PycURL

Release 7.21.5

January 30, 2016

Contents

The pycurl package is a Python interface to libcurl. pycurl has been successfully built and tested with Python versions 2.6, 2.7 and 3.1 to 3.5.

libcurl is a client-side URL transfer library supporting FTP, FTPS, HTTP, HTTPS, GOPHER, TELNET, DICT, FILE and LDAP. libcurl also supports HTTPS certificates, HTTP POST, HTTP PUT, FTP uploads, proxies, cookies, basic authentication, file transfer resume of FTP sessions, HTTP proxy tunneling and more.

All the functionality provided by libcurl can used through the pycurl interface. The following subsections describe how to use the pycurl interface, and assume familiarity with how libcurl works. For information on how libcurl works, please consult the curl library C API.

Contents:

CHAPTER 1

Release Notes

1.1 PycURL 7.21.5 - 2016-01-05

Highlights of this release:

1. Socket callbacks are now fully implemented (`CURLOPT_OPENSOCKETFUNCTION`, `CURLOPT_SOCKOPTFUNCTION`, `CURLOPT_CLOSESOCKETFUNCTION`). Unfortunately this required changing `OPENSOCKETFUNCTION` API once again in a backwards-incompatible manner. Support for `SOCKOPTFUNCTION` and `CLOSESOCKETFUNCTION` was added in this release. `OPENSOCKETFUNCTION` now supports Unix sockets.
2. Many other libcurl options and constants have been added to PycURL.
3. When `pycurl` module initialization fails, `ImportError` is raised instead of a fatal error terminating the process.
4. Usability of official Windows builds has been greatly improved:
   - Dependencies are linked statically, eliminating possible DLL conflicts.
   - OpenSSL is used instead of WinSSL.
   - libcurl is linked against C-Ares and libssh2.

1.2 PycURL 7.19.5.3 - 2015-11-03

PycURL 7.19.5.2 release did not include some of the test suite files in its manifest, leading to inability to run the test suite from the sdist tarball. This is now fixed thanks to Kamil Dudka.

1.3 PycURL 7.19.5.2 - 2015-11-02

Breaking change: DEBUGFUNCTION now takes bytes rather than (Unicode) string as its argument on Python 3.

Breaking change: CURLMOPT_* option constants moved from Easy to Multi class. They remain available in pycurl module.

SSL library detection improved again, –libcurl-dll option to setup.py added.

Options that required tuples now also accept lists, and vice versa.

This release fixes several memory leaks and one use after free issue.

Support for several new libcurl options and constants has been added.

1.4 PycURL 7.19.5.1 - 2015-01-06

This release primarily fixes build breakage against libcurl 7.19.4 through 7.21.1, such as versions shipped with CentOS.

1.5 PycURL 7.19.5 - 2014-07-12

PycURL C code has been significantly reorganized. Curl, CurlMulti and CurlShare classes are now properly exported, instead of factory functions for the respective objects. PycURL API has not changed.

Documentation has been transitioned to Sphinx and reorganized as well. Both docstrings and standalone documentation are now more informative.

Documentation is no longer included in released distributions. It can be generated from source by running *make docs*.

Tests are no longer included in released distributions. Instead the documentation and quickstart examples should be consulted for sample code.

Official Windows builds now are linked against zlib.

1.6 PycURL 7.19.3.1 - 2014-02-05

This release restores PycURL's ability to automatically detect SSL library in use in most circumstances, thanks to Andjelko Horvat.

1.7 PycURL 7.19.3 - 2014-01-09

This release brings official Python 3 support to PycURL. Several GNU/Linux distributions provided Python 3 packages of PycURL previously; these packages were based on patches that were incomplete and in some places incorrect. Behavior of PycURL 7.19.3 and later may therefore differ from behavior of unofficial Python 3 packages of previous PycURL versions.

To summarize the behavior under Python 3, PycURL will accept `bytes` where it accepted strings under Python 2, and will also accept Unicode strings containing ASCII codepoints only for convenience. Please refer to Unicode and file documentation for further details.

In the interests of compatibility, PycURL will also accept Unicode data on Python 2 given the same constraints as under Python 3.

While Unicode and file handling rules are expected to be sensible for all use cases, and retain backwards compatibility with previous PycURL versions, please treat behavior of this versions under Python 3 as experimental and subject to change.

Another potentially disruptive change in PycURL is the requirement for compile time and runtime SSL backends to match. Please see the readme for how to indicate the SSL backend to setup.py.

CHAPTER 2

PycURL Installation

NOTE: You need Python and libcurl installed on your system to use or build pycurl. Some RPM distributions of curl/libcurl do not include everything necessary to build pycurl, in which case you need to install the developer specific RPM which is usually called curl-dev.

2.1 Distutils

Build and install pycurl with the following commands:

```
(if necessary, become root)
tar -zxvf pycurl-$VER.tar.gz
cd pycurl-$VER
python setup.py install
```

$VER should be substituted with the pycurl version number, e.g. 7.10.5.

Note that the installation script assumes that 'curl-config' can be located in your path setting. If curl-config is installed outside your path or you want to force installation to use a particular version of curl-config, use the '–curl-config' command line option to specify the location of curl-config. Example:

```
python setup.py install --curl-config=/usr/local/bin/curl-config
```

If libcurl is linked dynamically with pycurl, you may have to alter the LD_LIBRARY_PATH environment variable accordingly. This normally applies only if there is more than one version of libcurl installed, e.g. one in /usr/lib and one in /usr/local/lib.

