bz2 — bzip2 Compression

Purpose: bzip2 compression

The bz2 module is an interface for the bzip2 library, used to compress data for storage or transmission. There are three APIs provided:

- “one shot” compression/decompression functions for operating on a blob of data
- iterative compression/decompression objects for working with streams of data
- a file-like class that supports reading and writing as with an uncompressed file

One-shot Operations in Memory

The simplest way to work with bz2 is to load all of the data to be compressed or decompressed in memory, and then use `compress()` and `decompress()` to transform it.

```
# bz2_memory.py

import bz2
import binascii

original_data = b'This is the original text.'
print('Original     : {} bytes'.format(len(original_data)))
print(original_data)

print()
compressed = bz2.compress(original_data)
print('Compressed   : {} bytes'.format(len(compressed)))
hex_version = binascii.hexlify(compressed)
for i in range(len(hex_version) // 40 + 1):
    print(hex_version[i * 40:(i + 1) * 40])

print()
decompressed = bz2.decompress(compressed)
print('Decompressed : {} bytes'.format(len(decompressed)))
print(decompressed)
```

The compressed data contains non-ASCII characters, so it needs to be converted to its hexadecimal representation before it can be printed. In the output from these examples, the hexadecimal version is reformatted to have at most 40 characters on each line.

```
$ python3 bz2_memory.py

Original     : 26 bytes
b'This is the original text.'

Compressed   : 62 bytes
b'425a68393141592653 59 16be35a6000002938040'
b'01040022e59c402000314c000111e93d434da223'
b'028cf9e73148cae0a0d6ed7f17724538509016be'
b'35a6'

Decompressed : 26 bytes
b'This is the original text.'
```

For short text, the compressed version can be significantly longer than the original. While the actual results depend on the input data, it is interesting to observe the compression overhead.

```
# bz2_lengths.py

import bz2

original_data = b'This is the original text.'
```

```
original_data = b'This is the original text.'

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

for i in range(5):
    data = original_data * i
    compressed = bz2.compress(data)
    print(fmt.format(len(data), len(compressed)), end='')
    print('*' if len(data) < len(compressed) else '')
```

The output lines ending with * show the points where the compressed data is longer than the raw input.

```
$ python3 bz2_lengths.py

      len(data)  len(compressed)
---------------  ---------------
              0               14*
             26               62*
             52               68*
             78               70
            104               72
```

Incremental Compression and Decompression

The in-memory approach has obvious drawbacks that make it impractical for real-world use cases. The alternative is to use `BZ2Compressor` and `BZ2Decompressor` objects to manipulate data incrementally so that the entire data set does not have to fit into memory.

```
# bz2_incremental.py

import bz2
import binascii
import io

compressor = bz2.BZ2Compressor()

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 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 bz2_incremental.py

buffering...
buffering...
buffering...
buffering...
Flushed: b'425a6839314159265359ba83a48c000014d5800010400504052fa
7fe003000ba9112793d4ca789068698a0d1a341901a0d53f4d1119a8d4c9e812
d755a67c10798387682c7ca7b5a3bb75da77755eb81c1cb1ca94c4b6faf209c5
2a90aaa4d16a4a1b9c167a01c8d9ef32589d831e77df7a5753a398b11660e392
126fc18a72a1088716cc8dedda5d489da410748531278043d70a8a131c2b8adc
d6a221bdb8c7ff76b88c1d5342ee48a70a12175074918'
```

Mixed Content Streams

BZ2Decompressor can also be used in situations where compressed and uncompressed data is mixed together.

```
# bz2_mixed.py

import bz2

lorem = open('lorem.txt', 'rt').read().encode('utf-8')
compressed = bz2.compress(lorem)
combined = compressed + lorem

decompressor = bz2.BZ2Decompressor()
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 bz2_mixed.py

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

Writing Compressed Files

BZ2File can be used to write to and read from bzip2-compressed files using the usual methods for writing and reading data.

```
# bz2_file_write.py

import bz2
import io
import os

data = 'Contents of the example file go here.\n'

with bz2.BZ2File('example.bz2', 'wb') as output:
    with io.TextIOWrapper(output, encoding='utf-8') as enc:
        enc.write(data)

os.system('file example.bz2')
```

To write data into a compressed file, open the file with mode 'wb'. This example wraps the BZ2File with a TextIOWrapper from the io module to encode Unicode text to bytes suitable for compression.

```
$ python3 bz2_file_write.py

example.bz2: bzip2 compressed data, block size = 900k
```

Different compression levels can be used by passing a compresslevel argument. Valid values range from 1 to 9, inclusive. Lower values are faster and result in less compression. Higher values are slower and compress more, up to a point.

