *None*.

Cookie.port
String representing a port or a set of ports (eg. '80', or '80,8080'), or *None*.

Cookie.path
Cookie path (a string, eg. `'/acme/rocket_launchers'`).

Cookie.secure
`True` if cookie should only be returned over a secure connection.

Cookie.expires
Integer expiry date in seconds since epoch, or *None*. See also the *is_expired()* method.

Cookie.discard
`True` if this is a session cookie.

Cookie.comment
String comment from the server explaining the function of this cookie, or *None*.

Cookie.comment_url
URL linking to a comment from the server explaining the function of this cookie, or *None*.

Cookie.rfc2109
`True` if this cookie was received as an **RFC 2109** cookie (ie. the cookie arrived in a *Set-Cookie* header, and the value of the Version cookie-attribute in that header was 1). This attribute is provided because *http.cookiejar* may 'downgrade' RFC 2109 cookies to Netscape cookies, in which case *version* is 0.

Cookie.port_specified
`True` if a port or set of ports was explicitly specified by the server (in the *Set-Cookie* / *Set-Cookie2* header).

Cookie.domain_specified
`True` if a domain was explicitly specified by the server.

Cookie.domain_initial_dot
`True` if the domain explicitly specified by the server began with a dot (`'.'`).

Cookies may have additional non-standard cookie-attributes. These may be accessed using the following methods:

Cookie.has_nonstandard_attr(*name*)
Return true if cookie has the named cookie-attribute.

Cookie.get_nonstandard_attr(*name*, *default=None*)
If cookie has the named cookie-attribute, return its value. Otherwise, return *default*.

Cookie.set_nonstandard_attr(*name*, *value*)
Set the value of the named cookie-attribute.

The *Cookie* class also defines the following method:

Cookie.is_expired(*now=None*)
`True` if cookie has passed the time at which the server requested it should expire. If *now* is given (in seconds since the epoch), return whether the cookie has expired at the specified time.

22.24.6 Examples

The first example shows the most common usage of *http.cookiejar*:

```
import http.cookiejar, urllib.request
cj = http.cookiejar.CookieJar()
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

This example illustrates how to open a URL using your Netscape, Mozilla, or Lynx cookies (assumes Unix/Netscape convention for location of the cookies file):

```
import os, http.cookiejar, urllib.request
cj = http.cookiejar.MozillaCookieJar()
cj.load(os.path.join(os.path.expanduser("~"), ".netscape", "cookies.txt"))
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

The next example illustrates the use of *DefaultCookiePolicy*. Turn on **RFC 2965** cookies, be more strict about domains when setting and returning Netscape cookies, and block some domains from setting cookies or having them returned:

```
import urllib.request
from http.cookiejar import CookieJar, DefaultCookiePolicy
policy = DefaultCookiePolicy(
    rfc2965=True, strict_ns_domain=Policy.DomainStrict,
    blocked_domains=["ads.net", ".ads.net"])
cj = CookieJar(policy)
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

22.25 `xmlrpc` — XMLRPC server and client modules

XML-RPC is a Remote Procedure Call method that uses XML passed via HTTP as a transport. With it, a client can call methods with parameters on a remote server (the server is named by a URI) and get back structured data.

`xmlrpc` is a package that collects server and client modules implementing XML-RPC. The modules are:

- *xmlrpc.client*
- *xmlrpc.server*

22.26 `xmlrpc.client` — XML-RPC client access

Source code: Lib/xmlrpc/client.py

XML-RPC is a Remote Procedure Call method that uses XML passed via HTTP(S) as a transport. With it, a client can call methods with parameters on a remote server (the server is named by a URI) and get back structured data. This module supports writing XML-RPC client code; it handles all the details of translating between conformable Python objects and XML on the wire.

Warning: The *xmlrpc.client* module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML vulnerabilities*.

Changed in version 3.5: For HTTPS URIs, *xmlrpc.client* now performs all the necessary certificate and hostname checks by default.

class xmlrpc.client.ServerProxy(*uri*, *transport=None*, *encoding=None*, *verbose=False*, *allow_none=False*, *use_datetime=False*, *use_builtin_types=False*, ***, *context=None*)

Changed in version 3.3: The *use_builtin_types* flag was added.

A *ServerProxy* instance is an object that manages communication with a remote XML-RPC server. The required first argument is a URI (Uniform Resource Indicator), and will normally be the URL of the server. The optional second argument is a transport factory instance; by default it is an internal `SafeTransport` instance for https: URLs and an internal HTTP `Transport` instance otherwise. The optional third argument is an encoding, by default UTF-8. The optional fourth argument is a debugging flag.

The following parameters govern the use of the returned proxy instance. If *allow_none* is true, the Python constant `None` will be translated into XML; the default behaviour is for `None` to raise a *TypeError*. This is a commonly-used extension to the XML-RPC specification, but isn't supported by all clients and servers; see http://ontosys.com/xml-rpc/extensions.php for a description. The *use_builtin_types* flag can be used to cause date/time values to be presented as *datetime.datetime* objects and binary data to be presented as *bytes* objects; this flag is false by default. *datetime.datetime*, *bytes* and *bytearray* objects may be passed to calls. The obsolete *use_datetime* flag is similar to *use_builtin_types* but it applies only to date/time values.

Both the HTTP and HTTPS transports support the URL syntax extension for HTTP Basic Authentication: `http://user:pass@host:port/path`. The `user:pass` portion will be base64-encoded as an HTTP 'Authorization' header, and sent to the remote server as part of the connection process when invoking an XML-RPC method. You only need to use this if the remote server requires a Basic Authentication user and password. If an HTTPS URL is provided, *context* may be *ssl.SSLContext* and configures the SSL settings of the underlying HTTPS connection.

The returned instance is a proxy object with methods that can be used to invoke corresponding RPC calls on the remote server. If the remote server supports the introspection API, the proxy can also be used to query the remote server for the methods it supports (service discovery) and fetch other server-associated metadata.

Types that are conformable (e.g. that can be marshalled through XML), include the following (and except where noted, they are unmarshalled as the same Python type):

XML-RPC type	Python type
`boolean`	*bool*
`int`, `i1`, `i2`, `i4`, `i8` or `biginteger`	*int* in range from -2147483648 to 2147483647. Values get the `<int>` tag.
`double` or `float`	*float*. Values get the `<double>` tag.
`string`	*str*
`array`	*list* or *tuple* containing conformable elements. Arrays are returned as *lists*.
`struct`	*dict*. Keys must be strings, values may be any conformable type. Objects of user-defined classes can be passed in; only their *__dict__* attribute is transmitted.
`dateTime.iso8601`	*DateTime* or *datetime.datetime*. Returned type depends on values of *use_builtin_types* and *use_datetime* flags.
`base64`	*Binary*, *bytes* or *bytearray*. Returned type depends on the value of the *use_builtin_types* flag.
`nil`	The `None` constant. Passing is allowed only if *allow_none* is true.
`bigdecimal`	*decimal.Decimal*. Returned type only.

This is the full set of data types supported by XML-RPC. Method calls may also raise a special *Fault* instance, used to signal XML-RPC server errors, or *ProtocolError* used to signal an error in the HTTP/HTTPS transport layer. Both *Fault* and *ProtocolError* derive from a base class called `Error`. Note that the xmlrpc client module currently does not marshal instances of subclasses of built-in types.

When passing strings, characters special to XML such as `<`, `>`, and `&` will be automatically escaped. However, it's the caller's responsibility to ensure that the string is free of characters that aren't allowed in XML, such as the control characters with ASCII values between 0 and 31 (except, of course, tab, newline and carriage return); failing to do this will result in an XML-RPC request that isn't well-formed XML. If you have to pass arbitrary bytes via XML-RPC, use *bytes* or *bytearray* classes or the *Binary* wrapper class described below.

`Server` is retained as an alias for *ServerProxy* for backwards compatibility. New code should use *ServerProxy*.

Changed in version 3.5: Added the *context* argument.

Changed in version 3.6: Added support of type tags with prefixes (e.g. `ex:nil`). Added support of unmarshalling additional types used by Apache XML-RPC implementation for numerics: `i1`, `i2`, `i8`, `biginteger`, `float` and `bigdecimal`. See http://ws.apache.org/xmlrpc/types.html for a description.

See also:

XML-RPC HOWTO A good description of XML-RPC operation and client software in several languages. Contains pretty much everything an XML-RPC client developer needs to know.

XML-RPC Introspection Describes the XML-RPC protocol extension for introspection.

XML-RPC Specification The official specification.

Unofficial XML-RPC Errata Fredrik Lundh's "unofficial errata, intended to clarify certain details in the XML-RPC specification, as well as hint at 'best practices' to use when designing your own XML-RPC implementations."

22.26.1 ServerProxy Objects

A *ServerProxy* instance has a method corresponding to each remote procedure call accepted by the XML-RPC server. Calling the method performs an RPC, dispatched by both name and argument signature (e.g. the same method name can be overloaded with multiple argument signatures). The RPC finishes by returning a value, which may be either returned data in a conformant type or a *Fault* or *ProtocolError* object indicating an error.

Servers that support the XML introspection API support some common methods grouped under the reserved `system` attribute:

`ServerProxy.system.listMethods()`
This method returns a list of strings, one for each (non-system) method supported by the XML-RPC server.

`ServerProxy.system.methodSignature(`*name*`)`
This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns an array of possible signatures for this method. A signature is an array of types. The first of these types is the return type of the method, the rest are parameters.

Because multiple signatures (ie. overloading) is permitted, this method returns a list of signatures rather than a singleton.

Signatures themselves are restricted to the top level parameters expected by a method. For instance if a method expects one array of structs as a parameter, and it returns a string, its signature is simply "string, array". If it expects three integers and returns a string, its signature is "string, int, int, int".

If no signature is defined for the method, a non-array value is returned. In Python this means that the type of the returned value will be something other than list.

ServerProxy.system.methodHelp(*name*)
This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns a documentation string describing the use of that method. If no such string is available, an empty string is returned. The documentation string may contain HTML markup.

Changed in version 3.5: Instances of *ServerProxy* support the *context manager* protocol for closing the underlying transport.

A working example follows. The server code:

```
from xmlrpc.server import SimpleXMLRPCServer

def is_even(n):
    return n % 2 == 0

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(is_even, "is_even")
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client

with xmlrpc.client.ServerProxy("http://localhost:8000/") as proxy:
    print("3 is even: %s" % str(proxy.is_even(3)))
    print("100 is even: %s" % str(proxy.is_even(100)))
```

22.26.2 DateTime Objects

class xmlrpc.client.DateTime
This class may be initialized with seconds since the epoch, a time tuple, an ISO 8601 time/date string, or a *datetime.datetime* instance. It has the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

decode(*string*)
Accept a string as the instance's new time value.

encode(*out*)
Write the XML-RPC encoding of this *DateTime* item to the *out* stream object.

It also supports certain of Python's built-in operators through rich comparison and **__repr__()** methods.

A working example follows. The server code:

```
import datetime
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def today():
    today = datetime.datetime.today()
    return xmlrpc.client.DateTime(today)
```

(continues on next page)

(continued from previous page)

```
server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(today, "today")
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client
import datetime

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")

today = proxy.today()
# convert the ISO8601 string to a datetime object
converted = datetime.datetime.strptime(today.value, "%Y%m%dT%H:%M:%S")
print("Today: %s" % converted.strftime("%d.%m.%Y, %H:%M"))
```

22.26.3 Binary Objects

class xmlrpc.client.**Binary**

This class may be initialized from bytes data (which may include NULs). The primary access to the content of a *Binary* object is provided by an attribute:

data

The binary data encapsulated by the *Binary* instance. The data is provided as a *bytes* object.

Binary objects have the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

decode(*bytes*)

Accept a base64 *bytes* object and decode it as the instance's new data.

encode(*out*)

Write the XML-RPC base 64 encoding of this binary item to the *out* stream object.

The encoded data will have newlines every 76 characters as per **RFC 2045 section 6.8**, which was the de facto standard base64 specification when the XML-RPC spec was written.

It also supports certain of Python's built-in operators through `__eq__()` and `__ne__()` methods.

Example usage of the binary objects. We're going to transfer an image over XMLRPC:

```
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def python_logo():
    with open("python_logo.jpg", "rb") as handle:
        return xmlrpc.client.Binary(handle.read())

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(python_logo, 'python_logo')

server.serve_forever()
```

The client gets the image and saves it to a file:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
with open("fetched_python_logo.jpg", "wb") as handle:
    handle.write(proxy.python_logo().data)
```

22.26.4 Fault Objects

class xmlrpc.client.**Fault**

A *Fault* object encapsulates the content of an XML-RPC fault tag. Fault objects have the following attributes:

faultCode

A string indicating the fault type.

faultString

A string containing a diagnostic message associated with the fault.

In the following example we're going to intentionally cause a *Fault* by returning a complex type object. The server code:

```
from xmlrpc.server import SimpleXMLRPCServer

# A marshalling error is going to occur because we're returning a
# complex number
def add(x, y):
    return x+y+0j

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(add, 'add')

server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
try:
    proxy.add(2, 5)
except xmlrpc.client.Fault as err:
    print("A fault occurred")
    print("Fault code: %d" % err.faultCode)
    print("Fault string: %s" % err.faultString)
```

22.26.5 ProtocolError Objects

class xmlrpc.client.**ProtocolError**

A *ProtocolError* object describes a protocol error in the underlying transport layer (such as a 404 'not found' error if the server named by the URI does not exist). It has the following attributes:

url

The URI or URL that triggered the error.

errcode
The error code.

errmsg
The error message or diagnostic string.

headers
A dict containing the headers of the HTTP/HTTPS request that triggered the error.