2.1.1 SSL

PycURL requires that the SSL library that it is built against is the same one libcurl, and therefore PycURL, uses at runtime. PycURL's `setup.py` uses `curl-config` to attempt to figure out which SSL library libcurl was compiled against, however this does not always work. If PycURL is unable to determine the SSL library in use it will print a warning similar to the following:

```
src/pycurl.c:137:4: warning: #warning "libcurl was compiled with SSL support, but configure could not
```

It will then fail at runtime as follows:

```
ImportError: pycurl: libcurl link-time ssl backend (openssl) is different from compile-time ssl backe
```

To fix this, you need to tell `setup.py` what SSL backend is used:

```
python setup.py --with-[openssl|gnutls|nss] install
```

Note: as of PycURL 7.21.5, setup.py accepts `--with-openssl` option to indicate that libcurl is built against OpenSSL. `--with-ssl` is an alias for `--with-openssl` and continues to be accepted for backwards compatibility.

You can also ask `setup.py` to obtain SSL backend information from installed libcurl shared library, as follows:

> python setup.py –libcurl-dll=libcurl.so

An unqualified `libcurl.so` would use the system libcurl, or you can specify a full path.

2.2 easy_install / pip

```
easy_install pycurl
pip install pycurl
```

If you need to specify an alternate curl-config, it can be done via an environment variable:

```
export PYCURL_CURL_CONFIG=/usr/local/bin/curl-config
easy_install pycurl
```

The same applies to the SSL backend, if you need to specify it (see the SSL note above):

```
export PYCURL_SSL_LIBRARY=[openssl|gnutls|nss]
easy_install pycurl
```

2.2.1 pip and cached pycurl package

If you have already installed pycurl and are trying to reinstall it via pip with different SSL options for example, pip may reinstall the package it has previously compiled instead of recompiling pycurl with newly specified options. More details are given in this Stack Overflow post.

To force pip to recompile pycurl, run:

```
# upgrade pip if necessary
pip install --upgrade pip

# remove current pycurl
pip remove pycurl

# set PYCURL_SSL_LIBRARY
export PYCURL_SSL_LIBRARY=nss

# recompile and install pycurl
pip install --compile pycurl
```

2.3 Windows

2.3.1 Binary Packages

Binary packages are available in the download area for some Windows and Python version combinations. Currently, 32-bit packages are available for Python 2.6, 2.7, 3.2 and 3.3. 64-bit packages are not presently available.

In order to use the official binary packages, your installation of Python must have been compiled against the same MS Visual C++ runtime that the packages have been compiled against. Importantly, which version of MSVC is used has changed in minor releases of Python, for example between 2.7.3 and 2.7.6. As such, you may need to upgrade or downgrade your version of Python to use official PycURL packages.

Currently official PycURL packages are built against the following Python versions:

- 2.6.6
- 2.7.6
- 3.2.5
- 3.3.5

If CRTs used by PycURL and Python do not match, you will receive a message like following when trying to import pycurl module:

```
ImportError: DLL load failed: The specified procedure could not be found.
```

To troubleshoot this situation use the application profiling feature of Dependency Walker and look for msvcrt.dll variants being loaded. You may find the entire thread starting here helpful.

2.3.2 Installing From Source

First, you will need to obtain dependencies. These can be precompiled binaries or source packages that you are going to compile yourself.

For a minimum build you will just need libcurl source. Follow its Windows build instructions to build either a static or a DLL version of the library, then configure PycURL as follows to use it:

```
python setup.py --curl-dir=c:\dev\curl-7.33.0\builds\libcurl-vc-x86-release-dll-ipv6-sspi-spnego-wins
```

Note that `--curl-dir` does not point to libcurl source but rather to headers and compiled libraries.

If libcurl and Python are not linked against the same exact C runtime (version number, static/dll, single-threaded/multi-threaded) you must use `--avoid-stdio` option (see below).

Additional Windows setup.py options:

- `--use-libcurl-dll`: build against libcurl DLL, if not given PycURL will be built against libcurl statically.
- `--libcurl-lib-name=libcurl_imp.lib`: specify a different name for libcurl import library. The default is `libcurl.lib` which is appropriate for static linking and is sometimes the correct choice for dynamic linking as well. The other possibility for dynamic linking is `libcurl_imp.lib`.
- `--with-openssl`: use OpenSSL crypto locks when libcurl was built against OpenSSL.
- `--with-ssl`: legacy alias for `--with-openssl`.
- `--avoid-stdio`: on Windows, a process and each library it is using may be linked to its own version of the C runtime (msvcrt). FILE pointers from one C runtime may not be passed to another C runtime. This option prevents direct passing of FILE pointers from Python to libcurl, thus permitting Python and libcurl to be linked against different C runtimes. This option may carry a performance penalty when Python file objects are given directly to PycURL in CURLOPT_READDATA, CURLOPT_WRITEDATA or CURLOPT_WRITEHEADER options. This option applies only on Python 2; on Python 3, file objects no longer expose C library FILE pointers and the C runtime issue does not exist. On Python 3, this option is recognized but does nothing. You can also give `--avoid-stdio` option in PYCURL_SETUP_OPTIONS environment variable as follows:

```
    PYCURL_SETUP_OPTIONS=--avoid-stdio pip install pycurl
```

A good `setup.py` target to use is `bdist_wininst` which produces an executable installer that you can run to install PycURL.

