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

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.

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

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

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

data must be a bytes object specifying additional data to be sent to the server, or `None` if no such data is needed. *data* may also be an iterable object and in that case Content-Length value must be specified in the headers. Currently HTTP requests are the only ones that use *data*; the HTTP request will be a POST instead of a GET when the *data* parameter is provided.

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 a string in this format. It should be encoded to bytes before being used as the *data* parameter. The charset parameter in `Content-Type` header may be used to specify the encoding. If charset parameter is not sent with the Content-Type header, the server following the HTTP 1.1 recommendation may assume that the data is encoded in ISO-8859-1 encoding. It is advisable to use charset parameter with encoding used in `Content-Type` header with the `Request`.

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.

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 specifies whether to fall back to loading a default certificate store defined by the underlying OpenSSL library if the *cafile* and *capath* parameters are omitted. This will only work on some non-Windows platforms.

Warning: If neither *cafile* nor *capath* is specified, and *cadefault* is `False`, an HTTPS request will not do any verification of the server's certificate.

For http and https urls, this function returns a `http.client.HTTPResponse` object which has the following *HTTPResponse Objects* methods.

For ftp, file, and data urls and requests explicity handled by legacy `URLopener` and `FancyURLopener` classes, this function returns a `urllib.response.addinfourl` object which can work as *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.

Raises `URLError` on 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.

`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.

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 a bytes 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 HTTP request will be a POST instead of a GET when the *data* parameter is provided.

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 a string in this format. It should be encoded to bytes before being used as the *data* parameter. The charset parameter in `Content-Type` header may be used to specify the encoding. If charset parameter is not sent with the Content-Type header, the server following the HTTP 1.1 recommendation may assume that the data is encoded in ISO-8859-1 encoding. It is advisable to use charset parameter with encoding used in `Content-Type` header with the `Request`.

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, 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 example of using `Content-Type` header with *data* argument would be sending a dictionary like `{"Content-Type":" application/x-www-form-urlencoded;charset=utf-8"}`

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'`). Its value is stored in the `method` attribute and is used by `get_method()`.

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

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.

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.`**`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.

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.`**`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.

21.6.1. 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.

`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.

`Request.`**`unverifiable`**

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

`Request.`**`method`**

The HTTP request method to use. This value is used by `get_method()` to override the computed HTTP request method that would otherwise be returned. This attribute is initialized with the value of the *method* argument passed to the constructor.

New in version 3.3.

`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.`**`get_full_url`**()

Return the URL given in the constructor.

`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.`**`add_data`**(*data*)

Set the `Request` data to *data*. This is ignored by all handlers except HTTP handlers — and there it should be a byte string, and will change the request to be `POST` rather than `GET`. Deprecated in 3.3, use `Request.data`.

Deprecated since version 3.3, will be removed in version 3.4.

`Request.`**`has_data`**()

Return whether the instance has a non-`None` data. Deprecated in 3.3, use `Request.data`.

Deprecated since version 3.3, will be removed in version 3.4.

`Request.`**`get_data`**()

Return the instance's data. Deprecated in 3.3, use `Request.data`.

> *Deprecated since version 3.3, will be removed in version 3.4.*

`Request.`**`get_type`**()

Return the type of the URL — also known as the scheme. Deprecated in 3.3, use `Request.type`.

> *Deprecated since version 3.3, will be removed in version 3.4.*

`Request.`**`get_host`**()

Return the host to which a connection will be made. Deprecated in 3.3, use `Request.host`.

> *Deprecated since version 3.3, will be removed in version 3.4.*

`Request.`**`get_selector`**()

Return the selector — the part of the URL that is sent to the server. Deprecated in 3.3, use `Request.selector`.

> *Deprecated since version 3.3, will be removed in version 3.4.*

`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.

`Request.`**`set_proxy`**(*host*, *type*)

`Request.`**`get_origin_req_host`**()

Return the request-host of the origin transaction, as defined by **RFC 2965**. See the documentation for the `Request` constructor. Deprecated in 3.3, use `Request.origin_req_host`.