In the following example we're going to intentionally cause a *ProtocolError* by providing an invalid URI:

```
import xmlrpc.client

# create a ServerProxy with a URI that doesn't respond to XMLRPC requests
proxy = xmlrpc.client.ServerProxy("http://google.com/")

try:
    proxy.some_method()
except xmlrpc.client.ProtocolError as err:
    print("A protocol error occurred")
    print("URL: %s" % err.url)
    print("HTTP/HTTPS headers: %s" % err.headers)
    print("Error code: %d" % err.errcode)
    print("Error message: %s" % err.errmsg)
```

22.26.6 MultiCall Objects

The *MultiCall* object provides a way to encapsulate multiple calls to a remote server into a single request[1].

class xmlrpc.client.MultiCall(*server*)
Create an object used to boxcar method calls. *server* is the eventual target of the call. Calls can be made to the result object, but they will immediately return `None`, and only store the call name and parameters in the *MultiCall* object. Calling the object itself causes all stored calls to be transmitted as a single `system.multicall` request. The result of this call is a *generator*; iterating over this generator yields the individual results.

A usage example of this class follows. The server code:

```
from xmlrpc.server import SimpleXMLRPCServer

def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    return x // y

# A simple server with simple arithmetic functions
```

(continues on next page)

[1] This approach has been first presented in a discussion on xmlrpc.com.

(continued from previous page)

```
server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_multicall_functions()
server.register_function(add, 'add')
server.register_function(subtract, 'subtract')
server.register_function(multiply, 'multiply')
server.register_function(divide, 'divide')
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
multicall = xmlrpc.client.MultiCall(proxy)
multicall.add(7, 3)
multicall.subtract(7, 3)
multicall.multiply(7, 3)
multicall.divide(7, 3)
result = multicall()

print("7+3=%d, 7-3=%d, 7*3=%d, 7//3=%d" % tuple(result))
```

22.26.7 Convenience Functions

xmlrpc.client.**dumps**(*params*, *methodname=None*, *methodresponse=None*, *encoding=None*, *allow_none=False*)

Convert *params* into an XML-RPC request. or into a response if *methodresponse* is true. *params* can be either a tuple of arguments or an instance of the *Fault* exception class. If *methodresponse* is true, only a single value can be returned, meaning that *params* must be of length 1. *encoding*, if supplied, is the encoding to use in the generated XML; the default is UTF-8. Python's *None* value cannot be used in standard XML-RPC; to allow using it via an extension, provide a true value for *allow_none*.

xmlrpc.client.**loads**(*data*, *use_datetime=False*, *use_builtin_types=False*)

Convert an XML-RPC request or response into Python objects, a `(params, methodname)`. *params* is a tuple of argument; *methodname* is a string, or `None` if no method name is present in the packet. If the XML-RPC packet represents a fault condition, this function will raise a *Fault* exception. The *use_builtin_types* flag can be used to cause date/time values to be presented as *datetime.datetime* objects and binary data to be presented as *bytes* objects; this flag is false by default.

The obsolete *use_datetime* flag is similar to *use_builtin_types* but it applies only to date/time values.

Changed in version 3.3: The *use_builtin_types* flag was added.

22.26.8 Example of Client Usage

```
# simple test program (from the XML-RPC specification)
from xmlrpc.client import ServerProxy, Error

# server = ServerProxy("http://localhost:8000") # local server
with ServerProxy("http://betty.userland.com") as proxy:
```

(continues on next page)

(continued from previous page)

```
    print(proxy)

    try:
        print(proxy.examples.getStateName(41))
    except Error as v:
        print("ERROR", v)
```

To access an XML-RPC server through a HTTP proxy, you need to define a custom transport. The following example shows how:

```
import http.client
import xmlrpc.client

class ProxiedTransport(xmlrpc.client.Transport):

    def set_proxy(self, host, port=None, headers=None):
        self.proxy = host, port
        self.proxy_headers = headers

    def make_connection(self, host):
        connection = http.client.HTTPConnection(*self.proxy)
        connection.set_tunnel(host, headers=self.proxy_headers)
        self._connection = host, connection
        return connection

transport = ProxiedTransport()
transport.set_proxy('proxy-server', 8080)
server = xmlrpc.client.ServerProxy('http://betty.userland.com', transport=transport)
print(server.examples.getStateName(41))
```

22.26.9 Example of Client and Server Usage

See *SimpleXMLRPCServer Example*.

22.27 `xmlrpc.server` — Basic XML-RPC servers

Source code: Lib/xmlrpc/server.py

The *`xmlrpc.server`* module provides a basic server framework for XML-RPC servers written in Python. Servers can either be free standing, using *`SimpleXMLRPCServer`*, or embedded in a CGI environment, using *`CGIXMLRPCRequestHandler`*.

Warning: The *`xmlrpc.server`* module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML vulnerabilities*.

class xmlrpc.server.**SimpleXMLRPCServer**(*addr*, *requestHandler=SimpleXMLRPCRequestHandler*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=False*)

Create a new server instance. This class provides methods for registration of functions that can be called

by the XML-RPC protocol. The *requestHandler* parameter should be a factory for request handler instances; it defaults to *SimpleXMLRPCRequestHandler*. The *addr* and *requestHandler* parameters are passed to the *socketserver.TCPServer* constructor. If *logRequests* is true (the default), requests will be logged; setting this parameter to false will turn off logging. The *allow_none* and *encoding* parameters are passed on to *xmlrpc.client* and control the XML-RPC responses that will be returned from the server. The *bind_and_activate* parameter controls whether `server_bind()` and `server_activate()` are called immediately by the constructor; it defaults to true. Setting it to false allows code to manipulate the *allow_reuse_address* class variable before the address is bound. The *use_builtin_types* parameter is passed to the *loads()* function and controls which types are processed when date/times values or binary data are received; it defaults to false.

Changed in version 3.3: The *use_builtin_types* flag was added.

class xmlrpc.server.**CGIXMLRPCRequestHandler**(*allow_none=False*, *encoding=None*, *use_builtin_types=False*)

Create a new instance to handle XML-RPC requests in a CGI environment. The *allow_none* and *encoding* parameters are passed on to *xmlrpc.client* and control the XML-RPC responses that will be returned from the server. The *use_builtin_types* parameter is passed to the *loads()* function and controls which types are processed when date/times values or binary data are received; it defaults to false.