You may find the following mailing list posts helpful:

- http://curl.haxx.se/mail/curlpython-2009-11/0010.html
- http://curl.haxx.se/mail/curlpython-2013-11/0002.html

2.3.3 winbuild.py

This script is used to build official PycURL Windows packages. You can use it to build a full complement of packages with your own options or modify it to build a single package you need.

Prerequisites:

- msysgit.
- Appropriate Python versions installed.
- MS Visual C++ 9/2008 for Python <= 3.2, MS Visual C++ 10/2010 for Python >= 3.3. Express versions of Visual Studio work fine for this.

`winbuild.py` assumes all programs are installed in their default locations, if this is not the case edit it as needed. `winbuild.py` itself can be run with any Python it supports - 2.6, 2.7, 3.2, 3.3 or 3.4.

2.4 Git Checkout

In order to build PycURL from a Git checkout, some files need to be generated. On Unix systems it is easiest to build PycURL with `make`:

```
make
```

To specify which curl or SSL backend to compile against, use the same environment variables as easy_install/pip, namely `PYCURL_CURL_CONFIG` and `PYCURL_SSL_LIBRARY`.

To generate generated files only you may run:

```
make gen
```

This might be handy if you are on Windows. Remember to run `make gen` whenever you change sources.

To generate documentation, run:

```
make docs
```

Generating documentation requires Sphinx to be installed.

2.5 A Note Regarding SSL Backends

libcurl's functionality varies depending on which SSL backend it is compiled against. For example, users have reported problems with GnuTLS backend. As of this writing, generally speaking, OpenSSL backend has the most functionality as well as the best compatibility with other software.

If you experience SSL issues, especially if you are not using OpenSSL backend, you can try rebuilding libcurl and PycURL against another SSL backend.

CHAPTER 3

PycURL Quick Start

3.1 Retrieving A Network Resource

Once PycURL is installed we can perform network operations. The simplest one is retrieving a resource by its URL. To issue a network request with PycURL, the following steps are required:

1. Create a `pycurl.Curl` instance.
2. Use `setopt` to set options.
3. Call `perform` to perform the operation.

Here is how we can retrieve a network resource in Python 2:

```
import pycurl
from StringIO import StringIO

buffer = StringIO()
c = pycurl.Curl()
c.setopt(c.URL, 'http://pycurl.sourceforge.net/')
c.setopt(c.WRITEDATA, buffer)
c.perform()
c.close()

body = buffer.getvalue()
# Body is a string in some encoding.
# In Python 2, we can print it without knowing what the encoding is.
print(body)
```

This code is available as `examples/quickstart/get_python2.py`.

PycURL does not provide storage for the network response - that is the application's job. Therefore we must setup a buffer (in the form of a StringIO object) and instruct PycURL to write to that buffer.

Most of the existing PycURL code uses WRITEFUNCTION instead of WRITEDATA as follows:

```
c.setopt(c.WRITEFUNCTION, buffer.write)
```

While the WRITEFUNCTION idiom continues to work, it is now unnecessary. As of PycURL 7.19.3 WRITEDATA accepts any Python object with a `write` method.

Python 3 version is slightly more complicated:

```
import pycurl
from io import BytesIO

```

```
buffer = BytesIO()
c = pycurl.Curl()
c.setopt(c.URL, 'http://pycurl.sourceforge.net/')
c.setopt(c.WRITEDATA, buffer)
c.perform()
c.close()

body = buffer.getvalue()
# Body is a byte string.
# We have to know the encoding in order to print it to a text file
# such as standard output.
print(body.decode('iso-8859-1'))
```

This code is available as `examples/quickstart/get_python3.py`.

In Python 3, PycURL response the response body as a byte string. This is handy if we are downloading a binary file, but for text documents we must decode the byte string. In the above example, we assume that the body is encoded in iso-8859-1.

Python 2 and Python 3 versions can be combined. Doing so requires decoding the response body as in Python 3 version. The code for the combined example can be found in `examples/quickstart/get.py`.

3.2 Examining Response Headers

In reality we want to decode the response using the encoding specified by the server rather than assuming an encoding. To do this we need to examine the response headers:

```
import pycurl
import re
try:
    from io import BytesIO
except ImportError:
    from StringIO import StringIO as BytesIO

headers = {}
def header_function(header_line):
    # HTTP standard specifies that headers are encoded in iso-8859-1.
    # On Python 2, decoding step can be skipped.
    # On Python 3, decoding step is required.
    header_line = header_line.decode('iso-8859-1')

    # Header lines include the first status line (HTTP/1.x ...).
    # We are going to ignore all lines that don't have a colon in them.
    # This will botch headers that are split on multiple lines...
    if ':' not in header_line:
        return

    # Break the header line into header name and value.
    name, value = header_line.split(':', 1)

    # Remove whitespace that may be present.
    # Header lines include the trailing newline, and there may be whitespace
    # around the colon.
    name = name.strip()
    value = value.strip()

    # Header names are case insensitive.
    # Lowercase name here.
    name = name.lower()

    # Now we can actually record the header name and value.
    headers[name] = value

buffer = BytesIO()
c = pycurl.Curl()
c.setopt(c.URL, 'http://pycurl.sourceforge.net')
c.setopt(c.WRITEFUNCTION, buffer.write)
# Set our header function.
c.setopt(c.HEADERFUNCTION, header_function)
c.perform()
c.close()

# Figure out what encoding was sent with the response, if any.
# Check against lowercased header name.
encoding = None
if 'content-type' in headers:
    content_type = headers['content-type'].lower()
    match = re.search('charset=(\S+)', content_type)
    if match:
        encoding = match.group(1)
        print('Decoding using %s' % encoding)
if encoding is None:
    # Default encoding for HTML is iso-8859-1.
    # Other content types may have different default encoding,
    # or in case of binary data, may have no encoding at all.
    encoding = 'iso-8859-1'
    print('Assuming encoding is %s' % encoding)

body = buffer.getvalue()
# Decode using the encoding we figured out.
print(body.decode(encoding))
```

