codecs — String Encoding and Decoding

Purpose: Encoders and decoders for converting text between different representations.

The `codecs` module provides stream and file interfaces for transcoding data. It is most commonly used to work with Unicode text, but other encodings are also available for other purposes.

Unicode Primer

CPython 3.x differentiates between *text* and *byte* strings. `bytes` instances use a sequence of 8-bit byte values. In contrast, `str` strings are managed internally as a sequence of Unicode *code points*. The code point values are saved as a sequence of 2 or 4 bytes each, depending on the options given when Python was compiled.

When `str` values are output, they are encoded using one of several standard schemes so that the sequence of bytes can be reconstructed as the same string of text later. The bytes of the encoded value are not necessarily the same as the code point values, and the encoding defines a way to translate between the two sets of values. Reading Unicode data also requires knowing the encoding so that the incoming bytes can be converted to the internal representation used by the `unicode` class.

The most common encodings for Western languages are `UTF-8` and `UTF-16`, which use sequences of one and two byte values respectively to represent each code point. Other encodings can be more efficient for storing languages where most of the characters are represented by code points that do not fit into two bytes.

See also

For more introductory information about Unicode, refer to the list of references at the end of this section. The *Python Unicode HOWTO* is especially helpful.

Encodings

The best way to understand encodings is to look at the different series of bytes produced by encoding the same string in different ways. The following examples use this function to format the byte string to make it easier to read.

```
# codecs_to_hex.py

import binascii


def to_hex(t, nbytes):
    """Format text t as a sequence of nbyte long values
    separated by spaces.
    """
    chars_per_item = nbytes * 2
    hex_version = binascii.hexlify(t)
    return b' '.join(
        hex_version[start:start + chars_per_item]
        for start in range(0, len(hex_version), chars_per_item)
    )


if __name__ == '__main__':
    print(to_hex(b'abcdef', 1))
    print(to_hex(b'abcdef', 2))
```

The function uses `binascii` to get a hexadecimal representation of the input byte string, then insert a space between every nbytes bytes before returning the value.

```
$ python3 codecs_to_hex.py

b'61 62 63 64 65 66'
b'6162 6364 6566'
```

The first encoding example begins by printing the text `'français'` using the raw representation of the `unicode` class followed by the name of each character from the Unicode database. The next two lines encode the string as UTF-8 and UTF-16 respectively, and show the hexadecimal values resulting from the encoding.

```
# codecs_encodings.py

import unicodedata
from codecs_to_hex import to_hex

text = 'français'

print('Raw   : {!r}'.format(text))
for c in text:
    print('  {!r}: {}'.format(c, unicodedata.name(c, c)))
print('UTF-8 : {!r}'.format(to_hex(text.encode('utf-8'), 1)))
print('UTF-16: {!r}'.format(to_hex(text.encode('utf-16'), 2)))
```

The result of encoding a `str` is a bytes object.

```
$ python3 codecs_encodings.py

Raw   : 'français'
  'f': LATIN SMALL LETTER F
  'r': LATIN SMALL LETTER R
  'a': LATIN SMALL LETTER A
  'n': LATIN SMALL LETTER N
  'ç': LATIN SMALL LETTER C WITH CEDILLA
  'a': LATIN SMALL LETTER A
  'i': LATIN SMALL LETTER I
  's': LATIN SMALL LETTER S
UTF-8 : b'66 72 61 6e c3 a7 61 69 73'
UTF-16: b'fffe 6600 7200 6100 6e00 e700 6100 6900 7300'
```

Given a sequence of encoded bytes as a `bytes` instance, the `decode()` method translates them to code points and returns the sequence as a `str` instance.

```
# codecs_decode.py

from codecs_to_hex import to_hex

text = 'français'
encoded = text.encode('utf-8')
decoded = encoded.decode('utf-8')

print('Original :', repr(text))
print('Encoded  :', to_hex(encoded, 1), type(encoded))
print('Decoded  :', repr(decoded), type(decoded))
```

The choice of encoding used does not change the output type.

```
$ python3 codecs_decode.py

Original : 'français'
Encoded  : b'66 72 61 6e c3 a7 61 69 73' <class 'bytes'>
Decoded  : 'français' <class 'str'>
```

Note

The default encoding is set during the interpreter start-up process, when site is loaded. Refer to the Unicode Defaults section from the discussion of sys for a description of the default encoding settings.

Working with Files

Encoding and decoding strings is especially important when dealing with I/O operations. Whether writing to a file, socket, or other stream, the data must use the proper encoding. In general, all text data needs to be decoded from its byte representation as it is read, and encoded from the internal values to a specific representation as it is written. A program can explicitly encode and decode data, but depending on the encoding used it can be non-trivial to determine whether enough bytes have been read in order to fully decode the data. codecs provides classes that manage the data encoding and decoding, so applications do not have to do that work.