Changed in version 3.3: The *use_builtin_types* flag was added.

class xmlrpc.server.**SimpleXMLRPCRequestHandler**

Create a new request handler instance. This request handler supports `POST` requests and modifies logging so that the *logRequests* parameter to the *SimpleXMLRPCServer* constructor parameter is honored.

22.27.1 SimpleXMLRPCServer Objects

The *SimpleXMLRPCServer* class is based on *socketserver.TCPServer* and provides a means of creating simple, stand alone XML-RPC servers.

SimpleXMLRPCServer.**register_function**(*function=None*, *name=None*)

Register a function that can respond to XML-RPC requests. If *name* is given, it will be the method name associated with *function*, otherwise `function.__name__` will be used. *name* is a string, and may contain characters not legal in Python identifiers, including the period character.

This method can also be used as a decorator. When used as a decorator, *name* can only be given as a keyword argument to register *function* under *name*. If no *name* is given, `function.__name__` will be used.

Changed in version 3.7: *register_function()* can be used as a decorator.

SimpleXMLRPCServer.**register_instance**(*instance*, *allow_dotted_names=False*)

Register an object which is used to expose method names which have not been registered using *register_function()*. If *instance* contains a `_dispatch()` method, it is called with the requested method name and the parameters from the request. Its API is `def _dispatch(self, method, params)` (note that *params* does not represent a variable argument list). If it calls an underlying function to perform its task, that function is called as `func(*params)`, expanding the parameter list. The return value from `_dispatch()` is returned to the client as the result. If *instance* does not have a `_dispatch()` method, it is searched for an attribute matching the name of the requested method.

If the optional *allow_dotted_names* argument is true and the instance does not have a `_dispatch()` method, then if the requested method name contains periods, each component of the method name is searched for individually, with the effect that a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

> **Warning:** Enabling the *allow_dotted_names* option allows intruders to access your module's global variables and may allow intruders to execute arbitrary code on your machine. Only use this option on a secure, closed network.

`SimpleXMLRPCServer.register_introspection_functions()`
Registers the XML-RPC introspection functions `system.listMethods`, `system.methodHelp` and `system.methodSignature`.

`SimpleXMLRPCServer.register_multicall_functions()`
Registers the XML-RPC multicall function system.multicall.

`SimpleXMLRPCRequestHandler.rpc_paths`
An attribute value that must be a tuple listing valid path portions of the URL for receiving XML-RPC requests. Requests posted to other paths will result in a 404 "no such page" HTTP error. If this tuple is empty, all paths will be considered valid. The default value is `('/', '/RPC2')`.

SimpleXMLRPCServer Example

Server code:

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Create server
with SimpleXMLRPCServer(('localhost', 8000),
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()

    # Register pow() function; this will use the value of
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name
    def adder_function(x, y):
        return x + y
    server.register_function(adder_function, 'add')

    # Register an instance; all the methods of the instance are
    # published as XML-RPC methods (in this case, just 'mul').
    class MyFuncs:
        def mul(self, x, y):
            return x * y

    server.register_instance(MyFuncs())

    # Run the server's main loop
    server.serve_forever()
```

The following client code will call the methods made available by the preceding server:

```
import xmlrpc.client

s = xmlrpc.client.ServerProxy('http://localhost:8000')
print(s.pow(2,3))  # Returns 2**3 = 8
print(s.add(2,3))  # Returns 5
print(s.mul(5,2))  # Returns 5*2 = 10

# Print list of available methods
print(s.system.listMethods())
```

`register_function()` can also be used as a decorator. The previous server example can register functions in a decorator way:

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

with SimpleXMLRPCServer(('localhost', 8000),
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()

    # Register pow() function; this will use the value of
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name, using
    # register_function as a decorator. *name* can only be given
    # as a keyword argument.
    @server.register_function(name='add')
    def adder_function(x, y):
        return x + y

    # Register a function under function.__name__.
    @server.register_function
    def mul(x, y):
        return x * y

    server.serve_forever()
```

The following example included in the `Lib/xmlrpc/server.py` module shows a server allowing dotted names and registering a multicall function.

Warning: Enabling the *allow_dotted_names* option allows intruders to access your module's global variables and may allow intruders to execute arbitrary code on your machine. Only use this example only within a secure, closed network.

```
import datetime

class ExampleService:
```

(continues on next page)

(continued from previous page)

```
    def getData(self):
        return '42'

    class currentTime:
        @staticmethod
        def getCurrentTime():
            return datetime.datetime.now()

with SimpleXMLRPCServer(("localhost", 8000)) as server:
    server.register_function(pow)
    server.register_function(lambda x,y: x+y, 'add')
    server.register_instance(ExampleService(), allow_dotted_names=True)
    server.register_multicall_functions()
    print('Serving XML-RPC on localhost port 8000')
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        print("\nKeyboard interrupt received, exiting.")
        sys.exit(0)
```

This ExampleService demo can be invoked from the command line:

```
python -m xmlrpc.server
```

The client that interacts with the above server is included in *Lib/xmlrpc/client.py*:

```
server = ServerProxy("http://localhost:8000")

try:
    print(server.currentTime.getCurrentTime())
except Error as v:
    print("ERROR", v)

multi = MultiCall(server)
multi.getData()
multi.pow(2,9)
multi.add(1,2)
try:
    for response in multi():
        print(response)
except Error as v:
    print("ERROR", v)
```

This client which interacts with the demo XMLRPC server can be invoked as:

```
python -m xmlrpc.client
```

22.27.2 CGIXMLRPCRequestHandler

The *CGIXMLRPCRequestHandler* class can be used to handle XML-RPC requests sent to Python CGI scripts.

CGIXMLRPCRequestHandler.**register_function**(*function=None*, *name=None*)
Register a function that can respond to XML-RPC requests. If *name* is given, it will be the method name associated with *function*, otherwise `function.__name__` will be used. *name* is a string, and may contain characters not legal in Python identifiers, including the period character.

This method can also be used as a decorator. When used as a decorator, *name* can only be given as a keyword argument to register *function* under *name*. If no *name* is given, `function.__name__` will be used.

Changed in version 3.7: *register_function()* can be used as a decorator.

CGIXMLRPCRequestHandler.**register_instance**(*instance*)
Register an object which is used to expose method names which have not been registered using *register_function()*. If instance contains a `_dispatch()` method, it is called with the requested method name and the parameters from the request; the return value is returned to the client as the result. If instance does not have a `_dispatch()` method, it is searched for an attribute matching the name of the requested method; if the requested method name contains periods, each component of the method name is searched for individually, with the effect that a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

CGIXMLRPCRequestHandler.**register_introspection_functions**()
Register the XML-RPC introspection functions `system.listMethods`, `system.methodHelp` and `system.methodSignature`.

CGIXMLRPCRequestHandler.**register_multicall_functions**()
Register the XML-RPC multicall function `system.multicall`.

