8.8. codecs — Codec registry and base classes¶

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.

It defines the following functions:

`codecs.register`(*search_function*)¶

Register a codec search function. Search functions are expected to take one argument, the encoding name in all lower case letters, and return a `CodecInfo` object having the following attributes:

- `name` The name of the encoding;
- `encode` The stateless encoding function;
- `decode` The stateless decoding function;
- `incrementalencoder` An incremental encoder class or factory function;
- `incrementaldecoder` An incremental decoder class or factory function;
- `streamwriter` A stream writer class or factory function;
- `streamreader` A stream reader class or factory function.

The various functions or classes take the following arguments:

encode and *decode*: These must be functions or methods which have the same interface as the `encode()`/`decode()` methods of Codec instances (see Codec Interface). The functions/methods are expected to work in a stateless mode.

incrementalencoder and *incrementaldecoder*: These have to be factory functions providing the following interface:

```
factory(errors='strict')
```

The factory functions must return objects providing the interfaces defined by the base classes `IncrementalEncoder` and `IncrementalDecoder`, respectively. Incremental codecs can maintain state.

streamreader and *streamwriter*: These have to be factory functions providing the following interface:

```
factory(stream, errors='strict')
```

The factory functions must return objects providing the interfaces defined by the base classes `StreamWriter` and `StreamReader`, respectively. Stream codecs can maintain state.

Possible values for errors are

- `'strict'`: raise an exception in case of an encoding error
- `'replace'`: replace malformed data with a suitable replacement marker, such as `'?'` or `'\ufffd'`
- `'ignore'`: ignore malformed data and continue without further notice
- `'xmlcharrefreplace'`: replace with the appropriate XML character reference (for encoding only)
- `'backslashreplace'`: replace with backslashed escape sequences (for encoding only

as well as any other error handling name defined via `register_error()`.

In case a search function cannot find a given encoding, it should return `None`.

`codecs.lookup`(*encoding*)¶

Looks up the codec info in the Python codec registry and returns a `CodecInfo` object as defined above.

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.

To simplify access to the various codecs, 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.

New in version 2.5.

`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.

New in version 2.5.

`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.

`codecs.register_error`(*name*, *error_handler*)¶

Register the error handling function *error_handler* under the name *name*. *error_handler* 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 encoder will encode the replacement and continue encoding the 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 similar, 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.

`codecs.lookup_error`(*name*)¶

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

Raises a `LookupError` in case the handler cannot be found.

`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: malformed data is replaced with a suitable replacement character such as `'?'` in bytestrings and `'\ufffd'` in Unicode strings.

`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 only): the unencodable character is replaced by an appropriate XML character reference.

`codecs.backslashreplace_errors`(*exception*)¶
Implements the `backslashreplace` error handling (for encoding only): the unencodable character is replaced by a backslashed escape sequence.

To simplify working with encoded files or stream, the module also defines these utility functions:

`codecs.open`(*filename*, *mode*[, *encoding*[, *errors*[, *buffering*]]])¶

Open an encoded file using the given *mode* and return a wrapped version providing transparent encoding/decoding. The default file mode is `'r'` meaning to open the file in read mode.

Note

The wrapped version will only accept the object format defined by the codecs, i.e. Unicode objects for most built-in codecs. Output is also codec-dependent and will usually be Unicode as well.

Note

Files are always opened in binary mode, even if no binary mode was specified. This is done to avoid data loss due to encodings using 8-bit values. This means that no automatic conversion of `'\n'` is done on reading and writing.

encoding specifies the encoding which is to be used for the file.

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*, *input*[, *output*[, *errors*]])¶

Return a wrapped version of file which provides transparent encoding translation.

Strings written to the wrapped file are interpreted according to the given *input* encoding and then written to the original file as strings using the *output* encoding. The intermediate encoding will usually be Unicode but depends on the specified codecs.

If *output* is not given, it defaults to *input*.

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`(*iterable*, *encoding*[, *errors*])¶

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

New in version 2.5.

`codecs.iterdecode`(*iterable*, *encoding*[, *errors*])¶

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

New in version 2.5.

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 encodings of the Unicode byte order mark (BOM) used in UTF-16 and UTF-32 data streams to indicate the byte order used in the stream or file 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.

8.8.1. Codec Base Classes¶

The `codecs` module defines a set of base classes which define the interface and can also be used to easily write your own codecs for use in Python.

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.

The `Codec` class defines the interface for stateless encoders/decoders.