The simplest interface provided by codecs is an alternative to the built-in open() function. The new version works just like the built-in, but adds two new arguments to specify the encoding and desired error handling technique.

```
# codecs_open_write.py

from codecs_to_hex import to_hex

import codecs
import sys

encoding = sys.argv[1]
filename = encoding + '.txt'

print('Writing to', filename)
with codecs.open(filename, mode='w', encoding=encoding) as f:
    f.write('français')

# Determine the byte grouping to use for to_hex()
nbytes = {
    'utf-8': 1,
    'utf-16': 2,
    'utf-32': 4,
}.get(encoding, 1)

# Show the raw bytes in the file
print('File contents:')
with open(filename, mode='rb') as f:
    print(to_hex(f.read(), nbytes))
```

This example starts with a unicode string with “ç” and saves the text to a file using an encoding specified on the command line.

```
$ python3 codecs_open_write.py utf-8

Writing to utf-8.txt
File contents:
b'66 72 61 6e c3 a7 61 69 73'

$ python3 codecs_open_write.py utf-16

Writing to utf-16.txt
File contents:
b'fffe 6600 7200 6100 6e00 e700 6100 6900 7300'

$ python3 codecs_open_write.py utf-32

Writing to utf-32.txt
File contents:
b'fffe0000 66000000 72000000 61000000 6e000000 e7000000 61000000
69000000 73000000'
```

Reading the data with open() is straightforward, with one catch: the encoding must be known in advance, in order to set up the decoder correctly. Some data formats, such as XML, specify the encoding as part of the file, but usually it is up to the application to manage. codecs simply takes the encoding as an argument and assumes it is correct.

```
# codecs_open_read.py

import codecs
import sys

encoding = sys.argv[1]
filename = encoding + '.txt'

print('Reading from', filename)
with codecs.open(filename, mode='r', encoding=encoding) as f:
    print(repr(f.read()))
```

This example reads the files created by the previous program, and prints the representation of the resulting unicode object to the console.

```
$ python3 codecs_open_read.py utf-8

Reading from utf-8.txt
'français'

$ python3 codecs_open_read.py utf-16

Reading from utf-16.txt
'français'

$ python3 codecs_open_read.py utf-32

Reading from utf-32.txt
'français'
```

Byte Order

Multi-byte encodings such as UTF-16 and UTF-32 pose a problem when transferring the data between different computer systems, either by copying the file directly or with network communication. Different systems use different ordering of the high and low order bytes. This characteristic of the data, known as its *endianness*, depends on factors such as the hardware architecture and choices made by the operating system and application developer. There is not always a way to know in advance what byte order to use for a given set of data, so the multi-byte encodings include a *byte-order marker* (BOM) as the first few bytes of encoded output. For example, UTF-16 is defined in such a way that 0xFFFE and 0xFEFF are not valid characters, and can be used to indicate the byte order. codecs defines constants for the byte order markers used by UTF-16 and UTF-32.

```
# codecs_bom.py

import codecs
from codecs_to_hex import to_hex

BOM_TYPES = [
    'BOM', 'BOM_BE', 'BOM_LE',
    'BOM_UTF8',
    'BOM_UTF16', 'BOM_UTF16_BE', 'BOM_UTF16_LE',
    'BOM_UTF32', 'BOM_UTF32_BE', 'BOM_UTF32_LE',
]

for name in BOM_TYPES:
    print('{:12} : {}'.format(
        name, to_hex(getattr(codecs, name), 2)))
```

BOM, BOM_UTF16, and BOM_UTF32 are automatically set to the appropriate big-endian or little-endian values depending on the current system’s native byte order.

```
$ python3 codecs_bom.py

BOM          : b'fffe'
BOM_BE       : b'feff'
BOM_LE       : b'fffe'
BOM_UTF8     : b'efbb bf'
BOM_UTF16    : b'fffe'
BOM_UTF16_BE : b'feff'
BOM_UTF16_LE : b'fffe'
BOM_UTF32    : b'fffe 0000'
BOM_UTF32_BE : b'0000 feff'
BOM_UTF32_LE : b'fffe 0000'
```

