

7.2. `codecs` — Codec registry and base classes

Source code: [Lib/codecs.py](#)

This module defines base classes for standard Python codecs (encoders and decoders) and provides access to the internal Python codec registry, which manages the codec and error handling lookup process. Most standard codecs are [text encodings](#), which encode text to bytes, but there are also codecs provided that encode text to text, and bytes to bytes. Custom codecs may encode and decode between arbitrary types, but some module features are restricted to use specifically with [text encodings](#), or with codecs that encode to [bytes](#).

The module defines the following functions for encoding and decoding with any codec:

`codecs.encode(obj, encoding='utf-8', errors='strict')`

Encodes *obj* using the codec registered for *encoding*.

Errors may be given to set the desired error handling scheme. The default error handler is 'strict' meaning that encoding errors raise [ValueError](#) (or a more codec specific subclass, such as [UnicodeEncodeError](#)). Refer to [Codec Base Classes](#) for more information on codec error handling.

`codecs.decode(obj, encoding='utf-8', errors='strict')`

Decodes *obj* using the codec registered for *encoding*.

Errors may be given to set the desired error handling scheme. The default error handler is 'strict' meaning that decoding errors raise [ValueError](#) (or a more codec specific subclass, such as [UnicodeDecodeError](#)). Refer to [Codec Base Classes](#) for more information on codec error handling.

The full details for each codec can also be looked up directly:

`codecs.lookup(encoding)`

Looks up the codec info in the Python codec registry and returns a [CodecInfo](#) object as defined below.

Encodings are first looked up in the registry's cache. If not found, the list of registered search functions is scanned. If no [CodecInfo](#) object is found, a [LookupError](#) is raised. Otherwise, the [CodecInfo](#) object is stored in the cache and returned to the caller.

```
class codecs. CodecInfo(encode, decode, streamreader=None,  
streamwriter=None, incrementalencoder=None, incrementaldecoder=None,  
name=None)
```

Codec details when looking up the codec registry. The constructor arguments are stored in attributes of the same name:

name

The name of the encoding.

encode

decode

The stateless encoding and decoding functions. These must be functions or methods which have the same interface as the [encode\(\)](#) and [decode\(\)](#) methods of Codec instances (see [Codec Interface](#)). The functions or methods are expected to work in a stateless mode.

incrementalencoder

incrementaldecoder

Incremental encoder and decoder classes or factory functions. These have to provide the interface defined by the base classes [IncrementalEncoder](#) and [IncrementalDecoder](#), respectively. Incremental codecs can maintain state.

streamwriter

streamreader

Stream writer and reader classes or factory functions. These have to provide the interface defined by the base classes [StreamWriter](#) and [StreamReader](#), respectively. Stream codecs can maintain state.

To simplify access to the various codec components, the module provides these additional functions which use [lookup\(\)](#) for the codec lookup:

```
codecs.getencoder(encoding)
```

Look up the codec for the given encoding and return its encoder function.

Raises a [LookupError](#) in case the encoding cannot be found.

```
codecs.getdecoder(encoding)
```

Look up the codec for the given encoding and return its decoder function.

Raises a [LookupError](#) in case the encoding cannot be found.

```
codecs.getincrementalencoder(encoding)
```

Look up the codec for the given encoding and return its incremental encoder class or factory function.

Raises a [LookupError](#) in case the encoding cannot be found or the codec doesn't support an incremental encoder.

`codecs.getincrementaldecoder(encoding)`

Look up the codec for the given encoding and return its incremental decoder class or factory function.

Raises a [LookupError](#) in case the encoding cannot be found or the codec doesn't support an incremental decoder.

`codecs.getreader(encoding)`

Look up the codec for the given encoding and return its [StreamReader](#) class or factory function.

Raises a [LookupError](#) in case the encoding cannot be found.

`codecs.getwriter(encoding)`

Look up the codec for the given encoding and return its [StreamWriter](#) class or factory function.

Raises a [LookupError](#) in case the encoding cannot be found.

Custom codecs are made available by registering a suitable codec search function:

`codecs.register(search_function)`

Register a codec search function. Search functions are expected to take one argument, being the encoding name in all lower case letters, and return a [CodecInfo](#) object. In case a search function cannot find a given encoding, it should return `None`.

Note: Search function registration is not currently reversible, which may cause problems in some cases, such as unit testing or module reloading.

While the builtin [open\(\)](#) and the associated [io](#) module are the recommended approach for working with encoded text files, this module provides additional utility functions and classes that allow the use of a wider range of codecs when working with binary files:

`codecs.open(filename, mode='r', encoding=None, errors='strict', buffering=1)`

Open an encoded file using the given *mode* and return an instance of [StreamReaderWriter](#), providing transparent encoding/decoding. The default file mode is 'r', meaning to open the file in read mode.

Note: Underlying encoded files are always opened in binary mode. No automatic conversion of `'\n'` is done on reading and writing. The *mode* argument may be any binary mode acceptable to the built-in `open()` function; the `'b'` is automatically added.

encoding specifies the encoding which is to be used for the file. Any encoding that encodes to and decodes from bytes is allowed, and the data types supported by the file methods depend on the codec used.

errors may be given to define the error handling. It defaults to `'strict'` which causes a `ValueError` to be raised in case an encoding error occurs.

buffering has the same meaning as for the built-in `open()` function. It defaults to line buffered.