This code is available as `examples/quickstart/response_headers.py`.

That was a lot of code for something very straightforward. Unfortunately, as libcurl refrains from allocating memory for response data, it is on our application to perform this grunt work.

3.3 Writing To A File

Suppose we want to save response body to a file. This is actually easy for a change:

```
import pycurl

# As long as the file is opened in binary mode, both Python 2 and Python 3
# can write response body to it without decoding.
with open('out.html', 'wb') as f:
    c = pycurl.Curl()
    c.setopt(c.URL, 'http://pycurl.sourceforge.net/')
    c.setopt(c.WRITEDATA, f)
    c.perform()
    c.close()
```

This code is available as `examples/quickstart/write_file.py`.

The important part is opening the file in binary mode - then response body can be written bytewise without decoding or encoding steps.

3.4 Following Redirects

By default libcurl, and PycURL, do not follow redirects. Changing this behavior involves using `setopt` like so:

```
import pycurl

c = pycurl.Curl()
# Redirects to https://www.python.org/.
c.setopt(c.URL, 'http://www.python.org/')
# Follow redirect.
c.setopt(c.FOLLOWLOCATION, True)
c.perform()
c.close()
```

This code is available as `examples/quickstart/follow_redirect.py`.

As we did not set a write callback, the default libcurl and PycURL behavior to write response body to standard output takes effect.

3.5 Setting Options

Following redirects is one option that libcurl provides. There are many more such options, and they are documented on curl_easy_setopt page. With very few exceptions, PycURL option names are derived from libcurl option names by removing the `CURLOPT_` prefix. Thus, `CURLOPT_URL` becomes simply `URL`.

3.6 Examining Response

We already covered examining response headers. Other response information is accessible via `getinfo` call as follows:

```
import pycurl
try:
    from io import BytesIO
except ImportError:
    from StringIO import StringIO as BytesIO

buffer = BytesIO()
c = pycurl.Curl()
c.setopt(c.URL, 'http://pycurl.sourceforge.net/')
c.setopt(c.WRITEDATA, buffer)
c.perform()

# HTTP response code, e.g. 200.
print('Status: %d' % c.getinfo(c.RESPONSE_CODE))
# Elapsed time for the transfer.
print('Status: %f' % c.getinfo(c.TOTAL_TIME))

# getinfo must be called before close.
c.close()
```

This code is available as `examples/quickstart/response_info.py`.

Here we write the body to a buffer to avoid printing uninteresting output to standard out.

Response information that libcurl exposes is documented on curl_easy_getinfo page. With very few exceptions, PycURL constants are derived from libcurl constants by removing the `CURLINFO_` prefix. Thus, `CURLINFO_RESPONSE_CODE` becomes simply `RESPONSE_CODE`.

3.7 Sending Form Data

To send form data, use `POSTFIELDS` option. Form data must be URL-encoded beforehand:

```
import pycurl
try:
    # python 3
    from urllib.parse import urlencode
except ImportError:
    # python 2
    from urllib import urlencode

c = pycurl.Curl()
c.setopt(c.URL, 'http://pycurl.sourceforge.net/tests/testpostvars.php')

post_data = {'field': 'value'}
# Form data must be provided already urlencoded.
postfields = urlencode(post_data)
# Sets request method to POST,
# Content-Type header to application/x-www-form-urlencoded
# and data to send in request body.
c.setopt(c.POSTFIELDS, postfields)

c.perform()
c.close()
```

This code is available as `examples/quickstart/form_post.py`.

`POSTFIELDS` automatically sets HTTP request method to POST. Other request methods can be specified via `CUSTOMREQUEST` option:

```
c.setopt(c.CUSTOMREQUEST, 'PATCH')
```

3.8 File Upload

To upload a file, use `HTTPPOST` option. To upload a physical file, use `FORM_FILE` as follows:

```
import pycurl

c = pycurl.Curl()
c.setopt(c.URL, 'http://pycurl.sourceforge.net/tests/testfileupload.php')

c.setopt(c.HTTPPOST, [
    ('fileupload', (
        # upload the contents of this file
        c.FORM_FILE, __file__,
    )),
])

c.perform()
c.close()
```

This code is available as `examples/quickstart/file_upload_real.py`.

`libcurl` provides a number of options to tweak file uploads and multipart form submissions in general. These are documented on curl_formadd page. For example, to set a different filename and content type:

```
import pycurl

c = pycurl.Curl()
c.setopt(c.URL, 'http://pycurl.sourceforge.net/tests/testfileupload.php')

c.setopt(c.HTTPPOST, [
    ('fileupload', (
        # upload the contents of this file
        c.FORM_FILE, __file__,
        # specify a different file name for the upload
        c.FORM_FILENAME, 'helloworld.py',
        # specify a different content type
        c.FORM_CONTENTTYPE, 'application/x-python',
    )),
])

c.perform()
c.close()
```

This code is available as `examples/quickstart/file_upload_real_fancy.py`.