```
# bz2_file_compresslevel.py

import bz2
import io
import os

data = open('lorem.txt', 'r', encoding='utf-8').read() * 1024
print('Input contains {} bytes'.format(
    len(data.encode('utf-8'))))
```

```
for i in range(1, 10):
    filename = 'compress-level-{}.bz2'.format(i)
    with bz2.BZ2File(filename, 'wb', compresslevel=i) as output:
        with io.TextIOWrapper(output, encoding='utf-8') as enc:
            enc.write(data)
    os.system('cksum {}'.format(filename))
```

The center column of numbers in the output of the script is the size in bytes of the files produced. For this input data, the higher compression values do not always pay off in decreased storage space for the same input data. Results will vary for other inputs.

```
$ python3 bz2_file_compresslevel.py

3018243926 8771 compress-level-1.bz2
1942389165 4949 compress-level-2.bz2
2596054176 3708 compress-level-3.bz2
1491394456 2705 compress-level-4.bz2
1425874420 2705 compress-level-5.bz2
2232840816 2574 compress-level-6.bz2
447681641 2394 compress-level-7.bz2
3699654768 1137 compress-level-8.bz2
3103658384 1137 compress-level-9.bz2
Input contains 754688 bytes
```

A `BZ2File` instance also includes a `writelines()` method that can be used to write a sequence of strings.

```
# bz2_file_writelines.py

import bz2
import io
import itertools
import os

data = 'The same line, over and over.\n'

with bz2.BZ2File('lines.bz2', 'wb') as output:
    with io.TextIOWrapper(output, encoding='utf-8') as enc:
        enc.writelines(itertools.repeat(data, 10))

os.system('bzcat lines.bz2')
```

The lines should end in a newline character, as when writing to a regular file.

```
$ python3 bz2_file_writelines.py

The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
```

Reading Compressed Files

To read data back from previously compressed files, open the file with read mode (`'rb'`). The value returned from `read()` will be a byte string.

```
# bz2_file_read.py

import bz2
import io

with bz2.BZ2File('example.bz2', 'rb') as input:
    with io.TextIOWrapper(input, encoding='utf-8') as dec:
```

```
    with io.TextIOWrapper(input, encoding='utf-8') as dec:
        print(dec.read())
```

This example reads the file written by `bz2_file_write.py` from the previous section. The `BZ2File` is wrapped with a `TextIOWrapper` to decode bytes read to Unicode text.

```
$ python3 bz2_file_read.py

Contents of the example file go here.
```

While reading a file, it is also possible to seek, and to read only part of the data.

```
# bz2_file_seek.py

import bz2
import contextlib

with bz2.BZ2File('example.bz2', 'rb') as input:
    print('Entire file:')
    all_data = input.read()
    print(all_data)

    expected = all_data[5:15]

    # rewind to beginning
    input.seek(0)

    # move ahead 5 bytes
    input.seek(5)
    print('Starting at position 5 for 10 bytes:')
    partial = input.read(10)
    print(partial)

    print()
    print(expected == partial)
```

The `seek()` position is relative to the *uncompressed* data, so the caller does not need to be aware that the data file is compressed. This allows a `BZ2File` instance to be passed to a function expecting a regular uncompressed file.

```
$ python3 bz2_file_seek.py

Entire file:
b'Contents of the example file go here.\n'
Starting at position 5 for 10 bytes:
b'nts of the'

True
```

Reading and Writing Unicode Data

The previous examples used `BZ2File` directly and managed the encoding and decoding of Unicode text strings inline with an `io.TextIOWrapper`, where necessary. These extra steps can be avoided by using `bz2.open()`, which sets up an `io.TextIOWrapper` to handle the encoding or decoding automatically.

```
# bz2_unicode.py

import bz2
import os

data = 'Character with an åccent.'

with bz2.open('example.bz2', 'wt', encoding='utf-8') as output:
    output.write(data)

with bz2.open('example.bz2', 'rt', encoding='utf-8') as input:
    print('Full file: {}'.format(input.read()))

# Move to the beginning of the accented character.
with bz2.open('example.bz2', 'rt', encoding='utf-8') as input:
```

```
with bz2.open('example.bz2', 'rt', encoding='utf-8') as input:
    input.seek(18)
    print('One character: {}'.format(input.read(1)))

# Move to the middle of the accented character.
with bz2.open('example.bz2', 'rt', encoding='utf-8') as input:
    input.seek(19)
    try:
        print(input.read(1))
    except UnicodeDecodeError:
        print('ERROR: failed to decode')
```

The file handle returned by `open()` supports `seek()`, but use care because the file pointer moves by *bytes* not *characters* and may end up in the middle of an encoded character.

```
$ python3 bz2_unicode.py

Full file: Character with an åccent.
One character: å
ERROR: failed to decode
```

Compressing Network Data

The code in the next example responds to requests consisting of filenames by writing a compressed version of the file to the socket used to communicate with the client. It has some artificial chunking in place to illustrate the buffering that occurs when the data passed to `compress()` or `decompress()` does not result in a complete block of compressed or uncompressed output.