`codecs.EncodedFile(file, data_encoding, file_encoding=None, errors='strict')`

Return a `StreamRecoder` instance, a wrapped version of *file* which provides transparent transcoding. The original file is closed when the wrapped version is closed.

Data written to the wrapped file is decoded according to the given *data_encoding* and then written to the original file as bytes using *file_encoding*. Bytes read from the original file are decoded according to *file_encoding*, and the result is encoded using *data_encoding*.

If *file_encoding* is not given, it defaults to *data_encoding*.

errors may be given to define the error handling. It defaults to `'strict'`, which causes `ValueError` to be raised in case an encoding error occurs.

`codecs.iterencode(iterator, encoding, errors='strict', **kwargs)`

Uses an incremental encoder to iteratively encode the input provided by *iterator*. This function is a `generator`. The *errors* argument (as well as any other keyword argument) is passed through to the incremental encoder.

This function requires that the codec accept text `str` objects to encode. Therefore it does not support bytes-to-bytes encoders such as `base64_codec`.

`codecs.iterdecode(iterator, encoding, errors='strict', **kwargs)`

Uses an incremental decoder to iteratively decode the input provided by *iterator*. This function is a `generator`. The *errors* argument (as well as any other keyword argument) is passed through to the incremental decoder.

This function requires that the codec accept `bytes` objects to decode. Therefore it does not support text-to-text encoders such as `rot_13`, although `rot_13` may be used equivalently with `iterencode()`.

The module also provides the following constants which are useful for reading and writing to platform dependent files:

```
codecs.BOM
codecs.BOM_BE
codecs.BOM_LE
codecs.BOM_UTF8
codecs.BOM_UTF16
codecs.BOM_UTF16_BE
codecs.BOM_UTF16_LE
codecs.BOM_UTF32
codecs.BOM_UTF32_BE
codecs.BOM_UTF32_LE
```

These constants define various byte sequences, being Unicode byte order marks (BOMs) for several encodings. They are used in UTF-16 and UTF-32 data streams to indicate the byte order used, and in UTF-8 as a Unicode signature. `BOM_UTF16` is either `BOM_UTF16_BE` or `BOM_UTF16_LE` depending on the platform's native byte order, `BOM` is an alias for `BOM_UTF16`, `BOM_LE` for `BOM_UTF16_LE` and `BOM_BE` for `BOM_UTF16_BE`. The others represent the BOM in UTF-8 and UTF-32 encodings.

7.2.1. Codec Base Classes

The `codecs` module defines a set of base classes which define the interfaces for working with codec objects, and can also be used as the basis for custom codec implementations.

Each codec has to define four interfaces to make it usable as codec in Python: stateless encoder, stateless decoder, stream reader and stream writer. The stream reader and writers typically reuse the stateless encoder/decoder to implement the file protocols. Codec authors also need to define how the codec will handle encoding and decoding errors.

7.2.1.1. Error Handlers

To simplify and standardize error handling, codecs may implement different error handling schemes by accepting the `errors` string argument. The following string values are defined and implemented by all standard Python codecs:

Value	Meaning
'strict'	Raise UnicodeError (or a subclass); this is the default. Implemented in strict_errors() .
'ignore'	Ignore the malformed data and continue without further notice. Implemented in ignore_errors() .

The following error handlers are only applicable to [text encodings](#):

Value	Meaning
'replace'	Replace with a suitable replacement marker; Python will use the official U+FFFD REPLACEMENT CHARACTER for the built-in codecs on decoding, and '?' on encoding. Implemented in replace_errors() .
'xmlcharrefreplace'	Replace with the appropriate XML character reference (only for encoding). Implemented in xmlcharrefreplace_errors() .
'backslashreplace'	Replace with backslashed escape sequences. Implemented in backslashreplace_errors() .
'namereplace'	Replace with <code>\N{...}</code> escape sequences (only for encoding). Implemented in namereplace_errors() .
'surrogateescape'	On decoding, replace byte with individual surrogate code ranging from U+DC80 to U+DCFF. This code will then be turned back into the same byte when the 'surrogateescape' error handler is used when encoding the data. (See PEP 383 for more.)

In addition, the following error handler is specific to the given codecs:

Value	Codecs	Meaning
'surrogatepass'	utf-8, utf-16, utf-32, utf-16-be, utf-16-le, utf-32-be, utf-32-le	Allow encoding and decoding of surrogate codes. These codecs normally treat the presence of surrogates as an error.

New in version 3.1: The 'surrogateescape' and 'surrogatepass' error handlers.

Changed in version 3.4: The 'surrogatepass' error handlers now works with utf-16* and utf-32* codecs.

New in version 3.5: The 'namereplace' error handler.

Changed in version 3.5: The 'backslashreplace' error handlers now works with decoding and translating.

The set of allowed values can be extended by registering a new named error handler:

`codecs.register_error(name, error_handler)`

Register the error handling function *error_handler* under the name *name*. The *error_handler* argument will be called during encoding and decoding in case of an error, when *name* is specified as the errors parameter.