To simplify and standardize error handling, the `encode()` and `decode()` methods may implement different error handling schemes by providing 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.
`'ignore'`	Ignore the character and continue with the next.
`'replace'`	Replace with a suitable replacement character; Python will use the official U+FFFD REPLACEMENT CHARACTER for the built-in Unicode codecs on decoding and '?' on encoding.
`'xmlcharrefreplace'`	Replace with the appropriate XML character reference (only for encoding).
`'backslashreplace'`	Replace with backslashed escape sequences (only for encoding).

The set of allowed values can be extended via `register_error()`.

8.8.1.1. Codec Objects¶

The `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). While codecs are not restricted to use with Unicode, in a Unicode context, encoding converts a Unicode object to a plain string using a particular character set encoding (e.g., `cp1252` or `iso-8859-1`).

errors defines the error handling to apply. It defaults to `'strict'` handling.

The method may not store state in the `Codec` instance. Use `StreamCodec` for codecs which have to keep state in order to make encoding/decoding 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). In a Unicode context, decoding converts a plain string encoded using a particular character set encoding to a Unicode object.

input must be an object which provides the `bf_getreadbuf` buffer slot. Python strings, buffer objects and memory mapped files are examples of objects providing this slot.

errors defines the error handling to apply. It defaults to `'strict'` handling.

The method may not store state in the `Codec` instance. Use `StreamCodec` for codecs which have to keep state in order to make encoding/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.

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.

8.8.1.2. IncrementalEncoder Objects¶

New in version 2.5.

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*])¶

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. These parameters are predefined:

- `'strict'` Raise `ValueError` (or a subclass); this is the default.
- `'ignore'` Ignore the character and continue with the next.
- `'replace'` Replace with a suitable replacement character
- `'xmlcharrefreplace'` Replace with the appropriate XML character reference
- `'backslashreplace'` Replace with backslashed escape sequences.

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.

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

`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.

8.8.1.3. 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*])¶

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. These parameters are predefined:

- `'strict'` Raise `ValueError` (or a subclass); this is the default.
- `'ignore'` Ignore the character and continue with the next.
- `'replace'` Replace with a suitable replacement character.

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.

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

`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.

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.

8.8.1.4. 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*])¶

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.

stream must be a file-like object open for writing binary data.

The `StreamWriter` may implement different error handling schemes by providing the *errors* keyword argument. These parameters are predefined:

- `'strict'` Raise `ValueError` (or a subclass); this is the default.
- `'ignore'` Ignore the character and continue with the next.
- `'replace'` Replace with a suitable replacement character
- `'xmlcharrefreplace'` Replace with the appropriate XML character reference
- `'backslashreplace'` Replace with backslashed escape sequences.

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.

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

`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).

`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.

8.8.1.5. 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*])¶

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.

stream must be a file-like object open for reading (binary) data.

The `StreamReader` may implement different error handling schemes by providing the *errors* keyword argument. These parameters are defined:

- `'strict'` Raise `ValueError` (or a subclass); this is the default.
- `'ignore'` Ignore the character and continue with the next.
- `'replace'` Replace with a suitable replacement character.

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.

chars indicates the number of characters to read from the stream. `read()` will never return more than *chars* characters, but it might return less, if there are not enough characters available.

size indicates the approximate maximum number of bytes to read from the stream for decoding purposes. The decoder can modify this setting as appropriate. The default value -1 indicates to read and decode as much as possible. *size* is intended to prevent having to decode huge files in one step.

firstline 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.

Changed in version 2.4: *chars* argument added.

Changed in version 2.4.2: *firstline* argument added.

`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 `readline()` method.

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

Changed in version 2.4: *keepends* argument added.

`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.

The next two base classes are included for convenience. They are not needed by the codec registry, but may provide useful in practice.

8.8.1.6. StreamReaderWriter Objects¶

The `StreamReaderWriter` 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*)¶
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.

8.8.1.7. StreamRecoder Objects¶

The `StreamRecoder` provide a frontend - backend view of encoding data 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*)¶

Creates a `StreamRecoder` instance which implements a two-way conversion: *encode* and *decode* work on the frontend (the input to `read()` and output of `write()`) while *Reader* and *Writer* work on the backend (reading and writing to the stream).

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

stream must be a file-like object.

encode, *decode* must adhere to the `Codec` interface. *Reader*, *Writer* must be factory functions or classes providing objects of the `StreamReader` and `StreamWriter` interface respectively.

encode and *decode* are needed for the frontend translation, *Reader* and *Writer* for the backend translation. The intermediate format used is determined by the two sets of codecs, e.g. the Unicode codecs will use Unicode as the intermediate encoding.

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.

