urllib.request — Network Resource Access

Purpose: A library for opening URLs that can be extended by defining custom protocol handlers.

The `urllib.request` module provides an API for using Internet resources identified by URLs. It is designed to be extended by individual applications to support new protocols or add variations to existing protocols (such as handling HTTP basic authentication).

HTTP GET

Note

The test server for these examples is in `http_server_GET.py`, from the examples for the `http.server` module. Start the server in one terminal window, then run these examples in another.

An HTTP GET operation is the simplest use of `urllib.request`. Pass the URL to `urlopen()` to get a "file-like" handle to the remote data.

```
# urllib_request_urlopen.py

from urllib import request

response = request.urlopen('http://localhost:8080/')
print('RESPONSE:', response)
print('URL     :', response.geturl())

headers = response.info()
print('DATE    :', headers['date'])
print('HEADERS :')
print('---------')
print(headers)

data = response.read().decode('utf-8')
print('LENGTH  :', len(data))
print('DATA    :')
print('---------')
print(data)
```

The example server accepts the incoming values and formats a plain text response to send back. The return value from `urlopen()` gives access to the headers from the HTTP server through the `info()` method, and the data for the remote resource via methods like `read()` and `readlines()`.

```
$ python3 urllib_request_urlopen.py

RESPONSE: <http.client.HTTPResponse object at 0x101744d68>
URL     : http://localhost:8080/
DATE    : Sat, 08 Oct 2016 18:08:54 GMT
HEADERS :
---------
Server: BaseHTTP/0.6 Python/3.5.2
Date: Sat, 08 Oct 2016 18:08:54 GMT
Content-Type: text/plain; charset=utf-8


LENGTH  : 349
DATA    :
---------
CLIENT VALUES:
client_address=('127.0.0.1', 58420) (127.0.0.1)
command=GET
path=/
real_path=/
```

```
query=
request_version=HTTP/1.1

SERVER VALUES:
server_version=BaseHTTP/0.6
sys_version=Python/3.5.2
protocol_version=HTTP/1.0

HEADERS RECEIVED:
Accept-Encoding=identity
Connection=close
Host=localhost:8080
User-Agent=Python-urllib/3.5
```

The file-like object returned by `urlopen()` is iterable:

```
# urllib_request_urlopen_iterator.py

from urllib import request

response = request.urlopen('http://localhost:8080/')
for line in response:
    print(line.decode('utf-8').rstrip())
```

This example strips the trailing newlines and carriage returns before printing the output.

```
$ python3 urllib_request_urlopen_iterator.py

CLIENT VALUES:
client_address=('127.0.0.1', 58444) (127.0.0.1)
command=GET
path=/
real path=/
query=
request_version=HTTP/1.1

SERVER VALUES:
server_version=BaseHTTP/0.6
sys_version=Python/3.5.2
protocol_version=HTTP/1.0

HEADERS RECEIVED:
Accept-Encoding=identity
Connection=close
Host=localhost:8080
User-Agent=Python-urllib/3.5
```

Encoding Arguments

Arguments can be passed to the server by encoding them with `urllib.parse.urlencode()` and appending them to the URL.

```
# urllib_request_http_get_args.py

from urllib import parse
from urllib import request

query_args = {'q': 'query string', 'foo': 'bar'}
encoded_args = parse.urlencode(query_args)
print('Encoded:', encoded_args)

url = 'http://localhost:8080/?' + encoded_args
print(request.urlopen(url).read().decode('utf-8'))
```

The list of client values returned in the example output contains the encoded query arguments.

```
$ python urllib_request_http_get_args.py
Encoded: q=query+string&foo=bar
CLIENT VALUES:
```

```
client_address=('127.0.0.1', 58455) (127.0.0.1)
command=GET
path=/?q=query+string&foo=bar
real path=/
query=q=query+string&foo=bar
request_version=HTTP/1.1

SERVER VALUES:
server_version=BaseHTTP/0.6
sys_version=Python/3.5.2
protocol_version=HTTP/1.0

HEADERS RECEIVED:
Accept-Encoding=identity
Connection=close
Host=localhost:8080
User-Agent=Python-urllib/3.5
```

HTTP POST

Note

The test server for these examples is in `http_server_POST.py`, from the examples for the `http.server` module. Start the server in one terminal window, then run these examples in another.