For encoding, *error_handler* will be called with a [UnicodeEncodeError](#) instance, which contains information about the location of the error. The error handler must either raise this or a different exception, or return a tuple with a replacement for the unencodable part of the input and a position where encoding should continue. The replacement may be either [str](#) or [bytes](#). If the replacement is bytes, the encoder will simply copy them into the output buffer. If the replacement is a string, the encoder will encode the replacement. Encoding continues on original input at the specified position. Negative position values will be treated as being relative to the end of the input string. If the resulting position is out of bound an [IndexError](#) will be raised.

Decoding and translating works similarly, except [UnicodeDecodeError](#) or [UnicodeTranslateError](#) will be passed to the handler and that the replacement from the error handler will be put into the output directly.

Previously registered error handlers (including the standard error handlers) can be looked up by name:

`codecs.lookup_error(name)`

Return the error handler previously registered under the name *name*.

Raises a [LookupError](#) in case the handler cannot be found.

The following standard error handlers are also made available as module level functions:

`codecs.strict_errors(exception)`

Implements the 'strict' error handling: each encoding or decoding error raises a [UnicodeError](#).

`codecs.replace_errors(exception)`

Implements the 'replace' error handling (for [text encodings](#) only): substitutes '?' for encoding errors (to be encoded by the codec), and '\ufffd' (the Unicode replacement character) for decoding errors.

`codecs.ignore_errors(exception)`

Implements the 'ignore' error handling: malformed data is ignored and encoding or decoding is continued without further notice.

`codecs.xmlcharrefreplace_errors(exception)`

Implements the 'xmlcharrefreplace' error handling (for encoding with [text encodings](#) only): the unencodable character is replaced by an appropriate XML character reference.

`codecs.backslashreplace_errors(exception)`

Implements the 'backslashreplace' error handling (for [text encodings](#) only): malformed data is replaced by a backslashed escape sequence.

`codecs.namereplace_errors(exception)`

Implements the 'namereplace' error handling (for encoding with [text encodings](#) only): the unencodable character is replaced by a `\N{...}` escape sequence.

New in version 3.5.

7.2.1.2. Stateless Encoding and Decoding

The base `Codec` class defines these methods which also define the function interfaces of the stateless encoder and decoder:

`Codec.encode(input[, errors])`

Encodes the object *input* and returns a tuple (output object, length consumed). For instance, [text encoding](#) converts a string object to a bytes object using a particular character set encoding (e.g., `cp1252` or `iso-8859-1`).

The *errors* argument defines the error handling to apply. It defaults to 'strict' handling.

The method may not store state in the `Codec` instance. Use [StreamWriter](#) for codecs which have to keep state in order to make encoding efficient.

The encoder must be able to handle zero length input and return an empty object of the output object type in this situation.

`Codec.decode(input[, errors])`

Decodes the object *input* and returns a tuple (output object, length consumed). For instance, for a [text encoding](#), decoding converts a bytes object encoded using a particular character set encoding to a string object.

For text encodings and bytes-to-bytes codecs, *input* must be a bytes object or one which provides the read-only buffer interface – for example, buffer objects and memory mapped files.

The *errors* argument defines the error handling to apply. It defaults to 'strict' handling.

The method may not store state in the Codec instance. Use [StreamReader](#) for codecs which have to keep state in order to make decoding efficient.

The decoder must be able to handle zero length input and return an empty object of the output object type in this situation.

7.2.1.3. Incremental Encoding and Decoding

The [IncrementalEncoder](#) and [IncrementalDecoder](#) classes provide the basic interface for incremental encoding and decoding. Encoding/decoding the input isn't done with one call to the stateless encoder/decoder function, but with multiple calls to the [encode\(\)/decode\(\)](#) method of the incremental encoder/decoder. The incremental encoder/decoder keeps track of the encoding/decoding process during method calls.

The joined output of calls to the [encode\(\)/decode\(\)](#) method is the same as if all the single inputs were joined into one, and this input was encoded/decoded with the stateless encoder/decoder.

7.2.1.3.1. IncrementalEncoder Objects

The [IncrementalEncoder](#) class is used for encoding an input in multiple steps. It defines the following methods which every incremental encoder must define in order to be compatible with the Python codec registry.

```
class codecs.IncrementalEncoder(errors='strict')
```

Constructor for an [IncrementalEncoder](#) instance.

All incremental encoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The [IncrementalEncoder](#) may implement different error handling schemes by providing the *errors* keyword argument. See [Error Handlers](#) for possible values.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the [IncrementalEncoder](#) object.

encode(object[, final])

Encodes *object* (taking the current state of the encoder into account) and returns the resulting encoded object. If this is the last call to `encode()` *final* must be true (the default is false).

reset()

Reset the encoder to the initial state. The output is discarded: call `.encode(object, final=True)`, passing an empty byte or text string if necessary, to reset the encoder and to get the output.

getstate()

Return the current state of the encoder which must be an integer. The implementation should make sure that 0 is the most common state. (States that are more complicated than integers can be converted into an integer by marshaling/pickling the state and encoding the bytes of the resulting string into an integer).

setstate(state)

Set the state of the encoder to *state*. *state* must be an encoder state returned by `getstate()`.