> *Deprecated since version 3.3, will be removed in version 3.4.*

`Request.`**`is_unverifiable`**()

Return whether the request is unverifiable, as defined by RFC 2965. See the documentation for the `Request` constructor. Deprecated in 3.3, use `Request.unverifiable`.

Deprecated since version 3.3, will be removed in version 3.4.

21.6.2. 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.

21.6.3. 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()`.

21.6.4. 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.

21.6.5. HTTPCookieProcessor Objects

`HTTPCookieProcessor` instances have one attribute:

`HTTPCookieProcessor.`**`cookiejar`**

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

21.6.6. 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.

21.6.7. 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.

21.6.8. 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).

21.6.9. HTTPBasicAuthHandler Objects

`HTTPBasicAuthHandler.`**`http_error_401`**(*req*, *fp*, *code*, *msg*, *hdrs*)

Retry the request with authentication information, if available.

21.6.10. ProxyBasicAuthHandler Objects

`ProxyBasicAuthHandler.`**`http_error_407`**(*req*, *fp*, *code*, *msg*, *hdrs*)

Retry the request with authentication information, if available.

21.6.11. 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.

21.6.12. HTTPDigestAuthHandler Objects

`HTTPDigestAuthHandler.`**`http_error_401`**(*req*, *fp*, *code*, *msg*, *hdrs*)

Retry the request with authentication information, if available.

21.6.13. ProxyDigestAuthHandler Objects

`ProxyDigestAuthHandler.`**`http_error_407`**(*req*, *fp*, *code*, *msg*, *hdrs*)

Retry the request with authentication information, if available.

21.6.14. HTTPHandler Objects

`HTTPHandler.`**`http_open`**(*req*)

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

21.6.15. HTTPSHandler Objects

`HTTPSHandler.`**`https_open`**(*req*)

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

21.6.16. 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.

21.6.17. FTPHandler Objects

`FTPHandler.`**`ftp_open`**(*req*)

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

21.6.18. 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*.

21.6.19. UnknownHandler Objects

`UnknownHandler.`**`unknown_open`**()

Raise a `URLError` exception.

21.6.20. HTTPErrorProcessor Objects

`HTTPErrorProcessor.`**`http_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`**()

Process HTTPS error responses.

The behavior is same as `http_response()`.

21.6.21. Examples

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

```
>>> import urllib.request
>>> f = urllib.request.urlopen('http://www.python.org/')
>>> 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<hea
<meta http-equiv="content-type" content="text/html; charset=utf-8" />
<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, http://www.w3.org/International/O-charset, lists the various ways in which a (X)HTML or a XML document could have specified its encoding information.

As the python.org website uses *utf-8* encoding as specified in it's 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
...                       data=b'This data is passed to stdin of the
>>> f = urllib.request.urlopen(req)
>>> 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,r
f = urllib.request.urlopen(req)
print(f.status)
print(f.reason)
```

Use of Basic HTTP Authentication:

```
import urllib.request
# Create an OpenerDirector with support for Basic HTTP Authenticatio
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.exar
proxy_auth_handler = urllib.request.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'passwo

opener = urllib.request.build_opener(proxy_handler, proxy_auth_handle
# This time, rather than install the OpenerDirector, we use it direc
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/')
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* without charset parameter 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': 
>>> f = urllib.request.urlopen("http://www.musi-cal.com/cgi-bin/quer
>>> 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('utf-8')
>>> request = urllib.request.Request("http://requestb.in/xrbl82xr")
>>> # adding charset parameter to the Content-Type header.
>>> request.add_header("Content-Type","application/x-www-form-urlenc
>>> f = urllib.request.urlopen(request, data)
>>> 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)
>>> f = opener.open("http://www.python.org")
>>> f.read().decode('utf-8')
```

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

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

21.6.22. 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 hook function that will be called once on establishment of the network connection and once after each block read thereafter. The hook 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://
>>> 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 a `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.

21.6.23. `urllib.request` Restrictions

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

21.7. `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.