If the file data is in memory, use `BUFFER`/`BUFFERPTR` as follows:

```
import pycurl

c = pycurl.Curl()
c.setopt(c.URL, 'http://pycurl.sourceforge.net/tests/testfileupload.php')

c.setopt(c.HTTPPOST, [
    ('fileupload', (
        c.FORM_BUFFER, 'readme.txt',
        c.FORM_BUFFERPTR, 'This is a fancy readme file',
    )),
])

c.perform()
c.close()
```

This code is available as `examples/quickstart/file_upload_buffer.py`.

CHAPTER 4

pycurl Module Functionality

`pycurl.`**`version`**

This is a string with version information on libcurl, corresponding to curl_version in libcurl.

Example usage:

```
>>> import pycurl
>>> pycurl.version
'PycURL/7.19.3 libcurl/7.33.0 OpenSSL/0.9.8x zlib/1.2.7'
```

CHAPTER 5

Curl Object

CHAPTER 6

CurlMulti Object

CHAPTER 7

CurlShare Object

CHAPTER 8

Callbacks

For more fine-grained control, libcurl allows a number of callbacks to be associated with each connection. In pycurl, callbacks are defined using the `setopt()` method for Curl objects with options `WRITEFUNCTION`, `READFUNCTION`, `HEADERFUNCTION`, `PROGRESSFUNCTION`, `IOCTLFUNCTION`, or `DEBUGFUNCTION`. These options correspond to the libcurl options with `CURLOPT_` prefix removed. A callback in pycurl must be either a regular Python function, a class method or an extension type function.

There are some limitations to some of the options which can be used concurrently with the pycurl callbacks compared to the libcurl callbacks. This is to allow different callback functions to be associated with different Curl objects. More specifically, `WRITEDATA` cannot be used with `WRITEFUNCTION`, `READDATA` cannot be used with `READFUNCTION`, `WRITEHEADER` cannot be used with `HEADERFUNCTION`, `PROGRESSDATA` cannot be used with `PROGRESSFUNCTION`, `IOCTLDATA` cannot be used with `IOCTLFUNCTION`, and `DEBUGDATA` cannot be used with `DEBUGFUNCTION`. In practice, these limitations can be overcome by having a callback function be a class instance method and rather use the class instance attributes to store per object data such as files used in the callbacks.

The signature of each callback used in pycurl is documented below.

8.1 WRITEFUNCTION

`WRITEFUNCTION`(*byte string*) → number of characters written

Callback for writing data. Corresponds to CURLOPT_WRITEFUNCTION in libcurl.

On Python 3, the argument is of type `bytes`.

The `WRITEFUNCTION` callback may return the number of bytes written. If this number is not equal to the size of the byte string, this signifies an error and libcurl will abort the request. Returning `None` is an alternate way of indicating that the callback has consumed all of the string passed to it and, hence, succeeded.

write_test.py test shows how to use `WRITEFUNCTION`.

8.1.1 Example: Callbacks for document header and body

This example prints the header data to stderr and the body data to stdout. Also note that neither callback returns the number of bytes written. For WRITEFUNCTION and HEADERFUNCTION callbacks, returning None implies that all bytes where written.

```
## Callback function invoked when body data is ready
def body(buf):
    # Print body data to stdout
    import sys
    sys.stdout.write(buf)
    # Returning None implies that all bytes were written

## Callback function invoked when header data is ready
def header(buf):
    # Print header data to stderr
    import sys
    sys.stderr.write(buf)
    # Returning None implies that all bytes were written

c = pycurl.Curl()
c.setopt(pycurl.URL, "http://www.python.org/")
c.setopt(pycurl.WRITEFUNCTION, body)
c.setopt(pycurl.HEADERFUNCTION, header)
c.perform()
```

8.2 HEADERFUNCTION

HEADERFUNCTION(*byte string*) → number of characters written

Callback for writing received headers. Corresponds to CURLOPT_HEADERFUNCTION in libcurl.

On Python 3, the argument is of type `bytes`.

The `HEADERFUNCTION` callback may return the number of bytes written. If this number is not equal to the size of the byte string, this signifies an error and libcurl will abort the request. Returning `None` is an alternate way of indicating that the callback has consumed all of the string passed to it and, hence, succeeded.

header_test.py test shows how to use `WRITEFUNCTION`.

8.3 READFUNCTION

READFUNCTION(*number of characters to read*) → byte string

Callback for reading data. Corresponds to CURLOPT_READFUNCTION in libcurl.

On Python 3, the callback must return either a byte string or a Unicode string consisting of ASCII code points only.

In addition, `READFUNCTION` may return `READFUNC_ABORT` or `READFUNC_PAUSE`. See the libcurl documentation for an explanation of these values.

The file_upload.py example in the distribution contains example code for using `READFUNCTION`.

8.4 SEEKFUNCTION

SEEKFUNCTION(*offset*, *origin*) → status

Callback for seek operations. Corresponds to CURLOPT_SEEKFUNCTION in libcurl.