7.2.1.3.2. IncrementalDecoder Objects

The `IncrementalDecoder` class is used for decoding an input in multiple steps. It defines the following methods which every incremental decoder must define in order to be compatible with the Python codec registry.

```
class codecs.IncrementalDecoder(errors='strict')
```

Constructor for an `IncrementalDecoder` instance.

All incremental decoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The `IncrementalDecoder` may implement different error handling schemes by providing the *errors* keyword argument. See [Error Handlers](#) for possible values.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `IncrementalDecoder` object.

decode(object[, final])

Decodes *object* (taking the current state of the decoder into account) and returns the resulting decoded object. If this is the last call to `decode()` *final*

must be true (the default is false). If *final* is true the decoder must decode the input completely and must flush all buffers. If this isn't possible (e.g. because of incomplete byte sequences at the end of the input) it must initiate error handling just like in the stateless case (which might raise an exception).

reset()

Reset the decoder to the initial state.

getstate()

Return the current state of the decoder. This must be a tuple with two items, the first must be the buffer containing the still undecoded input. The second must be an integer and can be additional state info. (The implementation should make sure that 0 is the most common additional state info.) If this additional state info is 0 it must be possible to set the decoder to the state which has no input buffered and 0 as the additional state info, so that feeding the previously buffered input to the decoder returns it to the previous state without producing any output. (Additional state info that is more complicated than integers can be converted into an integer by marshaling/pickling the info and encoding the bytes of the resulting string into an integer.)

setstate(state)

Set the state of the encoder to *state*. *state* must be a decoder state returned by [getstate\(\)](#).

7.2.1.4. Stream Encoding and Decoding

The [StreamWriter](#) and [StreamReader](#) classes provide generic working interfaces which can be used to implement new encoding submodules very easily. See [encodings.utf_8](#) for an example of how this is done.

7.2.1.4.1. StreamWriter Objects

The [StreamWriter](#) class is a subclass of [Codec](#) and defines the following methods which every stream writer must define in order to be compatible with the Python codec registry.

```
class codecs.StreamWriter(stream, errors='strict')
```

Constructor for a [StreamWriter](#) instance.

All stream writers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The *stream* argument must be a file-like object open for writing text or binary data, as appropriate for the specific codec.

The `StreamWriter` may implement different error handling schemes by providing the *errors* keyword argument. See [Error Handlers](#) for the standard error handlers the underlying stream codec may support.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `StreamWriter` object.

write(object)

Writes the object's contents encoded to the stream.

writelines(list)

Writes the concatenated list of strings to the stream (possibly by reusing the `write()` method). The standard bytes-to-bytes codecs do not support this method.

reset()

Flushes and resets the codec buffers used for keeping state.

Calling this method should ensure that the data on the output is put into a clean state that allows appending of new fresh data without having to rescan the whole stream to recover state.

In addition to the above methods, the `StreamWriter` must also inherit all other methods and attributes from the underlying stream.

7.2.1.4.2. StreamReader Objects

The `StreamReader` class is a subclass of `Codec` and defines the following methods which every stream reader must define in order to be compatible with the Python codec registry.

class `codecs.StreamReader(stream, errors='strict')`

Constructor for a `StreamReader` instance.

All stream readers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The *stream* argument must be a file-like object open for reading text or binary data, as appropriate for the specific codec.

The `StreamReader` may implement different error handling schemes by providing the *errors* keyword argument. See [Error Handlers](#) for the standard error handlers the underlying stream codec may support.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `StreamReader` object.

The set of allowed values for the *errors* argument can be extended with `register_error()`.

`read([size[, chars[, firstline]])`

Decodes data from the stream and returns the resulting object.

The *chars* argument indicates the number of decoded code points or bytes to return. The `read()` method will never return more data than requested, but it might return less, if there is not enough available.

The *size* argument indicates the approximate maximum number of encoded bytes or code points to read for decoding. The decoder can modify this setting as appropriate. The default value -1 indicates to read and decode as much as possible. This parameter is intended to prevent having to decode huge files in one step.

The *firstline* flag indicates that it would be sufficient to only return the first line, if there are decoding errors on later lines.

The method should use a greedy read strategy meaning that it should read as much data as is allowed within the definition of the encoding and the given size, e.g. if optional encoding endings or state markers are available on the stream, these should be read too.

`readline([size[, keepends]])`

Read one line from the input stream and return the decoded data.

size, if given, is passed as size argument to the stream's `read()` method.

If *keepends* is false line-endings will be stripped from the lines returned.

`readlines([sizehint[, keepends]])`

Read all lines available on the input stream and return them as a list of lines.

Line-endings are implemented using the codec's decoder method and are included in the list entries if *keepends* is true.

sizehint, if given, is passed as the *size* argument to the stream's `read()` method.

reset()

Resets the codec buffers used for keeping state.

Note that no stream repositioning should take place. This method is primarily intended to be able to recover from decoding errors.

In addition to the above methods, the `StreamReader` must also inherit all other methods and attributes from the underlying stream.

7.2.1.4.3. StreamReaderWriter Objects

The `StreamReaderWriter` is a convenience class that allows wrapping streams which work in both read and write modes.