Byte ordering is detected and handled automatically by the decoders in codecs, but an explicit ordering can be specified when encoding.

```
# codecs_bom_create_file.py

import codecs
from codecs_to_hex import to_hex
```

```
# Pick the nonnative version of UTF-16 encoding
if codecs.BOM_UTF16 == codecs.BOM_UTF16_BE:
    bom = codecs.BOM_UTF16_LE
    encoding = 'utf_16_le'
else:
    bom = codecs.BOM_UTF16_BE
    encoding = 'utf_16_be'

print('Native order  :', to_hex(codecs.BOM_UTF16, 2))
print('Selected order:', to_hex(bom, 2))

# Encode the text.
encoded_text = 'français'.encode(encoding)
print('{:14}: {}'.format(encoding, to_hex(encoded_text, 2)))

with open('nonnative-encoded.txt', mode='wb') as f:
    # Write the selected byte-order marker.  It is not included
    # in the encoded text because the byte order was given
    # explicitly when selecting the encoding.
    f.write(bom)
    # Write the byte string for the encoded text.
    f.write(encoded_text)
```

codecs_bom_create_file.py figures out the native byte ordering, then uses the alternate form explicitly so the next example can demonstrate auto-detection while reading.

```
$ python3 codecs_bom_create_file.py

Native order  : b'fffe'
Selected order: b'feff'
utf_16_be     : b'0066 0072 0061 006e 00e7 0061 0069 0073'
```

codecs_bom_detection.py does not specify a byte order when opening the file, so the decoder uses the BOM value in the first two bytes of the file to determine it.

```
# codecs_bom_detection.py

import codecs
from codecs_to_hex import to_hex

# Look at the raw data
with open('nonnative-encoded.txt', mode='rb') as f:
    raw_bytes = f.read()

print('Raw    :', to_hex(raw_bytes, 2))

# Re-open the file and let codecs detect the BOM
with codecs.open('nonnative-encoded.txt',
                 mode='r',
                 encoding='utf-16',
                 ) as f:
    decoded_text = f.read()

print('Decoded:', repr(decoded_text))
```

Since the first two bytes of the file are used for byte order detection, they are not included in the data returned by read().

```
$ python3 codecs_bom_detection.py

Raw    : b'feff 0066 0072 0061 006e 00e7 0061 0069 0073'
Decoded: 'français'
```

Error Handling

The previous sections pointed out the need to know the encoding being used when reading and writing Unicode files. Setting the encoding correctly is important for two reasons. If the encoding is configured incorrectly while reading from a file, the data will be interpreted wrong and may be corrupted or simply fail to decode. Not all Unicode characters can be represented in all encodings, so if the wrong encoding is used while writing then an error will be generated and data may be lost.

codecs uses the same five error handling options that are provided by the encode() method of str and the decode() method of bytes, listed in the table below.

Codec Error Handling Modes

Error Mode	Description
strict	Raises an exception if the data cannot be converted.
replace	Substitutes a special marker character for data that cannot be encoded.
ignore	Skips the data.
xmlcharrefreplace	XML character (encoding only)
backslashreplace	escape sequence (encoding only)

Encoding Errors

The most common error condition is receiving a UnicodeEncodeError when writing Unicode data to an ASCII output stream, such as a regular file or sys.stdout without a more robust encoding set. This sample program can be used to experiment with the different error handling modes.

```
# codecs_encode_error.py

import codecs
import sys

error_handling = sys.argv[1]

text = 'français'

try:
    # Save the data, encoded as ASCII, using the error
    # handling mode specified on the command line.
    with codecs.open('encode_error.txt', 'w',
                     encoding='ascii',
                     errors=error_handling) as f:
        f.write(text)

except UnicodeEncodeError as err:
    print('ERROR:', err)

else:
    # If there was no error writing to the file,
    # show what it contains.
    with open('encode_error.txt', 'rb') as f:
        print('File contents: {!r}'.format(f.read()))
```

While strict mode is safest for ensuring an application explicitly sets the correct encoding for all I/O operations, it can lead to program crashes when an exception is raised.

```
$ python3 codecs_encode_error.py strict

ERROR: 'ascii' codec can't encode character '\xe7' in position
4: ordinal not in range(128)
```

Some of the other error modes are more flexible. For example, replace ensures that no error is raised, at the expense of possibly losing data that cannot be converted to the requested encoding. The Unicode character for pi still cannot be encoded in ASCII, but instead of raising an exception the character is replaced with ? in the output.

```
$ python3 codecs_encode_error.py replace

File contents: b'fran?ais'
```

To skip over problem data entirely, use ignore. Any data that cannot be encoded is discarded.

```
$ python3 codecs_encode_error.py ignore

File contents: b'franais'
```

There are two lossless error handling options, both of which replace the character with an alternate representation defined by a standard separate from the encoding. `xmlcharrefreplace` uses an XML character reference as a substitute (the list of character references is specified in the W3C document *XML Entity Definitions for Characters*).

```
$ python3 codecs_encode_error.py xmlcharrefreplace

File contents: b'fran&#231;ais'
```

The other lossless error handling scheme is `backslashreplace`, which produces an output format like the value returned when `repr()` of a `unicode` object is printed. Unicode characters are replaced with `\u` followed by the hexadecimal value of the code point.

```
$ python3 codecs_encode_error.py backslashreplace

File contents: b'fran\\xe7ais'
```