CGIXMLRPCRequestHandler.**handle_request**(*request_text=None*)
Handle an XML-RPC request. If *request_text* is given, it should be the POST data provided by the HTTP server, otherwise the contents of stdin will be used.

Example:

```
class MyFuncs:
    def mul(self, x, y):
        return x * y


handler = CGIXMLRPCRequestHandler()
handler.register_function(pow)
handler.register_function(lambda x,y: x+y, 'add')
handler.register_introspection_functions()
handler.register_instance(MyFuncs())
handler.handle_request()
```

22.27.3 Documenting XMLRPC server

These classes extend the above classes to serve HTML documentation in response to HTTP GET requests. Servers can either be free standing, using *DocXMLRPCServer*, or embedded in a CGI environment, using *DocCGIXMLRPCRequestHandler*.

class xmlrpc.server.**DocXMLRPCServer**(*addr*, *requestHandler=DocXMLRPCRequestHandler*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=True*)
Create a new server instance. All parameters have the same meaning as for *SimpleXMLRPCServer*; *requestHandler* defaults to *DocXMLRPCRequestHandler*.

Changed in version 3.3: The *use_builtin_types* flag was added.

class xmlrpc.server.**DocCGIXMLRPCRequestHandler**
Create a new instance to handle XML-RPC requests in a CGI environment.

class xmlrpc.server.**DocXMLRPCRequestHandler**
Create a new request handler instance. This request handler supports XML-RPC POST requests, documentation GET requests, and modifies logging so that the *logRequests* parameter to the *DocXMLRPCServer* constructor parameter is honored.

22.27.4 DocXMLRPCServer Objects

The *DocXMLRPCServer* class is derived from *SimpleXMLRPCServer* and provides a means of creating self-documenting, stand alone XML-RPC servers. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

DocXMLRPCServer.**set_server_title**(*server_title*)
Set the title used in the generated HTML documentation. This title will be used inside the HTML "title" element.

DocXMLRPCServer.**set_server_name**(*server_name*)
Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a "h1" element.

DocXMLRPCServer.**set_server_documentation**(*server_documentation*)
Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.

22.27.5 DocCGIXMLRPCRequestHandler

The *DocCGIXMLRPCRequestHandler* class is derived from *CGIXMLRPCRequestHandler* and provides a means of creating self-documenting, XML-RPC CGI scripts. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

DocCGIXMLRPCRequestHandler.**set_server_title**(*server_title*)
Set the title used in the generated HTML documentation. This title will be used inside the HTML "title" element.

DocCGIXMLRPCRequestHandler.**set_server_name**(*server_name*)
Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a "h1" element.

DocCGIXMLRPCRequestHandler.**set_server_documentation**(*server_documentation*)
Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.

22.28 ipaddress — IPv4/IPv6 manipulation library

Source code: Lib/ipaddress.py

---

ipaddress provides the capabilities to create, manipulate and operate on IPv4 and IPv6 addresses and networks.

The functions and classes in this module make it straightforward to handle various tasks related to IP addresses, including checking whether or not two hosts are on the same subnet, iterating over all hosts in a particular subnet, checking whether or not a string represents a valid IP address or network definition, and so on.

This is the full module API reference—for an overview and introduction, see ipaddress-howto.

New in version 3.3.

22.28.1 Convenience factory functions

The *ipaddress* module provides factory functions to conveniently create IP addresses, networks and interfaces:

ipaddress.ip_address(*address*)
Return an *IPv4Address* or *IPv6Address* object depending on the IP address passed as argument. Either IPv4 or IPv6 addresses may be supplied; integers less than 2**32 will be considered to be IPv4 by default. A *ValueError* is raised if *address* does not represent a valid IPv4 or IPv6 address.

```
>>> ipaddress.ip_address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.ip_address('2001:db8::')
IPv6Address('2001:db8::')
```

ipaddress.ip_network(*address*, *strict=True*)
Return an *IPv4Network* or *IPv6Network* object depending on the IP address passed as argument. *address* is a string or integer representing the IP network. Either IPv4 or IPv6 networks may be supplied; integers less than 2**32 will be considered to be IPv4 by default. *strict* is passed to *IPv4Network* or *IPv6Network* constructor. A *ValueError* is raised if *address* does not represent a valid IPv4 or IPv6 address, or if the network has host bits set.

```
>>> ipaddress.ip_network('192.168.0.0/28')
IPv4Network('192.168.0.0/28')
```

ipaddress.ip_interface(*address*)
Return an *IPv4Interface* or *IPv6Interface* object depending on the IP address passed as argument. *address* is a string or integer representing the IP address. Either IPv4 or IPv6 addresses may be supplied; integers less than 2**32 will be considered to be IPv4 by default. A *ValueError* is raised if *address* does not represent a valid IPv4 or IPv6 address.

One downside of these convenience functions is that the need to handle both IPv4 and IPv6 formats means that error messages provide minimal information on the precise error, as the functions don't know whether the IPv4 or IPv6 format was intended. More detailed error reporting can be obtained by calling the appropriate version specific class constructors directly.

22.28.2 IP Addresses

Address objects

The *IPv4Address* and *IPv6Address* objects share a lot of common attributes. Some attributes that are only meaningful for IPv6 addresses are also implemented by *IPv4Address* objects, in order to make it easier to write code that handles both IP versions correctly. Address objects are *hashable*, so they can be used as keys in dictionaries.

class ipaddress.IPv4Address(*address*)
Construct an IPv4 address. An *AddressValueError* is raised if *address* is not a valid IPv4 address.

The following constitutes a valid IPv4 address:

1. A string in decimal-dot notation, consisting of four decimal integers in the inclusive range 0–255, separated by dots (e.g. `192.168.0.1`). Each integer represents an octet (byte) in the address. Leading zeroes are tolerated only for values less than 8 (as there is no ambiguity between the decimal and octal interpretations of such strings).
2. An integer that fits into 32 bits.
3. An integer packed into a *bytes* object of length 4 (most significant octet first).

```
>>> ipaddress.IPv4Address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(3232235521)
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(b'\xC0\xA8\x00\x01')
IPv4Address('192.168.0.1')
```

version
The appropriate version number: `4` for IPv4, `6` for IPv6.

max_prefixlen
The total number of bits in the address representation for this version: `32` for IPv4, `128` for IPv6.

The prefix defines the number of leading bits in an address that are compared to determine whether or not an address is part of a network.

compressed

exploded
The string representation in dotted decimal notation. Leading zeroes are never included in the representation.