The design is such that one can use the factory functions returned by the `lookup()` function to construct the instance.

class codecs. StreamReaderWriter(stream, Reader, Writer, errors='strict')

Creates a `StreamReaderWriter` instance. *stream* must be a file-like object. *Reader* and *Writer* must be factory functions or classes providing the `StreamReader` and `StreamWriter` interface resp. Error handling is done in the same way as defined for the stream readers and writers.

`StreamReaderWriter` instances define the combined interfaces of `StreamReader` and `StreamWriter` classes. They inherit all other methods and attributes from the underlying stream.

7.2.1.4.4. StreamRecoder Objects

The `StreamRecoder` translates data from one encoding to another, which is sometimes useful when dealing with different encoding environments.

The design is such that one can use the factory functions returned by the `lookup()` function to construct the instance.

class codecs. StreamRecoder(stream, encode, decode, Reader, Writer, errors='strict')

Creates a `StreamRecoder` instance which implements a two-way conversion: *encode* and *decode* work on the frontend — the data visible to code calling `read()` and `write()`, while *Reader* and *Writer* work on the backend — the data in *stream*.

You can use these objects to do transparent transcodings from e.g. Latin-1 to UTF-8 and back.

The *stream* argument must be a file-like object.

The *encode* and *decode* arguments must adhere to the Codec interface. *Reader* and *Writer* must be factory functions or classes providing objects of the [StreamReader](#) and [StreamWriter](#) interface respectively.

Error handling is done in the same way as defined for the stream readers and writers.

[StreamRecoder](#) instances define the combined interfaces of [StreamReader](#) and [StreamWriter](#) classes. They inherit all other methods and attributes from the underlying stream.

7.2.2. Encodings and Unicode

Strings are stored internally as sequences of code points in range 0x0–0x10FFFF. (See [PEP 393](#) for more details about the implementation.) Once a string object is used outside of CPU and memory, endianness and how these arrays are stored as bytes become an issue. As with other codecs, serialising a string into a sequence of bytes is known as *encoding*, and recreating the string from the sequence of bytes is known as *decoding*. There are a variety of different text serialisation codecs, which are collectively referred to as [text encodings](#).

The simplest text encoding (called 'latin-1' or 'iso-8859-1') maps the code points 0–255 to the bytes 0x0–0xff, which means that a string object that contains code points above U+00FF can't be encoded with this codec. Doing so will raise a [UnicodeEncodeError](#) that looks like the following (although the details of the error message may differ): `UnicodeEncodeError: 'latin-1' codec can't encode character '\u1234' in position 3: ordinal not in range(256)`.

There's another group of encodings (the so called charmap encodings) that choose a different subset of all Unicode code points and how these code points are mapped to the bytes 0x0–0xff. To see how this is done simply open e.g. `encodings/cp1252.py` (which is an encoding that is used primarily on Windows). There's a string constant with 256 characters that shows you which character is mapped to which byte value.

All of these encodings can only encode 256 of the 1114112 code points defined in Unicode. A simple and straightforward way that can store each Unicode code point, is to store each code point as four consecutive bytes. There are two possibilities: store the bytes in big endian or in little endian order. These two encodings are called

UTF-32-BE and UTF-32-LE respectively. Their disadvantage is that if e.g. you use UTF-32-BE on a little endian machine you will always have to swap bytes on encoding and decoding. UTF-32 avoids this problem: bytes will always be in natural endianness. When these bytes are read by a CPU with a different endianness, then bytes have to be swapped though. To be able to detect the endianness of a UTF-16 or UTF-32 byte sequence, there's the so called BOM ("Byte Order Mark"). This is the Unicode character U+FEFF. This character can be prepended to every UTF-16 or UTF-32 byte sequence. The byte swapped version of this character (0xFFFE) is an illegal character that may not appear in a Unicode text. So when the first character in an UTF-16 or UTF-32 byte sequence appears to be a U+FFFE the bytes have to be swapped on decoding. Unfortunately the character U+FEFF had a second purpose as a ZERO WIDTH NO-BREAK SPACE: a character that has no width and doesn't allow a word to be split. It can e.g. be used to give hints to a ligature algorithm. With Unicode 4.0 using U+FEFF as a ZERO WIDTH NO-BREAK SPACE has been deprecated (with U+2060 (WORD JOINER) assuming this role). Nevertheless Unicode software still must be able to handle U+FEFF in both roles: as a BOM it's a device to determine the storage layout of the encoded bytes, and vanishes once the byte sequence has been decoded into a string; as a ZERO WIDTH NO-BREAK SPACE it's a normal character that will be decoded like any other.

There's another encoding that is able to encoding the full range of Unicode characters: UTF-8. UTF-8 is an 8-bit encoding, which means there are no issues with byte order in UTF-8. Each byte in a UTF-8 byte sequence consists of two parts: marker bits (the most significant bits) and payload bits. The marker bits are a sequence of zero to four 1 bits followed by a 0 bit. Unicode characters are encoded like this (with x being payload bits, which when concatenated give the Unicode character):

Range	Encoding
U-00000000 ... U-0000007F	0xxxxxxx
U-00000080 ... U-000007FF	110xxxxx 10xxxxxx
U-00000800 ... U-0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
U-00010000 ... U-0010FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

The least significant bit of the Unicode character is the rightmost x bit.