To send form-encoded data to the remote server using POST instead GET, pass the encoded query arguments as data to `urlopen()`.

```
# urllib_request_urlopen_post.py
```

```
from urllib import parse
from urllib import request

query_args = {'q': 'query string', 'foo': 'bar'}
encoded_args = parse.urlencode(query_args).encode('utf-8')
url = 'http://localhost:8080/'
print(request.urlopen(url, encoded_args).read().decode('utf-8'))
```

The server can decode the form data and access the individual values by name.

```
$ python3 urllib_request_urlopen_post.py

Client: ('127.0.0.1', 58568)
User-agent: Python-urllib/3.5
Path: /
Form data:
    foo=bar
    q=query string
```

Adding Outgoing Headers

`urlopen()` is a convenience function that hides some of the details of how the request is made and handled. More precise control is possible by using a `Request` instance directly. For example, custom headers can be added to the outgoing request to control the format of data returned, specify the version of a document cached locally, and tell the remote server the name of the software client communicating with it.

As the output from the earlier examples shows, the default *User-agent* header value is made up of the constant `Python-urllib`, followed by the Python interpreter version. When creating an application that will access web resources owned by someone else, it is courteous to include real user agent information in the requests, so they can identify the source of the hits more easily. Using a custom agent also allows them to control crawlers using a `robots.txt` file (see the `http.robotparser` module).

```
# urllib_request_request_header.py
```

```
from urllib import request

r = request.Request('http://localhost:8080/')
```

```
r = request.Request('http://localhost:8080/')
r.add_header(
    'User-agent',
    'PyMOTW (https://pymotw.com/)',
)

response = request.urlopen(r)
data = response.read().decode('utf-8')
print(data)
```

After creating a Request object, use add_header() to set the user agent value before opening the request. The last line of the output shows the custom value.

```
$ python3 urllib_request_request_header.py

CLIENT VALUES:
client_address=('127.0.0.1', 58585) (127.0.0.1)
command=GET
path=/
real path=/
query=
request_version=HTTP/1.1

SERVER VALUES:
server_version=BaseHTTP/0.6
sys_version=Python/3.5.2
protocol_version=HTTP/1.0

HEADERS RECEIVED:
Accept-Encoding=identity
Connection=close
Host=localhost:8080
User-Agent=PyMOTW (https://pymotw.com/)
```

Posting Form Data from a Request

The outgoing data can be specified when building the Request to have it posted to the server.

```
# urllib_request_request_post.py

from urllib import parse
from urllib import request

query_args = {'q': 'query string', 'foo': 'bar'}

r = request.Request(
    url='http://localhost:8080/',
    data=parse.urlencode(query_args).encode('utf-8'),
)
print('Request method :', r.get_method())
r.add_header(
    'User-agent',
    'PyMOTW (https://pymotw.com/)',
)

print()
print('OUTGOING DATA:')
print(r.data)

print()
print('SERVER RESPONSE:')
print(request.urlopen(r).read().decode('utf-8'))
```

The HTTP method used by the Request changes from GET to POST automatically after the data is added.

```
$ python3 urllib_request_request_post.py

Request method : POST

OUTGOING DATA:
```

```
OUTGOING DATA:
b'q=query+string&foo=bar'

SERVER RESPONSE:
Client: ('127.0.0.1', 58613)
User-agent: PyMOTW (https://pymotw.com/)
Path: /
Form data:
    foo=bar
    q=query string
```