```
# bz2_server.py

import bz2
import logging
import socketserver
import binascii

BLOCK_SIZE = 32


class Bz2RequestHandler(socketserver.BaseRequestHandler):

    logger = logging.getLogger('Server')

    def handle(self):
        compressor = bz2.BZ2Compressor()

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

The main program starts a server in a thread, combining `SocketServer` and `Bz2RequestHandler`.

```
if __name__ == '__main__':
    import socket
    import sys
    from io import StringIO
    import threading

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

    # Set up a server, running in a separate thread
    address = ('localhost', 0)  # let the kernel assign a port
    server = socketserver.TCPServer(address, Bz2RequestHandler)
    ip, port = server.server_address  # what port was assigned?

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

    logger = logging.getLogger('Client')

    # Connect to the server
    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 = (sys.argv[0]
                      if len(sys.argv) > 1
                      else 'lorem.txt')
    logger.debug('sending filename: "%s"', requested_file)
    len_sent = s.send(requested_file.encode('utf-8'))

    # Receive a response
    buffer = StringIO()
    decompressor = bz2.BZ2Decompressor()
    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.
        decompressed = decompressor.decompress(response)
        if decompressed:
            logger.debug('DECOMPRESSED %r', decompressed)
            buffer.write(decompressed.decode('utf-8'))
        else:
            logger.debug('BUFFERING')

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

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

It then opens a socket to communicate with the server as a client, and requests the file (defaulting to `lorem.txt`) which contains:

```
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Donec
```

```
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Donec
egestas, enim et consectetuer ullamcorper, lectus ligula rutrum leo,
a elementum elit tortor eu quam. Duis tincidunt nisi ut ante. Nulla
facilisi.
```

Warning

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

Running `bz2_server.py` produces:

```
$ python3 bz2_server.py

Client: Contacting server on 127.0.0.1:57364
Client: sending filename: "lorem.txt"
Server: client asked for: "lorem.txt"
Server: RAW b'Lorem ipsum dolor sit amet, cons'
Server: BUFFERING
Server: RAW b'ectetuer adipiscing elit. Donec\n'
Server: BUFFERING
Server: RAW b'egestas, enim et consectetuer ul'
Server: BUFFERING
Server: RAW b'lamcorper, lectus ligula rutrum '
Server: BUFFERING
Server: RAW b'leo,\na elementum elit tortor eu '
Server: BUFFERING
Server: RAW b'quam. Duis tincidunt nisi ut ant'
Server: BUFFERING
Server: RAW b'e. Nulla\nfacilisi.\n'
Server: BUFFERING
Server: FLUSHING b'425a68393l4159265359ba83a48c000014d5800010400
504052fa7fe003000ba'
Server: FLUSHING b'9112793d4ca789068698a0d1a341901a0d53f4d1119a8
d4c9e812d755a67c107'
Client: READ b'425a68393l4159265359ba83a48c000014d58000104005040
52fa7fe003000ba'
Server: FLUSHING b'98387682c7ca7b5a3bb75da77755eb81c1cb1ca94c4b6
faf209c52a90aaa4d16'
Client: BUFFERING
Server: FLUSHING b'a4a1b9c167a01c8d9ef32589d831e77df7a5753a398b1
1660e392126fc18a72a'
Client: READ b'9112793d4ca789068698a0d1a341901a0d53f4d1119a8d4c9
e812d755a67c107'
Server: FLUSHING b'1088716cc8dedda5d489da410748531278043d70a8a13
1c2b8adcd6a221bdb8c'
Client: BUFFERING
Server: FLUSHING b'7ff76b88c1d5342ee48a70a12175074918'
Client: READ b'98387682c7ca7b5a3bb75da77755eb81c1cb1ca94c4b6faf2
09c52a90aaa4d16'
Client: BUFFERING
Client: READ b'a4a1b9c167a01c8d9ef32589d831e77df7a5753a398b11660
e392126fc18a72a'
Client: BUFFERING
Client: READ b'1088716cc8dedda5d489da410748531278043d70a8a131c2b
8adcd6a221bdb8c'
Client: BUFFERING
Client: READ b'7ff76b88c1d5342ee48a70a12175074918'
Client: DECOMPRESSED b'Lorem ipsum dolor sit amet, consectetuer
adipiscing elit. Donec\negestas, enim et consectetuer ullamcorpe
r, lectus ligula rutrum leo,\na elementum elit tortor eu quam. D
uis tincidunt nisi ut ante. Nulla\nfacilisi.\n'
Client: response matches file contents: True
```

See also

- Standard library documentation for bz2
- bzip2.org – The home page for bzip2.
- `zlib` – The `zlib` module for GNU zip compression.
- `gzip` – A file-like interface to GNU zip compressed files.

- gzip – A file-like interface to GNU zip compressed files.
- io – Building-blocks for creating input and output pipelines.
- Python 2 to 3 porting notes for bz2

gzip — Read and Write GNU zip Files

tarfile — Tar Archive Access

Quick Links

This page was last updated 2016-12-29.

Navigation

gzip — Read and Write GNU zip Files
tarfile — Tar Archive Access

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?