Decoding Errors

It is also possible to see errors when decoding data, especially if the wrong encoding is used.

```
# codecs_decode_error.py

import codecs
import sys

from codecs_to_hex import to_hex

error_handling = sys.argv[1]

text = 'français'
print('Original     :', repr(text))

# Save the data with one encoding
with codecs.open('decode_error.txt', 'w',
                 encoding='utf-16') as f:
    f.write(text)

# Dump the bytes from the file
with open('decode_error.txt', 'rb') as f:
    print('File contents:', to_hex(f.read(), 1))

# Try to read the data with the wrong encoding
with codecs.open('decode_error.txt', 'r',
                 encoding='utf-8',
                 errors=error_handling) as f:
    try:
        data = f.read()
    except UnicodeDecodeError as err:
        print('ERROR:', err)
    else:
        print('Read         :', repr(data))
```

As with encoding, `strict` error handling mode raises an exception if the byte stream cannot be properly decoded. In this case, a `UnicodeDecodeError` results from trying to convert part of the UTF-16 BOM to a character using the UTF-8 decoder.

```
$ python3 codecs_decode_error.py strict

Original     : 'français'
File contents: b'ff fe 66 00 72 00 61 00 6e 00 e7 00 61 00 69 00
73 00'
ERROR: 'utf-8' codec can't decode byte 0xff in position 0:
invalid start byte
```

Switching to `ignore` causes the decoder to skip over the invalid bytes. The result is still not quite what is expected, though, since it includes embedded null bytes.

```
$ python3 codecs_decode_error.py ignore
```

```
Original     : 'français'
File contents: b'ff fe 66 00 72 00 61 00 6e 00 e7 00 61 00 69 00
73 00'
Read         : 'f\x00r\x00a\x00n\x00\x00a\x00i\x00s\x00'
```

In `replace` mode invalid bytes are replaced with `\uFFFD`, the official Unicode replacement character, which looks like a diamond with a black background containing a white question mark.

```
$ python3 codecs_decode_error.py replace

Original     : 'français'
File contents: b'ff fe 66 00 72 00 61 00 6e 00 e7 00 61 00 69 00
73 00'
Read         : '��f\x00r\x00a\x00n\x00�\x00a\x00i\x00s\x00'
```

Encoding Translation

Although most applications will work with `str` data internally, decoding or encoding it as part of an I/O operation, there are times when changing a file's encoding without holding on to that intermediate data format is useful. `EncodedFile()` takes an open file handle using one encoding and wraps it with a class that translates the data to another encoding as the I/O occurs.

```python
# codecs_encodedfile.py

from codecs_to_hex import to_hex

import codecs
import io

# Raw version of the original data.
data = 'français'

# Manually encode it as UTF-8.
utf8 = data.encode('utf-8')
print('Start as UTF-8   :', to_hex(utf8, 1))

# Set up an output buffer, then wrap it as an EncodedFile.
output = io.BytesIO()
encoded_file = codecs.EncodedFile(output, data_encoding='utf-8',
                                  file_encoding='utf-16')
encoded_file.write(utf8)

# Fetch the buffer contents as a UTF-16 encoded byte string
utf16 = output.getvalue()
print('Encoded to UTF-16:', to_hex(utf16, 2))

# Set up another buffer with the UTF-16 data for reading,
# and wrap it with another EncodedFile.
buffer = io.BytesIO(utf16)
encoded_file = codecs.EncodedFile(buffer, data_encoding='utf-8',
                                  file_encoding='utf-16')

# Read the UTF-8 encoded version of the data.
recoded = encoded_file.read()
print('Back to UTF-8    :', to_hex(recoded, 1))
```

This example shows reading from and writing to separate handles returned by `EncodedFile()`. No matter whether the handle is used for reading or writing, the `file_encoding` always refers to the encoding in use by the open file handle passed as the first argument, and `data_encoding` value refers to the encoding in use by the data passing through the `read()` and `write()` calls.

```
$ python3 codecs_encodedfile.py

Start as UTF-8    : b'66 72 61 6e c3 a7 61 69 73'
Encoded to UTF-16: b'fffe 6600 7200 6100 6e00 e700 6100 6900
7300'
Back to UTF-8    : b'66 72 61 6e c3 a7 61 69 73'
```

Non-Unicode Encodings

Although most of the earlier examples use Unicode encodings, codecs can be used for many other data translations. For example, Python includes codecs for working with base-64, bzip2, ROT-13, ZIP, and other data formats.

```
# codecs_rot13.py

import codecs
import io

buffer = io.StringIO()
stream = codecs.getwriter('rot_13')(buffer)

text = 'abcdefghijklmnopqrstuvwxyz'

stream.write(text)
stream.flush()

print('Original:', text)
print('ROT-13  :', buffer.getvalue())
```