As UTF-8 is an 8-bit encoding no BOM is required and any U+FEFF character in the decoded string (even if it's the first character) is treated as a ZERO WIDTH NO-BREAK SPACE.

Without external information it's impossible to reliably determine which encoding was used for encoding a string. Each charmap encoding can decode any random byte sequence. However that's not possible with UTF-8, as UTF-8 byte sequences

have a structure that doesn't allow arbitrary byte sequences. To increase the reliability with which a UTF-8 encoding can be detected, Microsoft invented a variant of UTF-8 (that Python 2.5 calls "utf-8-sig") for its Notepad program: Before any of the Unicode characters is written to the file, a UTF-8 encoded BOM (which looks like this as a byte sequence: 0xef, 0xbb, 0xbf) is written. As it's rather improbable that any charmap encoded file starts with these byte values (which would e.g. map to

LATIN SMALL LETTER I WITH DIAERESIS
RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
INVERTED QUESTION MARK

in iso-8859-1), this increases the probability that a utf-8-sig encoding can be correctly guessed from the byte sequence. So here the BOM is not used to be able to determine the byte order used for generating the byte sequence, but as a signature that helps in guessing the encoding. On encoding the utf-8-sig codec will write 0xef, 0xbb, 0xbf as the first three bytes to the file. On decoding utf-8-sig will skip those three bytes if they appear as the first three bytes in the file. In UTF-8, the use of the BOM is discouraged and should generally be avoided.

7.2.3. Standard Encodings

Python comes with a number of codecs built-in, either implemented as C functions or with dictionaries as mapping tables. The following table lists the codecs by name, together with a few common aliases, and the languages for which the encoding is likely used. Neither the list of aliases nor the list of languages is meant to be exhaustive. Notice that spelling alternatives that only differ in case or use a hyphen instead of an underscore are also valid aliases; therefore, e.g. 'utf-8' is a valid alias for the 'utf_8' codec.

CPython implementation detail: Some common encodings can bypass the codecs lookup machinery to improve performance. These optimization opportunities are only recognized by CPython for a limited set of (case insensitive) aliases: utf-8, utf8, latin-1, latin1, iso-8859-1, iso8859-1, mbcs (Windows only), ascii, us-ascii, utf-16, utf16, utf-32, utf32, and the same using underscores instead of dashes. Using alternative aliases for these encodings may result in slower execution.

Changed in version 3.6: Optimization opportunity recognized for us-ascii.

Many of the character sets support the same languages. They vary in individual characters (e.g. whether the EURO SIGN is supported or not), and in the assignment of characters to code positions. For the European languages in particular, the following variants typically exist:

- an ISO 8859 codeset

- a Microsoft Windows code page, which is typically derived from an 8859 codeset, but replaces control characters with additional graphic characters
- an IBM EBCDIC code page
- an IBM PC code page, which is ASCII compatible

Codec	Aliases	Languages
ascii	646, us-ascii	English
big5	big5-tw, csbig5	Traditional Chinese
big5hkscs	big5-hkscs, hkscs	Traditional Chinese
cp037	IBM037, IBM039	English
cp273	273, IBM273, csIBM273	German <i>New in version 3.4.</i>
cp424	EBCDIC-CP-HE, IBM424	Hebrew
cp437	437, IBM437	English
cp500	EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500	Western Europe
cp720		Arabic
cp737		Greek
cp775	IBM775	Baltic languages
cp850	850, IBM850	Western Europe
cp852	852, IBM852	Central and Eastern Europe
cp855	855, IBM855	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
cp856		Hebrew
cp857	857, IBM857	Turkish
cp858	858, IBM858	Western Europe
cp860	860, IBM860	Portuguese
cp861	861, CP-IS, IBM861	Icelandic
cp862	862, IBM862	Hebrew
cp863	863, IBM863	Canadian
cp864	IBM864	Arabic
cp865	865, IBM865	Danish, Norwegian
cp866	866, IBM866	Russian
cp869	869, CP-GR, IBM869	Greek

Codec	Aliases	Languages
cp874		Thai
cp875		Greek
cp932	932, ms932, mskanji, ms-kanji	Japanese
cp949	949, ms949, uhc	Korean
cp950	950, ms950	Traditional Chinese
cp1006		Urdu
cp1026	ibm1026	Turkish
cp1125	1125, ibm1125, cp866u, ruscii	Ukrainian <i>New in version 3.4.</i>
cp1140	ibm1140	Western Europe
cp1250	windows-1250	Central and Eastern Europe
cp1251	windows-1251	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
cp1252	windows-1252	Western Europe
cp1253	windows-1253	Greek
cp1254	windows-1254	Turkish
cp1255	windows-1255	Hebrew
cp1256	windows-1256	Arabic
cp1257	windows-1257	Baltic languages
cp1258	windows-1258	Vietnamese
cp65001		Windows only: Windows UTF-8 (CP_UTF8) <i>New in version 3.3.</i>
euc_jp	eucjp, ujis, u-jis	Japanese
euc_jis_2004	jisx0213, eucjis2004	Japanese
euc_jisx0213	eucjisx0213	Japanese
euc_kr	euckr, korean, ksc5601, ks_c-5601, ks_c-5601-1987, ksx1001, ks_x-1001	Korean
gb2312	chinese, csiso58gb231280, euc-cn, euccn, eucgb2312-	Simplified Chinese

