Unicode Objects and Codecs

Unicode Objects

Since the implementation of **PEP 393** in Python 3.3, Unicode objects internally use a variety of representations, in order to allow handling the complete range of Unicode characters while staying memory efficient. There are special cases for strings where all code points are below 128, 256, or 65536; otherwise, code points must be below 1114112 (which is the full Unicode range).

`Py_UNICODE*` and UTF-8 representations are created on demand and cached in the Unicode object. The `Py_UNICODE*` representation is deprecated and inefficient; it should be avoided in performance- or memory-sensitive situations.

Due to the transition between the old APIs and the new APIs, unicode objects can internally be in two states depending on how they were created:

- "canonical" unicode objects are all objects created by a non-deprecated unicode API. They use the most efficient representation allowed by the implementation.
- "legacy" unicode objects have been created through one of the deprecated APIs (typically `PyUnicode_FromUnicode()`) and only bear the `Py_UNICODE*` representation; you will have to call `PyUnicode_READY()` on them before calling any other API.

Unicode Type

These are the basic Unicode object types used for the Unicode implementation in Python:

`Py_UCS4`
`Py_UCS2`
`Py_UCS1`

These types are typedefs for unsigned integer types wide enough to contain characters of 32 bits, 16 bits and 8 bits, respectively. When dealing with single Unicode characters, use `Py_UCS4`.

New in version 3.3.

`Py_UNICODE`

This is a typedef of `wchar_t`, which is a 16-bit type or 32-bit type depending on the platform.

Changed in version 3.3: In previous versions, this was a 16-bit type or a 32-bit type depending on whether you selected a "narrow" or "wide" Unicode version of Python at build time.

PyASCIIObject
PyCompactUnicodeObject
PyUnicodeObject

These subtypes of `PyObject` represent a Python Unicode object. In almost all cases, they shouldn't be used directly, since all API functions that deal with Unicode objects take and return `PyObject` pointers.

New in version 3.3.

PyTypeObject **PyUnicode_Type**

This instance of `PyTypeObject` represents the Python Unicode type. It is exposed to Python code as `str`.

The following APIs are really C macros and can be used to do fast checks and to access internal read-only data of Unicode objects:

int **PyUnicode_Check**(PyObject **o*)

Return true if the object *o* is a Unicode object or an instance of a Unicode subtype.

int **PyUnicode_CheckExact**(PyObject **o*)

Return true if the object *o* is a Unicode object, but not an instance of a subtype.

int **PyUnicode_READY**(PyObject **o*)

Ensure the string object *o* is in the "canonical" representation. This is required before using any of the access macros described below.

Returns `0` on success and `-1` with an exception set on failure, which in particular happens if memory allocation fails.

New in version 3.3.

Py_ssize_t **PyUnicode_GET_LENGTH**(PyObject **o*)

Return the length of the Unicode string, in code points. *o* has to be a Unicode object in the "canonical" representation (not checked).

New in version 3.3.

Py_UCS1* **PyUnicode_1BYTE_DATA**(PyObject **o*)
Py_UCS2* **PyUnicode_2BYTE_DATA**(PyObject **o*)
Py_UCS4* **PyUnicode_4BYTE_DATA**(PyObject **o*)

Return a pointer to the canonical representation cast to UCS1, UCS2 or UCS4 integer types for direct character access. No checks are performed if the canonical representation has the correct character size; use `PyUnicode_KIND()` to select the right macro. Make sure `PyUnicode_READY()` has been called before accessing this.

New in version 3.3.

`PyUnicode_WCHAR_KIND`
`PyUnicode_1BYTE_KIND`
`PyUnicode_2BYTE_KIND`
`PyUnicode_4BYTE_KIND`

Return values of the `PyUnicode_KIND()` macro.

New in version 3.3.

int **`PyUnicode_KIND`**(PyObject **o*)

Return one of the PyUnicode kind constants (see above) that indicate how many bytes per character this Unicode object uses to store its data. *o* has to be a Unicode object in the "canonical" representation (not checked).

New in version 3.3.

void* **`PyUnicode_DATA`**(PyObject **o*)

Return a void pointer to the raw unicode buffer. *o* has to be a Unicode object in the "canonical" representation (not checked).

New in version 3.3.

void **`PyUnicode_WRITE`**(int *kind*, void **data*, Py_ssize_t *index*, Py_UCS4 *value*)

Write into a canonical representation *data* (as obtained with `PyUnicode_DATA()`). This macro does not do any sanity checks and is intended for usage in loops. The caller should cache the *kind* value and *data* pointer as obtained from other macro calls. *index* is the index in the string (starts at 0) and *value* is the new code point value which should be written to that location.

New in version 3.3.