As IPv4 does not define a shorthand notation for addresses with octets set to zero, these two attributes are always the same as `str(addr)` for IPv4 addresses. Exposing these attributes makes it easier to write display code that can handle both IPv4 and IPv6 addresses.

packed
The binary representation of this address - a *bytes* object of the appropriate length (most significant octet first). This is 4 bytes for IPv4 and 16 bytes for IPv6.

reverse_pointer
The name of the reverse DNS PTR record for the IP address, e.g.:

```
>>> ipaddress.ip_address("127.0.0.1").reverse_pointer
'1.0.0.127.in-addr.arpa'
>>> ipaddress.ip_address("2001:db8::1").reverse_pointer
'1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa'
```

This is the name that could be used for performing a PTR lookup, not the resolved hostname itself.

New in version 3.5.

is_multicast
`True` if the address is reserved for multicast use. See **RFC 3171** (for IPv4) or **RFC 2373** (for IPv6).

is_private
`True` if the address is allocated for private networks. See iana-ipv4-special-registry (for IPv4) or iana-ipv6-special-registry (for IPv6).

is_global
`True` if the address is allocated for public networks. See iana-ipv4-special-registry (for IPv4) or iana-ipv6-special-registry (for IPv6).

New in version 3.4.

is_unspecified
`True` if the address is unspecified. See **RFC 5735** (for IPv4) or **RFC 2373** (for IPv6).

is_reserved
`True` if the address is otherwise IETF reserved.

is_loopback
`True` if this is a loopback address. See **RFC 3330** (for IPv4) or **RFC 2373** (for IPv6).

is_link_local
`True` if the address is reserved for link-local usage. See **RFC 3927**.

class ipaddress.IPv6Address(*address*)
Construct an IPv6 address. An *AddressValueError* is raised if *address* is not a valid IPv6 address.

The following constitutes a valid IPv6 address:

1. A string consisting of eight groups of four hexadecimal digits, each group representing 16 bits. The groups are separated by colons. This describes an *exploded* (longhand) notation. The string can also be *compressed* (shorthand notation) by various means. See **RFC 4291** for details. For example, `"0000:0000:0000:0000:0000:0abc:0007:0def"` can be compressed to `"::abc:7:def"`.
2. An integer that fits into 128 bits.
3. An integer packed into a *bytes* object of length 16, big-endian.

```
>>> ipaddress.IPv6Address('2001:db8::1000')
IPv6Address('2001:db8::1000')
```

compressed

The short form of the address representation, with leading zeroes in groups omitted and the longest sequence of groups consisting entirely of zeroes collapsed to a single empty group.

This is also the value returned by `str(addr)` for IPv6 addresses.

exploded

The long form of the address representation, with all leading zeroes and groups consisting entirely of zeroes included.

For the following attributes, see the corresponding documentation of the *IPv4Address* class:

packed

reverse_pointer

version

max_prefixlen

is_multicast

is_private

is_global

is_unspecified

is_reserved

is_loopback

`is_link_local`
: New in version 3.4: is_global

`is_site_local`
: `True` if the address is reserved for site-local usage. Note that the site-local address space has been deprecated by **RFC 3879**. Use *`is_private`* to test if this address is in the space of unique local addresses as defined by **RFC 4193**.

`ipv4_mapped`
: For addresses that appear to be IPv4 mapped addresses (starting with `::FFFF/96`), this property will report the embedded IPv4 address. For any other address, this property will be `None`.

`sixtofour`
: For addresses that appear to be 6to4 addresses (starting with `2002::/16`) as defined by **RFC 3056**, this property will report the embedded IPv4 address. For any other address, this property will be `None`.

`teredo`
: For addresses that appear to be Teredo addresses (starting with `2001::/32`) as defined by **RFC 4380**, this property will report the embedded `(server, client)` IP address pair. For any other address, this property will be `None`.

Conversion to Strings and Integers

To interoperate with networking interfaces such as the socket module, addresses must be converted to strings or integers. This is handled using the *`str()`* and *`int()`* builtin functions:

```
>>> str(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> int(ipaddress.IPv4Address('192.168.0.1'))
3232235521
>>> str(ipaddress.IPv6Address('::1'))
'::1'
>>> int(ipaddress.IPv6Address('::1'))
1
```

Operators

Address objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Comparison operators

Address objects can be compared with the usual set of comparison operators. Some examples:

```
>>> IPv4Address('127.0.0.2') > IPv4Address('127.0.0.1')
True
>>> IPv4Address('127.0.0.2') == IPv4Address('127.0.0.1')
False
>>> IPv4Address('127.0.0.2') != IPv4Address('127.0.0.1')
True
```

Arithmetic operators

Integers can be added to or subtracted from address objects. Some examples:

```
>>> IPv4Address('127.0.0.2') + 3
IPv4Address('127.0.0.5')
>>> IPv4Address('127.0.0.2') - 3
IPv4Address('126.255.255.255')
>>> IPv4Address('255.255.255.255') + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ipaddress.AddressValueError: 4294967296 (>= 2**32) is not permitted as an IPv4 address
```

22.28.3 IP Network definitions

The *IPv4Network* and *IPv6Network* objects provide a mechanism for defining and inspecting IP network definitions. A network definition consists of a *mask* and a *network address*, and as such defines a range of IP addresses that equal the network address when masked (binary AND) with the mask. For example, a network definition with the mask `255.255.255.0` and the network address `192.168.1.0` consists of IP addresses in the inclusive range `192.168.1.0` to `192.168.1.255`.

Prefix, net mask and host mask

There are several equivalent ways to specify IP network masks. A *prefix* `/<nbits>` is a notation that denotes how many high-order bits are set in the network mask. A *net mask* is an IP address with some number of high-order bits set. Thus the prefix `/24` is equivalent to the net mask `255.255.255.0` in IPv4, or `ffff:ff00::` in IPv6. In addition, a *host mask* is the logical inverse of a *net mask*, and is sometimes used (for example in Cisco access control lists) to denote a network mask. The host mask equivalent to `/24` in IPv4 is `0.0.0.255`.

Network objects

All attributes implemented by address objects are implemented by network objects as well. In addition, network objects implement additional attributes. All of these are common between *IPv4Network* and *IPv6Network*, so to avoid duplication they are only documented for *IPv4Network*. Network objects are *hashable*, so they can be used as keys in dictionaries.

class ipaddress.IPv4Network(*address*, *strict=True*)

Construct an IPv4 network definition. *address* can be one of the following:

1. A string consisting of an IP address and an optional mask, separated by a slash (`/`). The IP address is the network address, and the mask can be either a single number, which means it's a *prefix*, or a string representation of an IPv4 address. If it's the latter, the mask is interpreted as a *net mask* if it starts with a non-zero field, or as a *host mask* if it starts with a zero field, with the single exception of an all-zero mask which is treated as a *net mask*. If no mask is provided, it's considered to be `/32`.

   For example, the following *address* specifications are equivalent: `192.168.1.0/24`, `192.168.1.0/255.255.255.0` and `192.168.1.0/0.0.0.255`.

