zlib — GNU zlib Compression

Purpose: Low-level access to GNU zlib compression library

The `zlib` module provides a lower-level interface to many of the functions in the `zlib` compression library from the GNU project.

Working with Data in Memory

The simplest way to work with `zlib` requires holding all of the data to be compressed or decompressed in memory.

```
# zlib_memory.py

import zlib
import binascii

original_data = b'This is the original text.'
print('Original     :', len(original_data), original_data)

compressed = zlib.compress(original_data)
print('Compressed   :', len(compressed),
      binascii.hexlify(compressed))

decompressed = zlib.decompress(compressed)
print('Decompressed :', len(decompressed), decompressed)
```

The `compress()` and `decompress()` functions both take a byte sequence argument and return a byte sequence.

```
$ python3 zlib_memory.py

Original     : 26 b'This is the original text.'
Compressed   : 32 b'789c0bc9c82c5600a2928c5485fca2ccf4ccbcc41c85
92d48a123d007f2f097e'
Decompressed : 26 b'This is the original text.'
```

The previous example demonstrates that the compressed version of small amounts of data can be larger than the uncompressed version. While the actual results depend on the input data, it is interesting to observe the compression overhead for small data sets

```
# zlib_lengths.py

import zlib

original_data = b'This is the original text.'

template = '{:>15}  {:>15}'
print(template.format('len(data)', 'len(compressed)'))
print(template.format('-' * 15, '-' * 15))

for i in range(5):
    data = original_data * i
    compressed = zlib.compress(data)
    highlight = '*' if len(data) < len(compressed) else ''
    print(template.format(len(data), len(compressed)), highlight)
```

The * in the output highlight the lines where the compressed data takes up more memory than the uncompressed version.

```
$ python3 zlib_lengths.py

      len(data)  len(compressed)
---------------  ---------------
              0                8 *
```

```
            26              32 *
            52              35
            78              35
           104              36
```

`zlib` supports several different compression levels, allowing a balance between computational cost and the amount of space reduction. The default compression level, `zlib.Z_DEFAULT_COMPRESSION` is `-1` and corresponds to a hard-coded value that compromises between performance and compression outcome. This currently corresponds to level 6.

```
# zlib_compresslevel.py

import zlib

input_data = b'Some repeated text.\n' * 1024
template = '{:>5}  {:>5}'

print(template.format('Level', 'Size'))
print(template.format('-----', '----'))

for i in range(0, 10):
    data = zlib.compress(input_data, i)
    print(template.format(i, len(data)))
```

A level of 0 means no compression at all. A level of 9 requires the most computation and produces the smallest output. As this example shows, the same size reduction may be achieved with multiple compression levels for a given input.

```
$ python3 zlib_compresslevel.py

Level   Size
-----   ----
    0  20491
    1    172
    2    172
    3    172
    4     98
    5     98
    6     98
    7     98
    8     98
    9     98
```

Incremental Compression and Decompression

The in-memory approach has drawbacks that make it impractical for real-world use cases, primarily that the system needs enough memory to hold both the uncompressed and compressed versions resident in memory at the same time. The alternative is to use `Compress` and `Decompress` objects to manipulate data incrementally, so that the entire data set does not have to fit into memory.

```
# zlib_incremental.py

import zlib
import binascii

compressor = zlib.compressobj(1)

with open('lorem.txt', 'rb') as input:
    while True:
        block = input.read(64)
        if not block:
            break
        compressed = compressor.compress(block)
        if compressed:
            print('Compressed: {}'.format(
                binascii.hexlify(compressed)))
        else:
            print('buffering...')
    remaining = compressor.flush()
    print('Flushed: {}'.format(binascii.hexlify(remaining)))
```

This example reads small blocks of data from a plain text file and passes it to `compress()`. The compressor maintains an internal buffer of compressed data. Since the compression algorithm depends on checksums and minimum block sizes, the compressor may not be ready to return data each time it receives more input. If it does not have an entire compressed block ready, it returns an empty byte string. When all of the data is fed in, the `flush()` method forces the compressor to close the final block and return the rest of the compressed data.

```
$ python3 zlib_incremental.py

Compressed: b'7801'
buffering...
buffering...
buffering...
buffering...
buffering...
Flushed: b'55904b6ac4400c44f73e451da0f129b20c2110c85e696b8c40dde
dd167ce1f7915025a087daa9ef4be8c07e4f21c38962e834b800647435fd3b90
747b2810eb9c4bbcc13ac123bded6e4bef1c91ee40d3c6580e3ff52aad2e8cb2
eb6062dad74a89ca904cbb0f2545e0db4b1f2e01955b8c511cb2ac08967d228a
f1447c8ec72e40c4c714116e60cdef171bb6c0feaa255dff1c507c2c4439ec96
05b7e0ba9fc54bae39355cb89fd6ebe5841d673c7b7bc68a46f575a312eebd22
0d4b32441bdc1b36ebf0aedef3d57ea4b26dd986dd39af57dfb05d32279de'
```