Py_UCS4 **`PyUnicode_READ`**(int *kind*, void **data*, Py_ssize_t *index*)

Read a code point from a canonical representation *data* (as obtained with `PyUnicode_DATA()`). No checks or ready calls are performed.

New in version 3.3.

Py_UCS4 **`PyUnicode_READ_CHAR`**(PyObject **o*, Py_ssize_t *index*)

Read a character from a Unicode object *o*, which must be in the "canonical" representation. This is less efficient than `PyUnicode_READ()` if you do multiple consecutive reads.

New in version 3.3.

`PyUnicode_MAX_CHAR_VALUE`(PyObject **o*)

Return the maximum code point that is suitable for creating another string based on *o*, which must be in the "canonical" representation. This is always an approximation but more efficient than iterating over the string.

New in version 3.3.

int **`PyUnicode_ClearFreeList`**()

Clear the free list. Return the total number of freed items.

Py_ssize_t **`PyUnicode_GET_SIZE`**(PyObject **o*)

Return the size of the deprecated `Py_UNICODE` representation, in code units (this includes surrogate pairs as 2 units). *o* has to be a Unicode object (not checked).

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style Unicode API, please migrate to using `PyUnicode_GET_LENGTH()`.

Py_ssize_t **`PyUnicode_GET_DATA_SIZE`**(PyObject **o*)

Return the size of the deprecated `Py_UNICODE` representation in bytes. *o* has to be a Unicode object (not checked).

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style Unicode API, please migrate to using `PyUnicode_GET_LENGTH()`.

Py_UNICODE* **`PyUnicode_AS_UNICODE`**(PyObject **o*)
const char* **`PyUnicode_AS_DATA`**(PyObject **o*)

Return a pointer to a `Py_UNICODE` representation of the object. The returned buffer is always terminated with an extra null code point. It may also contain embedded null code points, which would cause the string to be truncated when used in most C functions. The `AS_DATA` form casts the pointer to `const char *`. The *o* argument has to be a Unicode object (not checked).

Changed in version 3.3: This macro is now inefficient – because in many cases the `Py_UNICODE` representation does not exist and needs to be created – and can fail (return *NULL* with an exception set). Try to port the code to use the new `PyUnicode_nBYTE_DATA()` macros or use `PyUnicode_WRITE()` or `PyUnicode_READ()`.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style Unicode API, please migrate to using the `PyUnicode_nBYTE_DATA()` family of macros.

Unicode Character Properties

Unicode provides many different character properties. The most often needed ones are available through these macros which are mapped to C functions depending on the Python configuration.

int **`Py_UNICODE_ISSPACE`**(Py_UNICODE *ch*)
Return `1` or `0` depending on whether *ch* is a whitespace character.

int **`Py_UNICODE_ISLOWER`**(Py_UNICODE *ch*)
Return `1` or `0` depending on whether *ch* is a lowercase character.

int **`Py_UNICODE_ISUPPER`**(Py_UNICODE *ch*)
Return `1` or `0` depending on whether *ch* is an uppercase character.

int **`Py_UNICODE_ISTITLE`**(Py_UNICODE *ch*)
Return `1` or `0` depending on whether *ch* is a titlecase character.

int **`Py_UNICODE_ISLINEBREAK`**(Py_UNICODE *ch*)
Return `1` or `0` depending on whether *ch* is a linebreak character.

int **`Py_UNICODE_ISDECIMAL`**(Py_UNICODE *ch*)
Return `1` or `0` depending on whether *ch* is a decimal character.

int **`Py_UNICODE_ISDIGIT`**(Py_UNICODE *ch*)
Return `1` or `0` depending on whether *ch* is a digit character.

int **`Py_UNICODE_ISNUMERIC`**(Py_UNICODE *ch*)
Return `1` or `0` depending on whether *ch* is a numeric character.

int **`Py_UNICODE_ISALPHA`**(Py_UNICODE *ch*)
Return `1` or `0` depending on whether *ch* is an alphabetic character.

int **`Py_UNICODE_ISALNUM`**(Py_UNICODE *ch*)
Return `1` or `0` depending on whether *ch* is an alphanumeric character.

int **`Py_UNICODE_ISPRINTABLE`**(Py_UNICODE *ch*)
Return `1` or `0` depending on whether *ch* is a printable character. Nonprintable characters are those characters defined in the Unicode character database as "Other" or "Separator", excepting the ASCII space (0x20) which is considered

printable. (Note that printable characters in this context are those which should not be escaped when `repr()` is invoked on a string. It has no bearing on the handling of strings written to `sys.stdout` or `sys.stderr`.)

These APIs can be used for fast direct character conversions:

Py_UNICODE **`Py_UNICODE_TOLOWER`**(Py_UNICODE *ch*)

Return the character *ch* converted to lower case.

Deprecated since version 3.3: This function uses simple case mappings.