2. An integer that fits into 32 bits. This is equivalent to a single-address network, with the network address being *address* and the mask being `/32`.

3. An integer packed into a *bytes* object of length 4, big-endian. The interpretation is similar to an integer *address*.

4. A two-tuple of an address description and a netmask, where the address description is either a string, a 32-bits integer, a 4-bytes packed integer, or an existing IPv4Address object; and the netmask is either an integer representing the prefix length (e.g. `24`) or a string representing the prefix mask (e.g. `255.255.255.0`).

An *AddressValueError* is raised if *address* is not a valid IPv4 address. A *NetmaskValueError* is raised if the mask is not valid for an IPv4 address.

If *strict* is `True` and host bits are set in the supplied address, then *ValueError* is raised. Otherwise, the host bits are masked out to determine the appropriate network address.

Unless stated otherwise, all network methods accepting other network/address objects will raise *TypeError* if the argument's IP version is incompatible to `self`.

Changed in version 3.5: Added the two-tuple form for the *address* constructor parameter.

version

max_prefixlen
Refer to the corresponding attribute documentation in *IPv4Address*.

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local
These attributes are true for the network as a whole if they are true for both the network address and the broadcast address.

network_address
The network address for the network. The network address and the prefix length together uniquely define a network.

broadcast_address
The broadcast address for the network. Packets sent to the broadcast address should be received by every host on the network.

hostmask
The host mask, as an *IPv4Address* object.

netmask
The net mask, as an *IPv4Address* object.

with_prefixlen

compressed

exploded
A string representation of the network, with the mask in prefix notation.

`with_prefixlen` and `compressed` are always the same as `str(network)`. `exploded` uses the exploded form the network address.

with_netmask
A string representation of the network, with the mask in net mask notation.

with_hostmask
A string representation of the network, with the mask in host mask notation.

num_addresses
The total number of addresses in the network.

prefixlen
Length of the network prefix, in bits.

hosts()
Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the network address itself and the network broadcast address. For networks with a mask length of 31, the network address and network broadcast address are also included in the result.

```
>>> list(ip_network('192.0.2.0/29').hosts())  #doctest: +NORMALIZE_WHITESPACE
[IPv4Address('192.0.2.1'), IPv4Address('192.0.2.2'),
 IPv4Address('192.0.2.3'), IPv4Address('192.0.2.4'),
 IPv4Address('192.0.2.5'), IPv4Address('192.0.2.6')]
>>> list(ip_network('192.0.2.0/31').hosts())
[IPv4Address('192.0.2.0'), IPv4Address('192.0.2.1')]
```

overlaps(*other*)
`True` if this network is partly or wholly contained in *other* or *other* is wholly contained in this network.

address_exclude(*network*)
Computes the network definitions resulting from removing the given *network* from this one. Returns an iterator of network objects. Raises *ValueError* if *network* is not completely contained in this network.

```
>>> n1 = ip_network('192.0.2.0/28')
>>> n2 = ip_network('192.0.2.1/32')
>>> list(n1.address_exclude(n2))  #doctest: +NORMALIZE_WHITESPACE
[IPv4Network('192.0.2.8/29'), IPv4Network('192.0.2.4/30'),
 IPv4Network('192.0.2.2/31'), IPv4Network('192.0.2.0/32')]
```

subnets(*prefixlen_diff=1*, *new_prefix=None*)
The subnets that join to make the current network definition, depending on the argument values. *prefixlen_diff* is the amount our prefix length should be increased by. *new_prefix* is the desired new prefix of the subnets; it must be larger than our prefix. One and only one of *prefixlen_diff* and *new_prefix* must be set. Returns an iterator of network objects.

```
>>> list(ip_network('192.0.2.0/24').subnets())
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
>>> list(ip_network('192.0.2.0/24').subnets(prefixlen_diff=2))  #doctest:␣
↪+NORMALIZE_WHITESPACE
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=26))  #doctest:␣
↪+NORMALIZE_WHITESPACE
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=23))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise ValueError('new prefix must be longer')
ValueError: new prefix must be longer
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=25))
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
```

supernet(*prefixlen_diff=1*, *new_prefix=None*)

The supernet containing this network definition, depending on the argument values. *prefixlen_diff* is the amount our prefix length should be decreased by. *new_prefix* is the desired new prefix of the supernet; it must be smaller than our prefix. One and only one of *prefixlen_diff* and *new_prefix* must be set. Returns a single network object.

```
>>> ip_network('192.0.2.0/24').supernet()
IPv4Network('192.0.2.0/23')
>>> ip_network('192.0.2.0/24').supernet(prefixlen_diff=2)
IPv4Network('192.0.0.0/22')
>>> ip_network('192.0.2.0/24').supernet(new_prefix=20)
IPv4Network('192.0.0.0/20')
```

subnet_of(*other*)

Returns *True* if this network is a subnet of *other*.

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> b.subnet_of(a)
True
```

New in version 3.7.

supernet_of(*other*)

Returns *True* if this network is a supernet of *other*.

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> a.supernet_of(b)
True
```

New in version 3.7.

compare_networks(*other*)

Compare this network to *other*. In this comparison only the network addresses are considered; host bits aren't. Returns either `-1`, `0` or `1`.

```
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.2/32'))
-1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.0/32'))
1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.1/32'))
0
```

Deprecated since version 3.7: It uses the same ordering and comparison algorithm as "<", "==", and ">"

class ipaddress.IPv6Network(*address*, *strict=True*)

Construct an IPv6 network definition. *address* can be one of the following:

1. A string consisting of an IP address and an optional prefix length, separated by a slash (`/`). The IP address is the network address, and the prefix length must be a single number, the *prefix*. If no prefix length is provided, it's considered to be `/128`.

   Note that currently expanded netmasks are not supported. That means `2001:db00::0/24` is a valid argument while `2001:db00::0/ffff:ff00::` not.

2. An integer that fits into 128 bits. This is equivalent to a single-address network, with the network address being *address* and the mask being `/128`.

3. An integer packed into a *bytes* object of length 16, big-endian. The interpretation is similar to an integer *address*.
4. A two-tuple of an address description and a netmask, where the address description is either a string, a 128-bits integer, a 16-bytes packed integer, or an existing IPv6Address object; and the netmask is an integer representing the prefix length.

An *AddressValueError* is raised if *address* is not a valid IPv6 address. A *NetmaskValueError* is raised if the mask is not valid for an IPv6 address.

If *strict* is `True` and host bits are set in the supplied address, then *ValueError* is raised. Otherwise, the host bits are masked out to determine the appropriate network address.