Any transformation that can be expressed as a function taking a single input argument and returning a byte or Unicode string can be registered as a codec. For the `'rot_13'` codec, the input should be a Unicode string and the output will also be a Unicode string.

```
$ python3 codecs_rot13.py

Original: abcdefghijklmnopqrstuvwxyz
ROT-13  : nopqrstuvwxyzabcdefghijklm
```

Using codecs to wrap a data stream provides a simpler interface than working directly with `zlib`.

```
# codecs_zlib.py

import codecs
import io

from codecs_to_hex import to_hex

buffer = io.BytesIO()
stream = codecs.getwriter('zlib')(buffer)

text = b'abcdefghijklmnopqrstuvwxyz\n' * 50

stream.write(text)
stream.flush()

print('Original length :', len(text))
compressed_data = buffer.getvalue()
print('ZIP compressed  :', len(compressed_data))

buffer = io.BytesIO(compressed_data)
stream = codecs.getreader('zlib')(buffer)

first_line = stream.readline()
print('Read first line :', repr(first_line))

uncompressed_data = first_line + stream.read()
print('Uncompressed    :', len(uncompressed_data))
print('Same            :', text == uncompressed_data)
```

Not all of the compression or encoding systems support reading a portion of the data through the stream interface using `readline()` or `read()` because they need to find the end of a compressed segment to expand it. If a program cannot hold the entire uncompressed data set in memory, use the incremental access features of the compression library, instead of codecs.

```
$ python3 codecs_zlib.py

Original length : 1350
ZIP compressed  : 48
Read first line : b'abcdefghijklmnopqrstuvwxyz\n'
```

```
Uncompressed    : 1350
Same            : True
```

Incremental Encoding

Some of the encodings provided, especially `bz2` and `zlib`, may dramatically change the length of the data stream as they work on it. For large data sets, these encodings operate better incrementally, working on one small chunk of data at a time. The `IncrementalEncoder` and `IncrementalDecoder` API is designed for this purpose.

```python
# codecs_incremental_bz2.py

import codecs
import sys

from codecs_to_hex import to_hex

text = b'abcdefghijklmnopqrstuvwxyz\n'
repetitions = 50

print('Text length :', len(text))
print('Repetitions :', repetitions)
print('Expected len:', len(text) * repetitions)

# Encode the text several times to build up a
# large amount of data
encoder = codecs.getincrementalencoder('bz2')()
encoded = []

print()
print('Encoding:', end=' ')
last = repetitions - 1
for i in range(repetitions):
    en_c = encoder.encode(text, final=(i == last))
    if en_c:
        print('\nEncoded : {} bytes'.format(len(en_c)))
        encoded.append(en_c)
    else:
        sys.stdout.write('.')

all_encoded = b''.join(encoded)
print()
print('Total encoded length:', len(all_encoded))
print()

# Decode the byte string one byte at a time
decoder = codecs.getincrementaldecoder('bz2')()
decoded = []

print('Decoding:', end=' ')
for i, b in enumerate(all_encoded):
    final = (i + 1) == len(text)
    c = decoder.decode(bytes([b]), final)
    if c:
        print('\nDecoded : {} characters'.format(len(c)))
        print('Decoding:', end=' ')
        decoded.append(c)
    else:
        sys.stdout.write('.')
print()

restored = b''.join(decoded)

print()
print('Total uncompressed length:', len(restored))
```

Each time data is passed to the encoder or decoder its internal state is updated. When the state is consistent (as defined by the codec), data is returned and the state resets. Until that point, calls to `encode()` or `decode()` will not return any data. When the last bit of data is passed in, the argument `final` should be set to `True` so the codec knows to flush any remaining buffered data.

```
$ python3 codecs_incremental_bz2.py

Text length : 27
Repetitions : 50
Expected len: 1350

Encoding: .................................................
Encoded : 99 bytes

Total encoded length: 99

Decoding: .....................................................
.................................
Decoded : 1350 characters
Decoding: ..........

Total uncompressed length: 1350
```

Unicode Data and Network Communication

Network sockets are byte-streams, and unlike the standard input and output streams they do not support encoding by default. That means programs that want to send or receive Unicode data over the network must encode into bytes before it is written to a socket. This server echos data it receives back to the sender.

```python
# codecs_socket_fail.py

import sys
import socketserver


class Echo(socketserver.BaseRequestHandler):

    def handle(self):
        # Get some bytes and echo them back to the client.
        data = self.request.recv(1024)
        self.request.send(data)
        return


if __name__ == '__main__':
    import codecs
    import socket
    import threading

    address = ('localhost', 0)  # let the kernel assign a port
    server = socketserver.TCPServer(address, Echo)
    ip, port = server.server_address  # what port was assigned?

    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True)  # don't hang on exit
    t.start()

    # Connect to the server
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))

    # Send the data
    # WRONG: Not encoded first!
    text = 'français'
    len_sent = s.send(text)

    # Receive a response
    response = s.recv(len_sent)
    print(repr(response))

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

The data could be encoded explicitly before each call to `send()`, but missing one call to `send()` would result in an encoding error.