Uploading Files

Encoding files for upload requires a little more work than simple forms. A complete MIME message needs to be constructed in the body of the request, so that the server can distinguish incoming form fields from uploaded files.

```
# urllib_request_upload_files.py

import io
import mimetypes
from urllib import request
import uuid


class MultiPartForm:
    """Accumulate the data to be used when posting a form."""

    def __init__(self):
        self.form_fields = []
        self.files = []
        # Use a large random byte string to separate
        # parts of the MIME data.
        self.boundary = uuid.uuid4().hex.encode('utf-8')
        return

    def get_content_type(self):
        return 'multipart/form-data; boundary={}'.format(
            self.boundary.decode('utf-8'))

    def add_field(self, name, value):
        """Add a simple field to the form data."""
        self.form_fields.append((name, value))

    def add_file(self, fieldname, filename, fileHandle,
                 mimetype=None):
        """Add a file to be uploaded."""
        body = fileHandle.read()
        if mimetype is None:
            mimetype = (
                mimetypes.guess_type(filename)[0] or
                'application/octet-stream'
            )
        self.files.append((fieldname, filename, mimetype, body))
        return

    @staticmethod
    def _form_data(name):
        return ('Content-Disposition: form-data; '
                'name="{}"\r\n').format(name).encode('utf-8')

    @staticmethod
    def _attached_file(name, filename):
        return ('Content-Disposition: file; '
                'name="{}"; filename="{}"\r\n').format(
                    name, filename).encode('utf-8')

    @staticmethod
    def _content_type(ct):
        return 'Content-Type: {}\r\n'.format(ct).encode('utf-8')

    def __bytes__(self):
```

```
    def __bytes__(self):
        """Return a byte-string representing the form data,
        including attached files.
        """
        buffer = io.BytesIO()
        boundary = b'--' + self.boundary + b'\r\n'

        # Add the form fields
        for name, value in self.form_fields:
            buffer.write(boundary)
            buffer.write(self._form_data(name))
            buffer.write(b'\r\n')
            buffer.write(value.encode('utf-8'))
            buffer.write(b'\r\n')

        # Add the files to upload
        for f_name, filename, f_content_type, body in self.files:
            buffer.write(boundary)
            buffer.write(self._attached_file(f_name, filename))
            buffer.write(self._content_type(f_content_type))
            buffer.write(b'\r\n')
            buffer.write(body)
            buffer.write(b'\r\n')

        buffer.write(b'--' + self.boundary + b'--\r\n')
        return buffer.getvalue()


if __name__ == '__main__':
    # Create the form with simple fields
    form = MultiPartForm()
    form.add_field('firstname', 'Doug')
    form.add_field('lastname', 'Hellmann')

    # Add a fake file
    form.add_file(
        'biography', 'bio.txt',
        fileHandle=io.BytesIO(b'Python developer and blogger.'))

    # Build the request, including the byte-string
    # for the data to be posted.
    data = bytes(form)
    r = request.Request('http://localhost:8080/', data=data)
    r.add_header(
        'User-agent',
        'PyMOTW (https://pymotw.com/)',
    )
    r.add_header('Content-type', form.get_content_type())
    r.add_header('Content-length', len(data))

    print()
    print('OUTGOING DATA:')
    for name, value in r.header_items():
        print('{}: {}'.format(name, value))
    print()
    print(r.data.decode('utf-8'))

    print()
    print('SERVER RESPONSE:')
    print(request.urlopen(r).read().decode('utf-8'))
```

The `MultiPartForm` class can represent an arbitrary form as a multi-part MIME message with attached files.

```
$ python3 urllib_request_upload_files.py

OUTGOING DATA:
User-agent: PyMOTW (https://pymotw.com/)
Content-type: multipart/form-data;
    boundary=d99b5dc60871491b9d63352eb24972b4
Content-length: 389

--d99b5dc60871491b9d63352eb24972b4
```

```
--d99b5dc60871491b9d63352eb24972b4
Content-Disposition: form-data; name="firstname"

Doug
--d99b5dc60871491b9d63352eb24972b4
Content-Disposition: form-data; name="lastname"

Hellmann
--d99b5dc60871491b9d63352eb24972b4
Content-Disposition: file; name="biography";
    filename="bio.txt"
Content-Type: text/plain

Python developer and blogger.
--d99b5dc60871491b9d63352eb24972b4--


SERVER RESPONSE:
Client: ('127.0.0.1', 59310)
User-agent: PyMOTW (https://pymotw.com/)
Path: /
Form data:
    Uploaded biography as 'bio.txt' (29 bytes)
    firstname=Doug
    lastname=Hellmann
```

Creating Custom Protocol Handlers

`urllib.request` has built-in support for HTTP(S), FTP, and local file access. To add support for other URL types, register another protocol handler. For example, to support URLs pointing to arbitrary files on remote NFS servers, without requiring users to mount the path before accessing the file, create a class derived from `BaseHandler` and with a method `nfs_open()`.