Codec	Aliases	Languages
	cn, gb2312-1980, gb2312-80, iso- ir-58	
gbk	936, cp936, ms936	Unified Chinese
gb18030	gb18030-2000	Unified Chinese
hz	hzgb, hz-gb, hz-gb-2312	Simplified Chinese
iso2022_jp	csiso2022jp, iso2022jp, iso-2022-jp	Japanese
iso2022_jp_1	iso2022jp-1, iso-2022-jp-1	Japanese
iso2022_jp_2	iso2022jp-2, iso-2022-jp-2	Japanese, Korean, Simplified Chinese, Western Europe, Greek
iso2022_jp_2004	iso2022jp-2004, iso-2022-jp-2004	Japanese
iso2022_jp_3	iso2022jp-3, iso-2022-jp-3	Japanese
iso2022_jp_ext	iso2022jp-ext, iso-2022-jp-ext	Japanese
iso2022_kr	csiso2022kr, iso2022kr, iso-2022-kr	Korean
latin_1	iso-8859-1, iso8859-1, 8859, cp819, latin, latin1, L1	West Europe
iso8859_2	iso-8859-2, latin2, L2	Central and Eastern Europe
iso8859_3	iso-8859-3, latin3, L3	Esperanto, Maltese
iso8859_4	iso-8859-4, latin4, L4	Baltic languages
iso8859_5	iso-8859-5, cyrillic	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
iso8859_6	iso-8859-6, arabic	Arabic
iso8859_7	iso-8859-7, greek, greek8	Greek
iso8859_8	iso-8859-8, hebrew	Hebrew
iso8859_9	iso-8859-9, latin5, L5	Turkish
iso8859_10	iso-8859-10, latin6, L6	Nordic languages
iso8859_11	iso-8859-11, thai	Thai languages
iso8859_13	iso-8859-13, latin7, L7	Baltic languages
iso8859_14	iso-8859-14, latin8, L8	Celtic languages
iso8859_15	iso-8859-15, latin9, L9	Western Europe
iso8859_16	iso-8859-16, latin10, L10	South-Eastern Europe
johab	cp1361, ms1361	Korean

Codec	Aliases	Languages
koi8_r		Russian
koi8_t		Tajik <i>New in version 3.5.</i>
koi8_u		Ukrainian
kz1048	kz_1048, strk1048_2002, rk1048	Kazakh <i>New in version 3.5.</i>
mac_cyrillic	maccyrillic	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
mac_greek	macgreek	Greek
mac_iceland	maciceland	Icelandic
mac_latin2	maclatin2, maccentraleurope	Central and Eastern Europe
mac_roman	macroman, macintosh	Western Europe
mac_turkish	macturkish	Turkish
ptcp154	csptcp154, pt154, cp154, cyrillic-asian	Kazakh
shift_jis	csshiftjis, shiftjis, sjis, s_jis	Japanese
shift_jis_2004	shiftjis2004, sjis_2004, sjis2004	Japanese
shift_jisx0213	shiftjisx0213, sjisx0213, s_jisx0213	Japanese
utf_32	U32, utf32	all languages
utf_32_be	UTF-32BE	all languages
utf_32_le	UTF-32LE	all languages
utf_16	U16, utf16	all languages
utf_16_be	UTF-16BE	all languages
utf_16_le	UTF-16LE	all languages
utf_7	U7, unicode-1-1-utf-7	all languages
utf_8	U8, UTF, utf8	all languages
utf_8_sig		all languages

Changed in version 3.4: The utf-16* and utf-32* encoders no longer allow surrogate code points (U+D800–U+DFFF) to be encoded. The utf-32* decoders no longer decode byte sequences that correspond to surrogate code points.

7.2.4. Python Specific Encodings

A number of predefined codecs are specific to Python, so their codec names have no meaning outside Python. These are listed in the tables below based on the expected input and output types (note that while text encodings are the most common use case for codecs, the underlying codec infrastructure supports arbitrary data transforms rather than just text encodings). For asymmetric codecs, the stated purpose describes the encoding direction.

7.2.4.1. Text Encodings

The following codecs provide `str` to `bytes` encoding and `bytes-like object` to `str` decoding, similar to the Unicode text encodings.

Codec	Aliases	Purpose
idna		Implements RFC 3490 , see also encodings.idna . Only <code>errors='strict'</code> is supported.
mbcs	ansi, dbcs	Windows only: Encode operand according to the ANSI codepage (CP_ACP)
oem		Windows only: Encode operand according to the OEM codepage (CP_OEMCP) <i>New in version 3.6.</i>
palmos		Encoding of PalmOS 3.5
punycode		Implements RFC 3492 . Stateful codecs are not supported.
raw_unicode_escape		Latin-1 encoding with <code>\uXXXX</code> and <code>\UXXXXXXXX</code> for other code points. Existing backslashes are not escaped in any way. It is used in the Python pickle protocol.
undefined		Raise an exception for all conversions, even empty strings. The error handler is ignored.
unicode_escape		

Codec	Aliases	Purpose
		Encoding suitable as the contents of a Unicode literal in ASCII-encoded Python source code, except that quotes are not escaped. Decodes from Latin-1 source code. Beware that Python source code actually uses UTF-8 by default.
unicode_internal		Return the internal representation of the operand. Stateful codecs are not supported. <i>Deprecated since version 3.3:</i> This representation is obsoleted by PEP 393 .