```
$ python3 codecs_socket_fail.py

Traceback (most recent call last):
  File "codecs_socket_fail.py", line 43, in <module>
    len_sent = s.send(text)
TypeError: a bytes-like object is required, not 'str'
```

Using `makefile()` to get a file-like handle for the socket, and then wrapping that with a stream-based reader or writer, means Unicode strings will be encoded on the way in to and out of the socket.

```
# codecs_socket.py

import sys
import socketserver


class Echo(socketserver.BaseRequestHandler):

    def handle(self):
        """Get some bytes and echo them back to the client.

        There is no need to decode them, since they are not used.

        """
        data = self.request.recv(1024)
        self.request.send(data)


class PassThrough:

    def __init__(self, other):
        self.other = other

    def write(self, data):
        print('Writing :', repr(data))
        return self.other.write(data)

    def read(self, size=-1):
        print('Reading :', end=' ')
        data = self.other.read(size)
        print(repr(data))
        return data

    def flush(self):
        return self.other.flush()

    def close(self):
        return self.other.close()


if __name__ == '__main__':
    import codecs
    import socket
    import threading

    address = ('localhost', 0)  # let the kernel assign a port
    server = socketserver.TCPServer(address, Echo)
    ip, port = server.server_address  # what port was assigned?

    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True)  # don't hang on exit
    t.start()

    # Connect to the server
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))

    # Wrap the socket with a reader and writer.
    # Wrap the socket with a reader and writer.
    read_file = s.makefile('rb')
    incoming = codecs.getreader('utf-8')(PassThrough(read_file))
    write_file = s.makefile('wb')
    outgoing = codecs.getwriter('utf-8')(PassThrough(write_file))

    # Send the data
    text = 'français'
    print('Sending :', repr(text))
    outgoing.write(text)
    outgoing.flush()

    # Receive a response
    response = incoming.read()
    print('Received:', repr(response))

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

This example uses `PassThrough` to show that the data is encoded before being sent, and the response is decoded after it is received in the client.

```
$ python3 codecs_socket.py

Sending : 'français'
Writing : b'fran\xc3\xa7ais'
Reading : b'fran\xc3\xa7ais'
Reading : b''
Received: 'français'
```

Defining a Custom Encoding

Since Python comes with a large number of standard codecs already, it is unlikely that an application will need to define a custom encoder or decoder. When it is necessary, though, there are several base classes in codecs to make the process easier.

The first step is to understand the nature of the transformation described by the encoding. These examples will use an “invertcaps” encoding which converts uppercase letters to lowercase, and lowercase letters to uppercase. Here is a simple definition of an encoding function that performs this transformation on an input string.

```
# codecs_invertcaps.py

import string


def invertcaps(text):
    """Return new string with the case of all letters switched.
    """
    return ''.join(
        c.upper() if c in string.ascii_lowercase
        else c.lower() if c in string.ascii_uppercase
        else c
        for c in text
    )


if __name__ == '__main__':
    print(invertcaps('ABCdef'))
    print(invertcaps('abcDEF'))
```

In this case, the encoder and decoder are the same function (as is also the case with `ROT-13`).

```
$ python3 codecs_invertcaps.py

abcDEF
ABCdef
```

Although it is easy to understand, this implementation is not efficient, especially for very large text strings. Fortunately, codecs includes some helper functions for creating *character map* based codecs such as invertcaps. A character map codecs includes some helper functions for creating *character map* based codecs such as invertcaps. A character map encoding is made up of two dictionaries. The *encoding map* converts character values from the input string to byte values in the output and the *decoding map* goes the other way. Create the decoding map first, and then use `make_encoding_map()` to convert it to an encoding map. The C functions `charmap_encode()` and `charmap_decode()` use the maps to convert their input data efficiently.

```
# codecs_invertcaps_charmap.py

import codecs
import string

# Map every character to itself
decoding_map = codecs.make_identity_dict(range(256))

# Make a list of pairs of ordinal values for the lower
# and uppercase letters
pairs = list(zip(
    [ord(c) for c in string.ascii_lowercase],
    [ord(c) for c in string.ascii_uppercase],
))

# Modify the mapping to convert upper to lower and
# lower to upper.
decoding_map.update({
    upper: lower
    for (lower, upper)
    in pairs
})
decoding_map.update({
    lower: upper
    for (lower, upper)
    in pairs
})

# Create a separate encoding map.
encoding_map = codecs.make_encoding_map(decoding_map)

if __name__ == '__main__':
    print(codecs.charmap_encode('abcDEF', 'strict',
                                encoding_map))
    print(codecs.charmap_decode(b'abcDEF', 'strict',
                                decoding_map))
    print(encoding_map == decoding_map)
```