The protocol-specific `open()` method is given a single argument, the `Request` instance, and it should return an object with a `read()` method that can be used to read the data, an `info()` method to return the response headers, and `geturl()` to return the actual URL of the file being read. A simple way to achieve that is to create an instance of `urllib.response.addinfourl`, passing the headers, URL, and open file handle in to the constructor.

```
# urllib_request_nfs_handler.py

import io
import mimetypes
import os
import tempfile
from urllib import request
from urllib import response


class NFSFile:

    def __init__(self, tempdir, filename):
        self.tempdir = tempdir
        self.filename = filename
        with open(os.path.join(tempdir, filename), 'rb') as f:
            self.buffer = io.BytesIO(f.read())

    def read(self, *args):
        return self.buffer.read(*args)

    def readline(self, *args):
        return self.buffer.readline(*args)

    def close(self):
        print('\nNFSFile:')
        print('  unmounting {}'.format(
            os.path.basename(self.tempdir)))
        print('  when {} is closed'.format(
            os.path.basename(self.filename)))


class FauxNFSHandler(request.BaseHandler):
```

```
    def __init__(self, tempdir):
        self.tempdir = tempdir
        super().__init__()

    def nfs_open(self, req):
        url = req.full_url
        directory_name, file_name = os.path.split(url)
        server_name = req.host
        print('FauxNFSHandler simulating mount:')
        print('  Remote path: {}'.format(directory_name))
        print('  Server     : {}'.format(server_name))
        print('  Local path : {}'.format(
            os.path.basename(tempdir)))
        print('  Filename   : {}'.format(file_name))
        local_file = os.path.join(tempdir, file_name)
        fp = NFSFile(tempdir, file_name)
        content_type = (
            mimetypes.guess_type(file_name)[0] or
            'application/octet-stream'
        )
        stats = os.stat(local_file)
        size = stats.st_size
        headers = {
            'Content-type': content_type,
            'Content-length': size,
        }
        return response.addinfourl(fp, headers,
                                   req.get_full_url())


if __name__ == '__main__':
    with tempfile.TemporaryDirectory() as tempdir:
        # Populate the temporary file for the simulation
        filename = os.path.join(tempdir, 'file.txt')
        with open(filename, 'w', encoding='utf-8') as f:
            f.write('Contents of file.txt')

        # Construct an opener with our NFS handler
        # and register it as the default opener.
        opener = request.build_opener(FauxNFSHandler(tempdir))
        request.install_opener(opener)

        # Open the file through a URL.
        resp = request.urlopen(
            'nfs://remote_server/path/to/the/file.txt'
        )
        print()
        print('READ CONTENTS:', resp.read())
        print('URL          :', resp.geturl())
        print('HEADERS:')
        for name, value in sorted(resp.info().items()):
            print('  {:<15} = {}'.format(name, value))
        resp.close()
```

The `FauxNFSHandler` and `NFSFile` classes print messages to illustrate where a real implementation would add mount and unmount calls. Since this is just a simulation, `FauxNFSHandler` is primed with the name of a temporary directory where it should look for all of its files.

```
$ python3 urllib_request_nfs_handler.py

FauxNFSHandler simulating mount:
  Remote path: nfs://remote_server/path/to/the
  Server     : remote_server
  Local path : tmprucom5sb
  Filename   : file.txt

READ CONTENTS: b'Contents of file.txt'
URL          : nfs://remote_server/path/to/the/file.txt
HEADERS:
  Content-length  = 20
  Content-type    = text/plain
```

```
  content-type    = text/plain

NFSFile:
  unmounting tmprucom5sb
  when file.txt is closed
```

See also

- Standard library documentation for urllib.request
- `urllib.parse` – Work with the URL string itself.
- Form content types – W3C specification for posting files or large amounts of data via HTTP forms.
- `mimetypes` – Map filenames to mimetype.
- requests – Third-party HTTP library with better support for secure connections and an easier to use API. The Python core development team recommends most developers use `requests`, in part because it receives more frequent security updates than the standard library.

urllib.parse — Split URLs into Components

urllib.robotparser — Internet Spider Access Control

Quick Links

HTTP GET
Encoding Arguments
HTTP POST
Adding Outgoing Headers
Posting Form Data from a Request
Uploading Files
Creating Custom Protocol Handlers

This page was last updated 2016-12-18.

Navigation

urllib.parse — Split URLs into Components
urllib.robotparser — Internet Spider Access Control

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

Other Writing

- Blog
- The Python Standard Library By Example