Mixed Content Streams

The `Decompress` class returned by `decompressobj()` can also be used in situations where compressed and uncompressed data is mixed together.

```
# zlib_mixed.py
```

```
import zlib

lorem = open('lorem.txt', 'rb').read()
compressed = zlib.compress(lorem)
combined = compressed + lorem

decompressor = zlib.decompressobj()
decompressed = decompressor.decompress(combined)

decompressed_matches = decompressed == lorem
print('Decompressed matches lorem:', decompressed_matches)

unused_matches = decompressor.unused_data == lorem
print('Unused data matches lorem :', unused_matches)
```

After decompressing all of the data, the `unused_data` attribute contains any data not used.

```
$ python3 zlib_mixed.py

Decompressed matches lorem: True
Unused data matches lorem : True
```

Checksums

In addition to compression and decompression functions, `zlib` includes two functions for computing checksums of data, `adler32()` and `crc32()`. Neither checksum is cryptographically secure, and they are only intended for use for data integrity verification.

```
# zlib_checksums.py
```

```
import zlib

data = open('lorem.txt', 'rb').read()

cksum = zlib.adler32(data)
print('Adler32: {:12d}'.format(cksum))
print('       : {:12d}'.format(zlib.adler32(data, cksum)))
```

```
cksum = zlib.crc32(data)
print('CRC-32 : {:12d}'.format(cksum))
print('       : {:12d}'.format(zlib.crc32(data, cksum)))
```

Both functions take the same arguments, a byte string containing the data and an optional value to be used as a starting point for the checksum. They return a 32-bit signed integer value which can also be passed back on subsequent calls as a new starting point argument to produce a *running* checksum.

```
$ python3 zlib_checksums.py

Adler32:   3542251998
       :    669447099
CRC-32 :   3038370516
       :   2870078631
```

Compressing Network Data

The server in the next listing uses the stream compressor to respond to requests consisting of filenames by writing a compressed version of the file to the socket used to communicate with the client.

```
# zlib_server.py

import zlib
import logging
import socketserver
import binascii

BLOCK_SIZE = 64


class ZlibRequestHandler(socketserver.BaseRequestHandler):

    logger = logging.getLogger('Server')

    def handle(self):
        compressor = zlib.compressobj(1)

        # Find out what file the client wants
        filename = self.request.recv(1024).decode('utf-8')
        self.logger.debug('client asked for: %r', filename)

        # Send chunks of the file as they are compressed
        with open(filename, 'rb') as input:
            while True:
                block = input.read(BLOCK_SIZE)
                if not block:
                    break
                self.logger.debug('RAW %r', block)
                compressed = compressor.compress(block)
                if compressed:
                    self.logger.debug(
                        'SENDING %r',
                        binascii.hexlify(compressed))
                    self.request.send(compressed)
                else:
                    self.logger.debug('BUFFERING')

        # Send any data being buffered by the compressor
        remaining = compressor.flush()
        while remaining:
            to_send = remaining[:BLOCK_SIZE]
            remaining = remaining[BLOCK_SIZE:]
            self.logger.debug('FLUSHING %r',
                              binascii.hexlify(to_send))
            self.request.send(to_send)
        return


if __name__ == '__main__':
    import socket
```

```
import socket
import threading
from io import BytesIO

logging.basicConfig(
    level=logging.DEBUG,
    format='%(name)s: %(message)s',
)
logger = logging.getLogger('Client')

# Set up a server, running in a separate thread
address = ('localhost', 0)  # let the kernel assign a port
server = socketserver.TCPServer(address, ZlibRequestHandler)
ip, port = server.server_address  # what port was assigned?

t = threading.Thread(target=server.serve_forever)
t.setDaemon(True)
t.start()

# Connect to the server as a client
logger.info('Contacting server on %s:%s', ip, port)
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip, port))

# Ask for a file
requested_file = 'lorem.txt'
logger.debug('sending filename: %r', requested_file)
len_sent = s.send(requested_file.encode('utf-8'))

# Receive a response
buffer = BytesIO()
decompressor = zlib.decompressobj()
while True:
    response = s.recv(BLOCK_SIZE)
    if not response:
        break
    logger.debug('READ %r', binascii.hexlify(response))

    # Include any unconsumed data when
    # feeding the decompressor.
    to_decompress = decompressor.unconsumed_tail + response
    while to_decompress:
        decompressed = decompressor.decompress(to_decompress)
        if decompressed:
            logger.debug('DECOMPRESSED %r', decompressed)
            buffer.write(decompressed)
            # Look for unconsumed data due to buffer overflow
            to_decompress = decompressor.unconsumed_tail
        else:
            logger.debug('BUFFERING')
            to_decompress = None

# deal with data reamining inside the decompressor buffer
remainder = decompressor.flush()
if remainder:
    logger.debug('FLUSHED %r', remainder)
    buffer.write(remainder)

full_response = buffer.getvalue()
lorem = open('lorem.txt', 'rb').read()
logger.debug('response matches file contents: %s',
             full_response == lorem)

# Clean up
s.close()
server.socket.close()
```

It has some artificial chunking in place to illustrate the buffering behavior that happens when the data passed to `compress()` or `decompress()` does not result in a complete block of compressed or uncompressed output.