Although the encoding and decoding maps for invertcaps are the same, that may not always be the case. `make_encoding_map()` detects situations where more than one input character is encoded to the same output byte and replaces the encoding value with None to mark the encoding as undefined.

```
$ python3 codecs_invertcaps_charmap.py

(b'ABCdef', 6)
('ABCdef', 6)
True
```

The character map encoder and decoder support all of the standard error handling methods described earlier, so no extra work is needed to comply with that part of the API.

```
# codecs_invertcaps_error.py

import codecs
from codecs_invertcaps_charmap import encoding_map

text = 'pi: \u03c0'

for error in ['ignore', 'replace', 'strict']:
    try:
        encoded = codecs.charmap_encode(
            text, error, encoding_map)
    except UnicodeEncodeError as err:
        encoded = str(err)
    print('{:7}: {}'.format(error, encoded))
```

Because the Unicode code point for π is not in the encoding map, the strict error handling mode raises an exception.

```
$ python3 codecs_invertcaps_error.py

ignore : (b'PI: ', 5)
replace: (b'PI: ?', 5)
strict : 'charmap' codec can't encode character '\u03c0' in
position 4: character maps to <undefined>
```

After the encoding and decoding maps are defined, a few additional classes need to be set up, and the encoding should be registered. `register()` adds a search function to the registry so that when a user wants to use the encoding `codecs` can locate it. The search function must take a single string argument with the name of the encoding, and return a `CodecInfo` object if it knows the encoding, or None if it does not.

```
# codecs_register.py

import codecs
import encodings


def search1(encoding):
    print('search1: Searching for:', encoding)
    return None


def search2(encoding):
    print('search2: Searching for:', encoding)
    return None


codecs.register(search1)
codecs.register(search2)

utf8 = codecs.lookup('utf-8')
print('UTF-8:', utf8)

try:
    unknown = codecs.lookup('no-such-encoding')
except LookupError as err:
    print('ERROR:', err)
```

Multiple search functions can be registered, and each will be called in turn until one returns a `CodecInfo` or the list is exhausted. The internal search function registered by `codecs` knows how to load the standard codecs such as UTF-8 from `encodings`, so those names will never be passed to custom search functions.

```
$ python3 codecs_register.py

UTF-8: <codecs.CodecInfo object for encoding utf-8 at
0x1007773a8>
search1: Searching for: no-such-encoding
search2: Searching for: no-such-encoding
ERROR: unknown encoding: no-such-encoding
```

The `CodecInfo` instance returned by the search function tells `codecs` how to encode and decode using all of the different mechanisms supported: stateless, incremental, and stream. `codecs` includes base classes to help with setting up a character map encoding. This example puts all of the pieces together to register a search function that returns a `CodecInfo` instance configured for the invertcaps codec.

```
# codecs_invertcaps_register.py

import codecs

from codecs_invertcaps_charmap import encoding_map, decoding_map


class InvertCapsCodec(codecs.Codec):
    "Stateless encoder/decoder"
    def encode(self, input, errors='strict'):
        return codecs.charmap_encode(input, errors, encoding_map)

    def decode(self, input, errors='strict'):
        return codecs.charmap_decode(input, errors, decoding_map)


class InvertCapsIncrementalEncoder(codecs.IncrementalEncoder):
    def encode(self, input, final=False):
        data, nbytes = codecs.charmap_encode(input,
                                             self.errors,
                                             encoding_map)
        return data


class InvertCapsIncrementalDecoder(codecs.IncrementalDecoder):
    def decode(self, input, final=False):
        data, nbytes = codecs.charmap_decode(input,
                                             self.errors,
                                             decoding_map)
        return data


class InvertCapsStreamReader(InvertCapsCodec,
                             codecs.StreamReader):
    pass


class InvertCapsStreamWriter(InvertCapsCodec,
                             codecs.StreamWriter):
    pass


def find_invertcaps(encoding):
    """Return the codec for 'invertcaps'.
    """
    if encoding == 'invertcaps':
        return codecs.CodecInfo(
            name='invertcaps',
            encode=InvertCapsCodec().encode,
            decode=InvertCapsCodec().decode,
            incrementalencoder=InvertCapsIncrementalEncoder,
            incrementaldecoder=InvertCapsIncrementalDecoder,
            streamreader=InvertCapsStreamReader,
            streamwriter=InvertCapsStreamWriter,
        )
    return None


codecs.register(find_invertcaps)

if __name__ == '__main__':

    # Stateless encoder/decoder
    encoder = codecs.getencoder('invertcaps')
    text = 'abcDEF'
    encoded_text, consumed = encoder(text)
    print('Encoded "{}" to "{}", consuming {} characters'.format(
        text, encoded_text, consumed))

    # Stream writer
    import io
    buffer = io.BytesIO()
    writer = codecs.getwriter('invertcaps')(buffer)
    print('StreamWriter for io buffer: ')
    print('  writing "abcDEF"')
    writer.write('abcDEF')
    print('  buffer contents: ', buffer.getvalue())

    # Incremental decoder
    decoder_factory = codecs.getincrementaldecoder('invertcaps')
    decoder = decoder_factory()
    decoded_text_parts = []
    for c in encoded_text:
        decoded_text_parts.append(
            decoder.decode(bytes([c]), final=False)
        )
    decoded_text_parts.append(decoder.decode(b'', final=True))
    decoded_text = ''.join(decoded_text_parts)
    print('IncrementalDecoder converted {!r} to {!r}'.format(
        encoded_text, decoded_text))
```

