`urllib2` — extensible library for opening URLs

Note: The `urllib2` module has been split across several modules in Python 3.0 named `urllib.request` and `urllib.error`. The *2to3* tool will automatically adapt imports when converting your sources to 3.0.

The `urllib2` 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 `urllib2` module defines the following functions:

`urllib2.`**`urlopen`**(*url*[, *data*][, *timeout*])

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

data may be a string 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.urlencode()` function takes a mapping or sequence of 2-tuples and returns a string in this format.

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, FTP and FTPS connections.

This function returns a file-like object with two additional methods:

- `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 `httplib.HTTPMessage` instance (see Quick Reference to HTTP Headers)

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).

Changed in version 2.6: timeout was added.

`urllib2.`**`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.

`urllib2.`**`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`, `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.

Beginning in Python 2.3, a `BaseHandler` subclass may also change its `handler_order` member variable to modify its position in the handlers list.

The following exceptions are raised as appropriate:

exception `urllib2.`**`URLError`**

The handlers raise this exception (or derived exceptions) when they run into a problem. It is a subclass of `IOError`.

`reason`

The reason for this error. It can be a message string or another exception instance (`socket.error` for remote URLs, `OSError` for local URLs).

exception `urllib2.`**`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 `BaseHTTPServer.BaseHTTPRequestHandler.responses`.

The following classes are provided:

class `urllib2.`**`Request`**(*url*[, *data*][, *headers*][, *origin_req_host*][, *unverifiable*])

This class is an abstraction of a URL request.

url should be a string containing a valid URL.

data may be a string 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.urlencode()` function takes a mapping or sequence of 2-tuples and returns a string in this format.

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 `urllib2`'s default user agent string is `"Python-urllib/2.6"` (on Python 2.6).

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 `cookielib.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.

class `urllib2.`**`OpenerDirector`**

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

class `urllib2.`**`BaseHandler`**

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

class `urllib2.`**`HTTPDefaultErrorHandler`**

A class which defines a default handler for HTTP error responses; all responses are turned into `HTTPError` exceptions.

class `urllib2.`**`HTTPRedirectHandler`**

A class to handle redirections.

class `urllib2.`**`HTTPCookieProcessor`**([*cookiejar*])

A class to handle HTTP Cookies.

class `urllib2.`**`ProxyHandler`**([*proxies*])

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 . To disable autodetected proxy pass an empty dictionary.

class `urllib2.`**`HTTPPasswordMgr`**

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

class `urllib2.`**`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 `urllib2.`**`AbstractBasicAuthHandler`**([*password_mgr*])

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 `urllib2.`**`HTTPBasicAuthHandler`**([*password_mgr*])

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.

class `urllib2.`**`ProxyBasicAuthHandler`**([*password_mgr*])

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 `urllib2.`**`AbstractDigestAuthHandler`**([*password_mgr*])

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 `urllib2.`**`HTTPDigestAuthHandler`**([*password_mgr*])

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.

class `urllib2.`**`ProxyDigestAuthHandler`**([*password_mgr*])

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 `urllib2.`**`HTTPHandler`**

A class to handle opening of HTTP URLs.

class `urllib2.`**`HTTPSHandler`**

A class to handle opening of HTTPS URLs.

class `urllib2.`**`FileHandler`**

Open local files.

class `urllib2.`**`FTPHandler`**

Open FTP URLs.

class `urllib2.`**`CacheFTPHandler`**

Open FTP URLs, keeping a cache of open FTP connections to minimize delays.

class `urllib2.`**`UnknownHandler`**

A catch-all class to handle unknown URLs.

Request Objects

The following methods describe all of `Request`‘s public interface, and so all must be overridden in subclasses.

`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`.

`Request.`**`get_method`**()

Return a string indicating the HTTP request method. This is only meaningful for HTTP requests, and currently always returns `'GET'` or `'POST'`.

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

Return whether the instance has a non-`None` data.

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

Return the instance's data.

`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.

New in version 2.4.

`Request.`**`has_header`**(*header*)

Return whether the instance has the named header (checks both regular and unredirected).

New in version 2.4.

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

Return the URL given in the constructor.

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

Return the type of the URL — also known as the scheme.

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

Return the host to which a connection will be made.

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

Return the selector — the part of the URL that is sent to the server.

`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_origin_req_host`**()

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

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

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

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*][, *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 usedi). The timeout feature actually works only for HTTP, HTTPS, FTP and FTPS connections).

Changed in version 2.6: timeout was added.

`OpenerDirector.`**`error`**(*proto*[, *arg*[, *...*]])

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.

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 members 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*)

(“protocol” is to be replaced by the protocol name.)

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*)

("protocol" is to be replaced by the protocol name.)

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*)

("protocol" is to be replaced by the protocol name.)

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()`.

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.

`HTTPRedirectHandler.`**`redirect_request`**(*req*, *fp*, *code*, *msg*, *hdrs*)

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. 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:` 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.

HTTPCookieProcessor Objects

New in version 2.4.

`HTTPCookieProcessor` instances have one attribute:

`HTTPCookieProcessor.`**`cookiejar`**

The `cookielib.CookieJar` in which cookies are stored.

ProxyHandler Objects

`ProxyHandler.`**`protocol_open`**(*request*)

("protocol" is to be replaced by the protocol name.)

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.

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.

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).

HTTPBasicAuthHandler Objects

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

Retry the request with authentication information, if available.

ProxyBasicAuthHandler Objects

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

Retry the request with authentication information, if available.

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.

HTTPDigestAuthHandler Objects

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

ProxyDigestAuthHandler Objects

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

HTTPHandler Objects

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

HTTPSHandler Objects

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

FileHandler Objects

`FileHandler.`**`file_open`**(*req*)
Open the file locally, if there is no host name, or the host name is `'localhost'`. Change the protocol to `ftp` otherwise, and retry opening it using **`parent`**.

FTPHandler Objects

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

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

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*.

UnknownHandler Objects

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

Raise a **`URLError`** exception.

HTTPErrorProcessor Objects

New in version 2.4.

`HTTPErrorProcessor.`**`unknown_open`**()

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, **`urllib2.HTTPDefaultErrorHandler`** will raise an **`HTTPError`** if no other handler handles the error.

Examples

This example gets the python.org main page and displays the first 100 bytes of it:

```
>>> import urllib2
>>> f = urllib2.urlopen('http://www.python.org/')
>>> print f.read(100)
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<?xml-stylesheet href="./css/ht2html
```

Here 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 urllib2
>>> req = urllib2.Request(url='https://localhost/cgi-bin/test.cgi',
...                       data='This data is passed to stdin of the CGI')
>>> f = urllib2.urlopen(req)
>>> print f.read()
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
```

Use of Basic HTTP Authentication:

```
import urllib2
# Create an OpenerDirector with support for Basic HTTP Authentication...
auth_handler = urllib2.HTTPBasicAuthHandler()
auth_handler.add_password(realm='PDQ Application',
                          uri='https://mahler:8092/site-updates.py',
                          user='klem',
                          passwd='kadidd!ehopper')
opener = urllib2.build_opener(auth_handler)
# ...and install it globally so it can be used with urlopen.
urllib2.install_opener(opener)
urllib2.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 = urllib2.ProxyHandler({'http': 'http://www.example.com:3128/'})
proxy_auth_handler = urllib2.HTTPBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')

opener = 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 urllib2
req = urllib2.Request('http://www.example.com/')
req.add_header('Referer', 'http://www.python.org/')
r = urllib2.urlopen(req)
```

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

```
import urllib2
opener = urllib2.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()`).