The client connects to the socket and requests a file. Then it loops, receiving blocks of compressed data. Since a block may not contain enough information to decompress it entirely, the remainder of any data received earlier is combined with the new

data and passed to the decompressor. As the data is decompressed, it is appended to a buffer, which is compared against the file contents at the end of the processing loop.

Warning

This server has obvious security implications. Do not run it on a system on the open Internet or in any environment where security might be an issue.

```
$ python3 zlib_server.py

Client: Contacting server on 127.0.0.1:53658
Client: sending filename: 'lorem.txt'
Server: client asked for: 'lorem.txt'
Server: RAW b'Lorem ipsum dolor sit amet, consectetuer adipiscin
g elit. Donec\n'
Server: SENDING b'7801'
Server: RAW b'egestas, enim et consectetuer ullamcorper, lectus
ligula rutrum '
Server: BUFFERING
Server: RAW b'leo, a\nelementum elit tortor eu quam. Duis tincid
unt nisi ut ant'
Server: BUFFERING
Server: RAW b'e. Nulla\nfacilisi. Sed tristique eros eu libero.
Pellentesque ve'
Server: BUFFERING
Server: RAW b'l arcu. Vivamus\npurus orci, iaculis ac, suscipit
sit amet, pulvi'
Client: READ b'7801'
Client: BUFFERING
Server: BUFFERING
Server: RAW b'nar eu,\nlacus.\n'
Server: BUFFERING
Server: FLUSHING b'55904b6ac4400c44f73e451da0f129b20c2110c85e696
b8c40ddedd167ce1f7915025a087daa9ef4be8c07e4f21c38962e834b8006474
35fd3b90747b2810eb9'
Server: FLUSHING b'c4bbcc13ac123bded6e4bef1c91ee40d3c6580e3ff52a
ad2e8cb2eb6062dad74a89ca904cbb0f2545e0db4b1f2e01955b8c511cb2ac08
967d228af1447c8ec72'
Client: READ b'55904b6ac4400c44f73e451da0f129b20c2110c85e696b8c4
0ddedd167ce1f7915025a087daa9ef4be8c07e4f21c38962e834b800647435fd
3b90747b2810eb9'
Server: FLUSHING b'e40c4c714116e60cdef171bb6c0feaa255dff1c507c2c
4439ec9605b7e0ba9fc54bae39355cb89fd6ebe5841d673c7b7bc68a46f575a3
12eebd220d4b32441bd'
Client: DECOMPRESSED b'Lorem ipsum dolor sit amet, consectetuer
adi'
Client: READ b'c4bbcc13ac123bded6e4bef1c91ee40d3c6580e3ff52aad2e
8cb2eb6062dad74a89ca904cbb0f2545e0db4b1f2e01955b8c511cb2ac08967d
228af1447c8ec72'
Client: DECOMPRESSED b'piscing elit. Donec\negestas, enim et con
sectetuer ullamcorper, lectus ligula rutrum leo, a\nelementum el
it tortor eu quam. Duis tinci'
Client: READ b'e40c4c714116e60cdef171bb6c0feaa255dff1c507c2c4439
ec9605b7e0ba9fc54bae39355cb89fd6ebe5841d673c7b7bc68a46f575a312ee
bd220d4b32441bd'
Client: DECOMPRESSED b'dunt nisi ut ante. Nulla\nfacilisi. Sed t
ristique eros eu libero. Pellentesque vel arcu. Vivamus\npurus o
rci, iaculis ac'
Server: FLUSHING b'c1b36ebf0aedef3d57ea4b26dd986dd39af57dfb05d32
279de'
Client: READ b'c1b36ebf0aedef3d57ea4b26dd986dd39af57dfb05d32279d
e'
Client: DECOMPRESSED b', suscipit sit amet, pulvinar eu,\nlacus.
\n'
Client: response matches file contents: True
```

See also

- Standard library documentation for zlib

- gzip – The gzip module includes a higher level (file-based) interface to the zlib library.
- http://www.zlib.net/ – Home page for zlib library.
- http://www.zlib.net/manual.html – Complete zlib documentation.
- bz2 – The bz2 module provides a similar interface to the bzip2 compression library.

Data Compression and Archiving

gzip — Read and Write GNU zip Files

Quick Links

This page was last updated 2016-12-18.

Navigation

Data Compression and Archiving
gzip — Read and Write GNU zip Files

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