The stateless encoder/decoder base class is `Codec`. Override `encode()` and `decode()` with the new implementation (in this case, calling `charmap_encode()` and `charmap_decode()` respectively). Each method must return a tuple containing the transformed data and the number of the input bytes or characters consumed. Conveniently, `charmap_encode()` and `charmap_decode()` already return that information.

`IncrementalEncoder` and `IncrementalDecoder` serve as base classes for the incremental interfaces. The `encode()` and `decode()` methods of the incremental classes are defined in such a way that they only return the actual transformed data. Any information about buffering is maintained as internal state. The invertcaps encoding does not need to buffer data (it uses a one-to-one mapping). For encodings that produce a different amount of output depending on the data being processed, such as compression algorithms, `BufferedIncrementalEncoder` and `BufferedIncrementalDecoder` are more appropriate base classes, since they manage the unprocessed portion of the input.

`StreamReader` and `StreamWriter` need `encode()` and `decode()` methods, too, and since they are expected to return the same value as the version from `Codec` multiple inheritance can be used for the implementation.

```
$ python3 codecs_invertcaps_register.py

Encoded "abcDEF" to "b'ABCdef'", consuming 6 characters
StreamWriter for io buffer:
  writing "abcDEF"
  buffer contents:  b'ABCdef'
IncrementalDecoder converted b'ABCdef' to 'abcDEF'
```

See also

- Standard library documentation for codecs
- `locale` – Accessing and managing the localization-based configuration settings and behaviors.
- `io` – The `io` module includes file and stream wrappers that handle encoding and decoding, too.
- `socketserver` – For a more detailed example of an echo server, see the `socketserver` module.
- `encodings` – Package in the standard library containing the encoder/decoder implementations provided by Python.
- **PEP 100** – Python Unicode Integration PEP.
- Unicode HOWTO – The official guide for using Unicode with Python.
- Text vs. Data Instead of Unicode vs. 8-bit – Section of the “What’s New” article for Python 3.0 covering the text handling changes.
- Python Unicode Objects – Fredrik Lundh’s article about using non-ASCII character sets in Python 2.0.
- How to Use UTF-8 with Python – Evan Jones’ quick guide to working with Unicode, including XML data and the Byte-Order Marker.
- On the Goodness of Unicode – Introduction to internationalization and Unicode by Tim Bray.
- On Character Strings – A look at the history of string processing in programming languages, by Tim Bray.
- Characters vs. Bytes – Part one of Tim Bray’s “essay on modern character string processing for computer programmers.” This installment covers in-memory representation of text in formats other than ASCII bytes.
- Endianness – Explanation of endianness in Wikipedia.
- W3C XML Entity Definitions for Characters – Specification for XML representations of character references that cannot be represented in an encoding.

mmap — Memory-map Files
io — Text, Binary, and Raw Stream I/O Tools

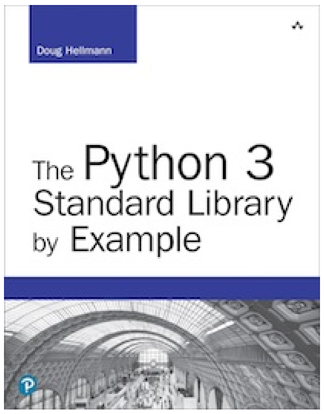
Quick Links

Unicode Primer
Encodings
Working with Files
Byte Order
Error Handling
Encoding Errors
Decoding Errors
Encoding Translation
Non-Unicode Encodings
Incremental Encoding
Unicode Data and Network Communication
Defining a Custom Encoding

This page was last updated 2016-12-28.

Navigation

mmap — Memory-map Files
io — Text, Binary, and Raw Stream I/O Tools



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?