Changed in version 3.5: Added the two-tuple form for the *address* constructor parameter.

version

max_prefixlen

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

network_address

broadcast_address

hostmask

netmask

with_prefixlen

compressed

exploded

with_netmask

with_hostmask

num_addresses

prefixlen

hosts()
Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the Subnet-Router anycast address. For networks with a mask length of 127, the Subnet-Router anycast address is also included in the result.

overlaps(*other*)

address_exclude(*network*)

subnets(*prefixlen_diff=1*, *new_prefix=None*)

supernet(*prefixlen_diff=1*, *new_prefix=None*)

subnet_of(*other*)

supernet_of(*other*)

compare_networks(*other*)
Refer to the corresponding attribute documentation in *IPv4Network*.

`is_site_local`
: These attribute is true for the network as a whole if it is true for both the network address and the broadcast address.

Operators

Network objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Logical operators

Network objects can be compared with the usual set of logical operators. Network objects are ordered first by network address, then by net mask.

Iteration

Network objects can be iterated to list all the addresses belonging to the network. For iteration, *all* hosts are returned, including unusable hosts (for usable hosts, use the *hosts()* method). An example:

```
>>> for addr in IPv4Network('192.0.2.0/28'):
...     addr
...
IPv4Address('192.0.2.0')
IPv4Address('192.0.2.1')
IPv4Address('192.0.2.2')
IPv4Address('192.0.2.3')
IPv4Address('192.0.2.4')
IPv4Address('192.0.2.5')
IPv4Address('192.0.2.6')
IPv4Address('192.0.2.7')
IPv4Address('192.0.2.8')
IPv4Address('192.0.2.9')
IPv4Address('192.0.2.10')
IPv4Address('192.0.2.11')
IPv4Address('192.0.2.12')
IPv4Address('192.0.2.13')
IPv4Address('192.0.2.14')
IPv4Address('192.0.2.15')
```

Networks as containers of addresses

Network objects can act as containers of addresses. Some examples:

```
>>> IPv4Network('192.0.2.0/28')[0]
IPv4Address('192.0.2.0')
>>> IPv4Network('192.0.2.0/28')[15]
IPv4Address('192.0.2.15')
>>> IPv4Address('192.0.2.6') in IPv4Network('192.0.2.0/28')
True
>>> IPv4Address('192.0.3.6') in IPv4Network('192.0.2.0/28')
False
```

22.28.4 Interface objects

Interface objects are *hashable*, so they can be used as keys in dictionaries.

class ipaddress.IPv4Interface(*address*)

Construct an IPv4 interface. The meaning of *address* is as in the constructor of *IPv4Network*, except that arbitrary host addresses are always accepted.

IPv4Interface is a subclass of *IPv4Address*, so it inherits all the attributes from that class. In addition, the following attributes are available:

ip

The address (*IPv4Address*) without network information.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.ip
IPv4Address('192.0.2.5')
```

network

The network (*IPv4Network*) this interface belongs to.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.network
IPv4Network('192.0.2.0/24')
```

with_prefixlen

A string representation of the interface with the mask in prefix notation.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_prefixlen
'192.0.2.5/24'
```

with_netmask

A string representation of the interface with the network as a net mask.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_netmask
'192.0.2.5/255.255.255.0'
```

with_hostmask

A string representation of the interface with the network as a host mask.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_hostmask
'192.0.2.5/0.0.0.255'
```

class ipaddress.IPv6Interface(*address*)

Construct an IPv6 interface. The meaning of *address* is as in the constructor of *IPv6Network*, except that arbitrary host addresses are always accepted.

IPv6Interface is a subclass of *IPv6Address*, so it inherits all the attributes from that class. In addition, the following attributes are available:

ip

network

with_prefixlen

with_netmask

with_hostmask
Refer to the corresponding attribute documentation in *IPv4Interface*.

Operators

Interface objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Logical operators

Interface objects can be compared with the usual set of logical operators.

For equality comparison (`==` and `!=`), both the IP address and network must be the same for the objects to be equal. An interface will not compare equal to any address or network object.

For ordering (`<`, `>`, etc) the rules are different. Interface and address objects with the same IP version can be compared, and the address objects will always sort before the interface objects. Two interface objects are first compared by their networks and, if those are the same, then by their IP addresses.

22.28.5 Other Module Level Functions

The module also provides the following module level functions:

ipaddress.v4_int_to_packed(*address*)
Represent an address as 4 packed bytes in network (big-endian) order. *address* is an integer representation of an IPv4 IP address. A *ValueError* is raised if the integer is negative or too large to be an IPv4 IP address.

```
>>> ipaddress.ip_address(3221225985)
IPv4Address('192.0.2.1')
>>> ipaddress.v4_int_to_packed(3221225985)
b'\xc0\x00\x02\x01'
```

ipaddress.v6_int_to_packed(*address*)
Represent an address as 16 packed bytes in network (big-endian) order. *address* is an integer representation of an IPv6 IP address. A *ValueError* is raised if the integer is negative or too large to be an IPv6 IP address.

ipaddress.summarize_address_range(*first*, *last*)
Return an iterator of the summarized network range given the first and last IP addresses. *first* is the first *IPv4Address* or *IPv6Address* in the range and *last* is the last *IPv4Address* or *IPv6Address* in the range. A *TypeError* is raised if *first* or *last* are not IP addresses or are not of the same version. A *ValueError* is raised if *last* is not greater than *first* or if *first* address version is not 4 or 6.

```
>>> [ipaddr for ipaddr in ipaddress.summarize_address_range(
...    ipaddress.IPv4Address('192.0.2.0'),
...    ipaddress.IPv4Address('192.0.2.130'))]
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/31'), IPv4Network('192.0.2.
↪130/32')]
```

ipaddress.collapse_addresses(*addresses*)
Return an iterator of the collapsed *IPv4Network* or *IPv6Network* objects. *addresses* is an iterator of *IPv4Network* or *IPv6Network* objects. A *TypeError* is raised if *addresses* contains mixed version objects.

```
>>> [ipaddr for ipaddr in
... ipaddress.collapse_addresses([ipaddress.IPv4Network('192.0.2.0/25'),
... ipaddress.IPv4Network('192.0.2.128/25')])]
[IPv4Network('192.0.2.0/24')]
```

ipaddress.get_mixed_type_key(*obj*)

Return a key suitable for sorting between networks and addresses. Address and Network objects are not sortable by default; they're fundamentally different, so the expression:

```
IPv4Address('192.0.2.0') <= IPv4Network('192.0.2.0/24')
```

doesn't make sense. There are some times however, where you may wish to have *ipaddress* sort these anyway. If you need to do this, you can use this function as the *key* argument to *sorted()*.

obj is either a network or address object.

22.28.6 Custom Exceptions

To support more specific error reporting from class constructors, the module defines the following exceptions:

exception ipaddress.AddressValueError(*ValueError*)

Any value error related to the address.

exception ipaddress.NetmaskValueError(*ValueError*)

Any value error related to the net mask.