8.5 IOCTLFUNCTION

IOCTLFUNCTION(*ioctl cmd*) → status

Callback for I/O operations. Corresponds to CURLOPT_IOCTLFUNCTION in libcurl.

Note: this callback is deprecated. Use *SEEKFUNCTION* instead.

8.6 DEBUGFUNCTION

DEBUGFUNCTION(*debug message type*, *debug message byte string*) → None

Callback for debug information. Corresponds to CURLOPT_DEBUGFUNCTION in libcurl.

Changed in version 7.19.5.2: The second argument to a `DEBUGFUNCTION` callback is now of type `bytes` on Python 3. Previously the argument was of type `str`.

debug_test.py test shows how to use `DEBUGFUNCTION`.

8.6.1 Example: Debug callbacks

This example shows how to use the debug callback. The debug message type is an integer indicating the type of debug message. The VERBOSE option must be enabled for this callback to be invoked.

```
def test(debug_type, debug_msg):
    print "debug(%d): %s" % (debug_type, debug_msg)

c = pycurl.Curl()
c.setopt(pycurl.URL, "http://curl.haxx.se/")
c.setopt(pycurl.VERBOSE, 1)
c.setopt(pycurl.DEBUGFUNCTION, test)
c.perform()
```

8.7 PROGRESSFUNCTION

PROGRESSFUNCTION(*download total*, *downloaded*, *upload total*, *uploaded*) → status

Callback for progress meter. Corresponds to CURLOPT_PROGRESSFUNCTION in libcurl.

8.7.1 Example: Download/upload progress callback

This example shows how to use the progress callback. When downloading a document, the arguments related to uploads are zero, and vice versa.

```
## Callback function invoked when download/upload has progress
def progress(download_t, download_d, upload_t, upload_d):
    print "Total to download", download_t
    print "Total downloaded", download_d
    print "Total to upload", upload_t
    print "Total uploaded", upload_d

c = pycurl.Curl()
c.setopt(c.URL, "http://slashdot.org/")
c.setopt(c.NOPROGRESS, 0)
c.setopt(c.PROGRESSFUNCTION, progress)
c.perform()
```

8.8 OPENSOCKETFUNCTION

OPENSOCKETFUNCTION(*purpose*, *address*) → int

Callback for opening sockets. Corresponds to CURLOPT_OPENSOCKETFUNCTION in libcurl.

purpose is a `SOCKTYPE_*` value.

address is a namedtuple with `family`, `socktype`, `protocol` and `addr` fields, per CURLOPT_OPENSOCKETFUNCTION documentation.

addr is an object representing the address. Currently the following address families are supported:

- `AF_INET`: *addr* is a 2-tuple of `(host, port)`.
- `AF_INET6`: *addr* is a 4-tuple of `(host, port, flow info, scope id)`.
- `AF_UNIX`: *addr* is a byte string containing path to the Unix socket.

  Availability: Unix.

This behavior matches that of Python's socket module.

The callback should return a socket object, a socket file descriptor or a Python object with a `fileno` property containing the socket file descriptor.

The callback may be unset by calling setopt with `None` as the value or by calling unsetopt.

open_socket_cb_test.py test shows how to use `OPENSOCKETFUNCTION`.

Changed in version 7.21.5: Previously, the callback received `family`, `socktype`, `protocol` and `addr` parameters (`purpose` was not passed and `address` was flattened). Also, `AF_INET6` addresses were exposed as 2-tuples of `(host, port)` rather than 4-tuples.

Changed in version 7.19.3: `addr` parameter added to the callback.

8.9 CLOSESOCKETFUNCTION

CLOSESOCKETFUNCTION(*curlfd*) → int

Callback for setting socket options. Corresponds to CURLOPT_CLOSESOCKETFUNCTION in libcurl.

curlfd is the file descriptor to be closed.

The callback should return an `int`.

The callback may be unset by calling setopt with `None` as the value or by calling unsetopt.

close_socket_cb_test.py test shows how to use `CLOSESOCKETFUNCTION`.

8.10 SOCKOPTFUNCTION

SOCKOPTFUNCTION(*curlfd*, *purpose*) → int

Callback for setting socket options. Corresponds to CURLOPT_SOCKOPTFUNCTION in libcurl.

curlfd is the file descriptor of the newly created socket.

purpose is a `SOCKTYPE_*` value.

The callback should return an `int`.

The callback may be unset by calling setopt with `None` as the value or by calling unsetopt.

sockopt_cb_test.py test shows how to use `SOCKOPTFUNCTION`.

8.11 SSH_KEYFUNCTION

SSH_KEYFUNCTION(*known_key*, *found_key*, *match*) → int

Callback for known host matching logic. Corresponds to CURLOPT_SSH_KEYFUNCTION in libcurl.

known_key and *found_key* are instances of `KhKey` class which is a namedtuple with `key` and `keytype` fields, corresponding to libcurl's `struct curl_khkey`:

```
KhKey = namedtuple('KhKey', ('key', 'keytype'))
```

On Python 2, the *key* field of `KhKey` is a `str`. On Python 3, the *key* field is `bytes`. *keytype* is an `int`.

known_key may be `None` when there is no known matching host key.

`SSH_KEYFUNCTION` callback should return a `KHSTAT_*` value.

The callback may be unset by calling setopt with `None` as the value or by calling unsetopt.

ssh_key_cb_test.py test shows how to use `SSH_KEYFUNCTION`.