8.8.2. Encodings and Unicode¶

Unicode strings are stored internally as sequences of codepoints (to be precise as `Py_UNICODE` arrays). Depending on the way Python is compiled (either via *--enable-unicode=ucs2* or *--enable-unicode=ucs4*, with the former being the default) `Py_UNICODE` is either a 16-bit or 32-bit data type. Once a Unicode object is used outside of CPU and memory, CPU endianness and how these arrays are stored as bytes become an issue. Transforming a unicode object into a sequence of bytes is called encoding and recreating the unicode object from the sequence of bytes is known as decoding. There are many different methods for how this transformation can be done (these methods are also called encodings). The simplest method is to map the codepoints 0-255 to the bytes `0x0`-`0xff`. This means that a unicode object that contains codepoints above `U+00FF` can't be encoded with this method (which is called `'latin-1'` or `'iso-8859-1'`). `unicode.encode()` will raise a `UnicodeEncodeError` that looks like this: `UnicodeEncodeError: 'latin-1' codec can't encode character u'\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 codepoints 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 65536 (or 1114111) codepoints defined in unicode. A simple and straightforward way that can store each Unicode code point, is to store each codepoint as two consecutive bytes. There are two possibilities: Store the bytes in big endian or in little endian order. These two encodings are called UTF-16-BE and UTF-16-LE respectively. Their disadvantage is that if e.g. you use UTF-16-BE on a little endian machine you will always have to swap bytes on encoding and decoding. UTF-16 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 byte sequence, there's the so called BOM (the "Byte Order Mark"). This is the Unicode character `U+FEFF`. This character will be prepended to every UTF-16 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 byte sequence appears to be a `U+FFFE` the bytes have to be swapped on decoding. Unfortunately upto Unicode 4.0 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 Unicode 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 six 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-001FFFFF`	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
`U-00200000` ... `U-03FFFFFF`	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
`U-04000000` ... `U-7FFFFFFF`	1111110x 10xxxxxx 10xxxxxx 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 Unicode 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 Unicode 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.

8.8.3. 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.

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 a 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
cp424	EBCDIC-CP-HE, IBM424	Hebrew
cp437	437, IBM437	English
cp500	EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500	Western Europe
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
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
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
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
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-cn, gb2312-1980, gb2312-80, iso- ir-58	Simplified Chinese
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_13	iso-8859-13	Baltic languages
iso8859_14	iso-8859-14, latin8, L8	Celtic languages
iso8859_15	iso-8859-15	Western Europe
johab	cp1361, ms1361	Korean
koi8_r		Russian
koi8_u		Ukrainian
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	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 (BMP only)
utf_16_le	UTF-16LE	all languages (BMP only)
utf_7	U7, unicode-1-1-utf-7	all languages
utf_8	U8, UTF, utf8	all languages
utf_8_sig		all languages

A number of codecs are specific to Python, so their codec names have no meaning outside Python. Some of them don't convert from Unicode strings to byte strings, but instead use the property of the Python codecs machinery that any bijective function with one argument can be considered as an encoding.

For the codecs listed below, the result in the "encoding" direction is always a byte string. The result of the "decoding" direction is listed as operand type in the table.

Codec	Aliases	Operand type	Purpose
base64_codec	base64, base-64	byte string	Convert operand to MIME base64
bz2_codec	bz2	byte string	Compress the operand using bz2
hex_codec	hex	byte string	Convert operand to hexadecimal representation, with two digits per byte
idna		Unicode string	Implements **RFC 3490**, see also `encodings.idna`
mbcs	dbcs	Unicode string	Windows only: Encode operand according to the ANSI codepage (CP_ACP)
palmos		Unicode string	Encoding of PalmOS 3.5
punycode		Unicode string	Implements **RFC 3492**
quopri_codec	quopri, quoted-printable, quotedprintable	byte string	Convert operand to MIME quoted printable
raw_unicode_escape		Unicode string	Produce a string that is suitable as raw Unicode literal in Python source code
rot_13	rot13	Unicode string	Returns the Caesar-cypher encryption of the operand
string_escape		byte string	Produce a string that is suitable as string literal in Python source code
undefined		any	Raise an exception for all conversions. Can be used as the system encoding if no automatic *coercion* between byte and Unicode strings is desired.
unicode_escape		Unicode string	Produce a string that is suitable as Unicode literal in Python source code
unicode_internal		Unicode string	Return the internal representation of the operand
uu_codec	uu	byte string	Convert the operand using uuencode
zlib_codec	zip, zlib	byte string	Compress the operand using gzip

New in version 2.3: The `idna` and `punycode` encodings.

8.8.4. `encodings.idna` — Internationalized Domain Names in Applications¶

New in version 2.3.

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 allows to convert between Unicode and the ACE. 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 `httplib` and `ftplib`, accept Unicode host names (`httplib` 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**.

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

New in version 2.5.

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.

Table Of Contents