7.2.4.2. Binary Transforms

The following codecs provide binary transforms: [bytes-like object](#) to [bytes](#) mappings. They are not supported by [bytes.decode\(\)](#) (which only produces [str](#) output).

Codec	Aliases	Purpose	Encoder / decoder
base64_codec [1]	base64, base_64	Convert operand to multiline MIME base64 (the result always includes a trailing '\n') <i>Changed in version 3.4:</i> accepts any bytes-like object as input for encoding and decoding	base64.encodebytes() / base64.decodebytes()
bz2_codec	bz2	Compress the operand using bz2	bz2.compress() / bz2.decompress()
hex_codec	hex	Convert operand to hexadecimal representation, with two digits per byte	binascii.b2a_hex() / binascii.a2b_hex()
quopri_codec			

Codec	Aliases	Purpose	Encoder / decoder
	quopri, quotedprintable, quoted_printable	Convert operand to MIME quoted printable	<code>quopri.encode()</code> with <code>quotetabs=True</code> / <code>quopri.decode()</code>
uu_codec	uu	Convert the operand using uuencode	<code>uu.encode()</code> / <code>uu.decode()</code>
zlib_codec	zip, zlib	Compress the operand using gzip	<code>zlib.compress()</code> / <code>zlib.decompress()</code>

[1] In addition to [bytes-like objects](#), 'base64_codec' also accepts ASCII-only instances of [str](#) for decoding

New in version 3.2: Restoration of the binary transforms.

Changed in version 3.4: Restoration of the aliases for the binary transforms.

7.2.4.3. Text Transforms

The following codec provides a text transform: a [str](#) to [str](#) mapping. It is not supported by `str.encode()` (which only produces [bytes](#) output).

Codec	Aliases	Purpose
rot_13	rot13	Returns the Caesar-cypher encryption of the operand

New in version 3.2: Restoration of the rot_13 text transform.

Changed in version 3.4: Restoration of the rot13 alias.

7.2.5. [encodings.idna](#) — Internationalized Domain Names in Applications

This module implements [RFC 3490](#) (Internationalized Domain Names in Applications) and [RFC 3492](#) (Nameprep: A Stringprep Profile for Internationalized Domain Names (IDN)). It builds upon the punycode encoding and [stringprep](#).

These RFCs together define a protocol to support non-ASCII characters in domain names. A domain name containing non-ASCII characters (such as `www.Alliancefrançaise.nu`) is converted into an ASCII-compatible encoding (ACE, such as `www.xn--alliancefranaise-npb.nu`). The ACE form of the domain name is then used in all places where arbitrary characters are not allowed by the protocol, such as DNS queries, HTTP *Host* fields, and so on. This conversion is carried out in the application; if possible invisible to the user: The application should

transparently convert Unicode domain labels to IDNA on the wire, and convert back ACE labels to Unicode before presenting them to the user.

Python supports this conversion in several ways: the `idna` codec performs conversion between Unicode and ACE, separating an input string into labels based on the separator characters defined in [section 3.1 of RFC 3490](#) and converting each label to ACE as required, and conversely separating an input byte string into labels based on the `.` separator and converting any ACE labels found into unicode. Furthermore, the `socket` module transparently converts Unicode host names to ACE, so that applications need not be concerned about converting host names themselves when they pass them to the socket module. On top of that, modules that have host names as function parameters, such as `http.client` and `ftplib`, accept Unicode host names (`http.client` then also transparently sends an IDNA hostname in the *Host* field if it sends that field at all).

When receiving host names from the wire (such as in reverse name lookup), no automatic conversion to Unicode is performed: Applications wishing to present such host names to the user should decode them to Unicode.

The module `encodings.idna` also implements the nameprep procedure, which performs certain normalizations on host names, to achieve case-insensitivity of international domain names, and to unify similar characters. The nameprep functions can be used directly if desired.

`encodings.idna.nameprep(label)`

Return the nameprepped version of *label*. The implementation currently assumes query strings, so `AllowUnassigned` is true.

`encodings.idna.ToASCII(label)`

Convert a label to ASCII, as specified in [RFC 3490](#). `UseSTD3ASCIIRules` is assumed to be false.

`encodings.idna.ToUnicode(label)`

Convert a label to Unicode, as specified in [RFC 3490](#).

7.2.6. `encodings.mbcs` — Windows ANSI codepage

Encode operand according to the ANSI codepage (CP_ACP).

Availability: Windows only.

Changed in version 3.3: Support any error handler.

Changed in version 3.2: Before 3.2, the *errors* argument was ignored; 'replace' was always used to encode, and 'ignore' to decode.

7.2.7. `encodings.utf_8_sig` — UTF-8 codec with BOM signature

This module implements a variant of the UTF-8 codec: On encoding a UTF-8 encoded BOM will be prepended to the UTF-8 encoded bytes. For the stateful encoder this is only done once (on the first write to the byte stream). For decoding an optional UTF-8 encoded BOM at the start of the data will be skipped.