CHAPTER 9

curl Module Functionality

9.1 High Level Curl Object

CHAPTER 10

String And Unicode Handling

Generally speaking, libcurl does not perform data encoding or decoding. In particular, libcurl is not Unicode-aware, but operates on byte streams. libcurl leaves it up to the application - PycURL library or an application using PycURL in this case - to encode and decode Unicode data into byte streams.

PycURL, being a thin wrapper around libcurl, generally does not perform this encoding and decoding either, leaving it up to the application. Specifically:

- Data that PycURL passes to an application, such as via callback functions, is normally byte strings. The application must decode them to obtain text (Unicode) data.
- Data that an application passes to PycURL, such as via `setopt` calls, must normally be byte strings appropriately encoded. For convenience and compatibility with existing code, PycURL will accept Unicode strings that contain ASCII code points only [1], and transparently encode these to byte strings.

Why doesn't PycURL automatically encode and decode, say, HTTP request or response data? The key to remember is that libcurl supports over 20 protocols, and PycURL generally has no knowledge of what protocol is being used by a particular request as PycURL does not track application state. Having to manually encode and decode data is unfortunately the price of libcurl's flexibility.

10.1 Setting Options - Python 2.x

Under Python 2, the `str` type can hold arbitrary encoded byte strings. PycURL will pass whatever byte strings it is given verbatim to libcurl. The following code will work:

```
>>> import pycurl
>>> c = pycurl.Curl()
>>> c.setopt(c.USERAGENT, 'Foo\xa9')
# ok
```

Unicode strings can be used but must contain ASCII code points only:

```
>>> c.setopt(c.USERAGENT, u'Foo')
# ok

>>> c.setopt(c.USERAGENT, u'Foo\xa9')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character u'\xa9' in position 3: ordinal not in range
```

[1] Only ASCII is accepted; ISO-8859-1/Latin 1, for example, will be rejected.

```
>>> c.setopt(c.USERAGENT, u'Foo\xa9'.encode('iso-8859-1'))
# ok
```

10.2 Setting Options - Python 3.x

Under Python 3, the `bytes` type holds arbitrary encoded byte strings. PycURL will accept `bytes` values for all options where libcurl specifies a “string” argument:

```
>>> import pycurl
>>> c = pycurl.Curl()
>>> c.setopt(c.USERAGENT, b'Foo\xa9')
# ok
```

The `str` type holds Unicode data. PycURL will accept `str` values containing ASCII code points only:

```
>>> c.setopt(c.USERAGENT, 'Foo')
# ok

>>> c.setopt(c.USERAGENT, 'Foo\xa9')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character '\xa9' in position 3: ordinal not in range(

>>> c.setopt(c.USERAGENT, 'Foo\xa9'.encode('iso-8859-1'))
# ok
```

10.3 Writing To Files

PycURL will return all data read from the network as byte strings. On Python 2, this means the write callbacks will receive `str` objects, and on Python 3, write callbacks will receive `bytes` objects.

Under Python 2, when using e.g. `WRITEDATA` or `WRITEFUNCTION` options, files being written to *should* be opened in binary mode. Writing to files opened in text mode will not raise exceptions but may corrupt data.

Under Python 3, PycURL passes strings and binary data to the application using `bytes` instances. When writing to files, the files must be opened in binary mode for the writes to work:

```
import pycurl
c = pycurl.Curl()
c.setopt(c.URL,'http://pycurl.sourceforge.net')
# File opened in binary mode.
with open('/dev/null','wb') as f:
    c.setopt(c.WRITEDATA, f)
    # Same result if using WRITEFUNCTION instead:
    #c.setopt(c.WRITEFUNCTION, f.write)
    c.perform()
# ok
```

If a file is opened in text mode (`w` instead of `wb` mode), an error similar to the following will result:

```
TypeError: must be str, not bytes
Traceback (most recent call last):
  File "/tmp/test.py", line 8, in <module>
    c.perform()
pycurl.error: (23, 'Failed writing body (0 != 168)')
```

The TypeError is actually an exception raised by Python which will be printed, but not propagated, by PycURL. PycURL will raise a `pycurl.error` to signify operation failure.

10.4 Writing To StringIO/BytesIO

Under Python 2, response can be saved in memory by using a `StringIO` object:

```
import pycurl
from StringIO import StringIO
c = pycurl.Curl()
c.setopt(c.URL,'http://pycurl.sourceforge.net')
buffer = StringIO()
c.setopt(c.WRITEDATA, buffer)
# Same result if using WRITEFUNCTION instead:
#c.setopt(c.WRITEFUNCTION, buffer.write)
c.perform()
# ok
```

Under Python 3, as PycURL invokes the write callback with `bytes` argument, the response must be written to a `BytesIO` object:

```
import pycurl
from io import BytesIO
c = pycurl.Curl()
c.setopt(c.URL,'http://pycurl.sourceforge.net')
buffer = BytesIO()
c.setopt(c.WRITEDATA, buffer)
# Same result if using WRITEFUNCTION instead:
#c.setopt(c.WRITEFUNCTION, buffer.write)
c.perform()
# ok
```

Attempting to use a `StringIO` object will produce an error:

```
import pycurl
from io import StringIO
c = pycurl.Curl()
c.setopt(c.URL,'http://pycurl.sourceforge.net')
buffer = StringIO()
c.setopt(c.WRITEDATA, buffer)
c.perform()

TypeError: string argument expected, got 'bytes'
Traceback (most recent call last):
  File "/tmp/test.py", line 9, in <module>
    c.perform()
pycurl.error: (23, 'Failed writing body (0 != 168)')
```