Py_UNICODE **`Py_UNICODE_TOUPPER`**(Py_UNICODE *ch*)

Return the character *ch* converted to upper case.

Deprecated since version 3.3: This function uses simple case mappings.

Py_UNICODE **`Py_UNICODE_TOTITLE`**(Py_UNICODE *ch*)

Return the character *ch* converted to title case.

Deprecated since version 3.3: This function uses simple case mappings.

int **`Py_UNICODE_TODECIMAL`**(Py_UNICODE *ch*)

Return the character *ch* converted to a decimal positive integer. Return `-1` if this is not possible. This macro does not raise exceptions.

int **`Py_UNICODE_TODIGIT`**(Py_UNICODE *ch*)

Return the character *ch* converted to a single digit integer. Return `-1` if this is not possible. This macro does not raise exceptions.

double **`Py_UNICODE_TONUMERIC`**(Py_UNICODE *ch*)

Return the character *ch* converted to a double. Return `-1.0` if this is not possible. This macro does not raise exceptions.

These APIs can be used to work with surrogates:

`Py_UNICODE_IS_SURROGATE`(ch)

Check if *ch* is a surrogate (`0xD800 <= ch <= 0xDFFF`).

`Py_UNICODE_IS_HIGH_SURROGATE`(ch)

Check if *ch* is a high surrogate (`0xD800 <= ch <= 0xDBFF`).

`Py_UNICODE_IS_LOW_SURROGATE`(ch)

Check if *ch* is a low surrogate (`0xDC00 <= ch <= 0xDFFF`).

`Py_UNICODE_JOIN_SURROGATES`(high, low)

Join two surrogate characters and return a single Py_UCS4 value. *high* and *low* are respectively the leading and trailing surrogates in a surrogate pair.

Creating and accessing Unicode strings

To create Unicode objects and access their basic sequence properties, use these APIs:

PyObject* **`PyUnicode_New`**(Py_ssize_t *size*, Py_UCS4 *maxchar*)

Create a new Unicode object. *maxchar* should be the true maximum code point to be placed in the string. As an approximation, it can be rounded up to the nearest value in the sequence 127, 255, 65535, 1114111.

This is the recommended way to allocate a new Unicode object. Objects created using this function are not resizable.

New in version 3.3.

PyObject* **`PyUnicode_FromKindAndData`**(int *kind*, const void **buffer*, Py_ssize_t *size*)

Create a new Unicode object with the given *kind* (possible values are `PyUnicode_1BYTE_KIND` etc., as returned by `PyUnicode_KIND()`). The *buffer* must point to an array of *size* units of 1, 2 or 4 bytes per character, as given by the kind.

New in version 3.3.

PyObject* **`PyUnicode_FromStringAndSize`**(const char **u*, Py_ssize_t *size*)

Create a Unicode object from the char buffer *u*. The bytes will be interpreted as being UTF-8 encoded. The buffer is copied into the new object. If the buffer is not *NULL*, the return value might be a shared object, i.e. modification of the data is not allowed.

If *u* is *NULL*, this function behaves like `PyUnicode_FromUnicode()` with the buffer set to *NULL*. This usage is deprecated in favor of `PyUnicode_New()`.

PyObject ***`PyUnicode_FromString`**(const char **u*)

Create a Unicode object from a UTF-8 encoded null-terminated char buffer *u*.

PyObject* **`PyUnicode_FromFormat`**(const char **format*, ...)

Take a C `printf()`-style *format* string and a variable number of arguments, calculate the size of the resulting Python unicode string and return a string with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the *format* ASCII-encoded string. The following format characters are allowed:

Format Characters	**Type**	**Comment**
`%%`	*n/a*	The literal % character.
`%c`	int	A single character, represented as a C int.
`%d`	int	Exactly equivalent to `printf("%d")`.
`%u`	unsigned int	Exactly equivalent to `printf("%u")`.
`%ld`	long	Exactly equivalent to `printf("%ld")`.
`%li`	long	Exactly equivalent to `printf("%li")`.
`%lu`	unsigned long	Exactly equivalent to `printf("%lu")`.
`%lld`	long long	Exactly equivalent to `printf("%lld")`.
`%lli`	long long	Exactly equivalent to `printf("%lli")`.
`%llu`	unsigned long long	Exactly equivalent to `printf("%llu")`.
`%zd`	Py_ssize_t	Exactly equivalent to `printf("%zd")`.
`%zi`	Py_ssize_t	Exactly equivalent to `printf("%zi")`.
`%zu`	size_t	Exactly equivalent to `printf("%zu")`.
`%i`	int	Exactly equivalent to `printf("%i")`.
`%x`	int	Exactly equivalent to `printf("%x")`.
`%s`	char*	A null-terminated C character array.