The following idiom can be used for code that needs to be compatible with both Python 2 and Python 3:

```
import pycurl
try:
    # Python 3
    from io import BytesIO
except ImportError:
    # Python 2
    from StringIO import StringIO as BytesIO
c = pycurl.Curl()
```

```
c.setopt(c.URL,'http://pycurl.sourceforge.net')
buffer = BytesIO()
c.setopt(c.WRITEDATA, buffer)
c.perform()
# ok
# Decode the response body:
string_body = buffer.getvalue().decode('utf-8')
```

10.5 Header Functions

Although headers are often ASCII text, they are still returned as `bytes` instances on Python 3 and thus require appropriate decoding. HTTP headers are encoded in ISO/IEC 8859-1 according to the standards.

When using `WRITEHEADER` option to write headers to files, the files should be opened in binary mode in Python 2 and must be opened in binary mode in Python 3, same as with `WRITEDATA`.

10.6 Read Functions

Read functions are expected to provide data in the same fashion as string options expect it:

- On Python 2, the data can be given as `str` instances, appropriately encoded.
- On Python 2, the data can be given as `unicode` instances containing ASCII code points only.
- On Python 3, the data can be given as `bytes` instances.
- On Python 3. the data can be given as `str` instances containing ASCII code points only.

Caution: when using CURLOPT_READFUNCTION in tandem with CURLOPT_POSTFIELDSIZE, as would be done for HTTP for example, take care to pass the length of *encoded* data to CURLOPT_POSTFIELDSIZE if you are performing the encoding. If you pass the number of Unicode characters rather than encoded bytes to libcurl, the server will receive wrong Content-Length. Alternatively you can return Unicode strings from a CURLOPT_READFUNCTION function, if your data contains only ASCII code points, and let PycURL encode them for you.

10.7 How PycURL Handles Unicode Strings

If PycURL is given a Unicode string which contains non-ASCII code points, and as such cannot be encoded to ASCII, PycURL will return an error to libcurl, and libcurl in turn will fail the request with an error like “read function error/data error”. PycURL will then raise `pycurl.error` with this latter message. The encoding exception that was the underlying cause of the problem is stored as `sys.last_value`.

10.8 Figuring Out Correct Encoding

What encoding should be used when is a complicated question. For example, when working with HTTP:

- URLs and POSTFIELDS data must be URL-encoded. A URL-encoded string has only ASCII code points.
- Headers must be ISO/IEC 8859-1 encoded.
- Encoding for bodies is specified in Content-Type and Content-Encoding headers.

10.9 Legacy PycURL Versions

The Unicode handling documented here was implemented in PycURL 7.19.3 along with Python 3 support. Prior to PycURL 7.19.3 Unicode data was not accepted at all:

```
>>> import pycurl
>>> c = pycurl.Curl()
>>> c.setopt(c.USERAGENT, u'Foo\xa9')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: invalid arguments to setopt
```

Some GNU/Linux distributions provided Python 3 packages of PycURL prior to PycURL 7.19.3. These packages included unofficial patches ([2], [3]) which did not handle Unicode correctly, and did not behave as described in this document. Such unofficial versions of PycURL should be avoided.

[2] http://sourceforge.net/p/pycurl/patches/5/
[3] http://sourceforge.net/p/pycurl/patches/12/

CHAPTER 11

File Handling

In PycURL 7.19.0.3 and below, `CURLOPT_READDATA`, `CURLOPT_WRITEDATA` and `CURLOPT_WRITEHEADER` options accepted file objects and directly passed the underlying C library `FILE` pointers to libcurl.

Python 3 no longer implements files as C library `FILE` objects. In PycURL 7.19.3 and above, when running on Python 3, these options are implemented as calls to `CURLOPT_READFUNCTION`, `CURLOPT_WRITEFUNCTION` and `CURLOPT_HEADERFUNCTION`, respectively, with the write method of the Python file object as the parameter. As a result, any Python file-like object implementing a `read` method can be passed to `CURLOPT_READDATA`, and any Python file-like object implementing a `write` method can be passed to `CURLOPT_WRITEDATA` or `CURLOPT_WRITEHEADER` options.

When running PycURL 7.19.3 and above on Python 2, the old behavior of passing `FILE` pointers to libcurl remains when a true file object is given to `CURLOPT_READDATA`, `CURLOPT_WRITEDATA` and `CURLOPT_WRITEHEADER` options. For consistency with Python 3 behavior these options also accept file-like objects implementing a `read` or `write` method, as appropriate, as arguments, in which case the Python 3 code path is used converting these options to `CURLOPT_*FUNCTION` option calls.

Files given to PycURL as arguments to `CURLOPT_READDATA`, `CURLOPT_WRITEDATA` or `CURLOPT_WRITEHEADER` must be opened for reading or writing in binary mode. Files opened in text mode (without `"b"` flag to `open()`) expect string objects and reading from or writing to them from PycURL will fail. Similarly when passing `f.write` method of an open file to `CURLOPT_WRITEFUNCTION` or `CURLOPT_HEADERFUNCTION`, or `f.read` to `CURLOPT_READFUNCTION`, the file must have been be opened in binary mode.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Python Module Index