Format Characters	Type	Comment
%p	void*	The hex representation of a C pointer. Mostly equivalent to `printf("%p")` except that it is guaranteed to start with the literal `0x` regardless of what the platform's `printf` yields.
%A	PyObject*	The result of calling `ascii()`.
%U	PyObject*	A unicode object.
%V	PyObject*, char *	A unicode object (which may be *NULL*) and a null-terminated C character array as a second parameter (which will be used, if the first parameter is *NULL*).
%S	PyObject*	The result of calling `PyObject_Str()`.
%R	PyObject*	The result of calling `PyObject_Repr()`.

An unrecognized format character causes all the rest of the format string to be copied as-is to the result string, and any extra arguments discarded.

Note: The width formatter unit is number of characters rather than bytes. The precision formatter unit is number of bytes for `"%s"` and `"%V"` (if the `PyObject*` argument is NULL), and a number of characters for `"%A"`, `"%U"`, `"%S"`, `"%R"` and `"%V"` (if the `PyObject*` argument is not NULL).

Changed in version 3.2: Support for `"%lld"` and `"%llu"` added.

Changed in version 3.3: Support for `"%li"`, `"%lli"` and `"%zi"` added.

Changed in version 3.4: Support width and precision formatter for `"%s"`, `"%A"`, `"%U"`, `"%V"`, `"%S"`, `"%R"` added.

PyObject* **`PyUnicode_FromFormatV`**(const char **format*, va_list *vargs*)

Identical to `PyUnicode_FromFormat()` except that it takes exactly two arguments.

PyObject* **`PyUnicode_FromEncodedObject`**(PyObject **obj*, const char **encoding*, const char **errors*)

Return value: New reference.

Decode an encoded object *obj* to a Unicode object.

`bytes`, `bytearray` and other bytes-like objects are decoded according to the given *encoding* and using the error handling defined by *errors*. Both can be *NULL* to have the interface use the default values (see Built-in Codecs for details).

All other objects, including Unicode objects, cause a `TypeError` to be set.

The API returns *NULL* if there was an error. The caller is responsible for decref'ing the returned objects.

Py_ssize_t **`PyUnicode_GetLength`**(PyObject **unicode*)

Return the length of the Unicode object, in code points.

New in version 3.3.

Py_ssize_t **`PyUnicode_CopyCharacters`**(PyObject **to*, Py_ssize_t *to_start*, PyObject **from*, Py_ssize_t *from_start*, Py_ssize_t *how_many*)

Copy characters from one Unicode object into another. This function performs character conversion when necessary and falls back to `memcpy()` if possible. Returns `-1` and sets an exception on error, otherwise returns the number of copied characters.

New in version 3.3.

Py_ssize_t **`PyUnicode_Fill`**(PyObject **unicode*, Py_ssize_t *start*, Py_ssize_t *length*, Py_UCS4 *fill_char*)

Fill a string with a character: write *fill_char* into `unicode[start:start+length]`.

Fail if *fill_char* is bigger than the string maximum character, or if the string has more than 1 reference.

Return the number of written character, or return `-1` and raise an exception on error.

New in version 3.3.

int **`PyUnicode_WriteChar`**(PyObject **unicode*, Py_ssize_t *index*, Py_UCS4 *character*)

Write a character to a string. The string must have been created through `PyUnicode_New()`. Since Unicode strings are supposed to be immutable, the string must not be shared, or have been hashed yet.

This function checks that *unicode* is a Unicode object, that the index is not out of bounds, and that the object can be modified safely (i.e. that it its reference count is one).

New in version 3.3.

Py_UCS4 **`PyUnicode_ReadChar`**(PyObject **unicode*, Py_ssize_t *index*)

Read a character from a string. This function checks that *unicode* is a Unicode object and the index is not out of bounds, in contrast to the macro version `PyUnicode_READ_CHAR()`.

New in version 3.3.

PyObject* **`PyUnicode_Substring`**(PyObject **str*, Py_ssize_t *start*, Py_ssize_t *end*)

Return a substring of *str*, from character index *start* (included) to character index *end* (excluded). Negative indices are not supported.

New in version 3.3.

Py_UCS4* **`PyUnicode_AsUCS4`**(PyObject **u*, Py_UCS4 **buffer*, Py_ssize_t *buflen*, int *copy_null*)

Copy the string *u* into a UCS4 buffer, including a null character, if *copy_null* is set. Returns *NULL* and sets an exception on error (in particular, a `SystemError` if *buflen* is smaller than the length of *u*). *buffer* is returned on success.

New in version 3.3.

Py_UCS4* **`PyUnicode_AsUCS4Copy`**(PyObject **u*)

Copy the string *u* into a new UCS4 buffer that is allocated using `PyMem_Malloc()`. If this fails, *NULL* is returned with a `MemoryError` set. The returned buffer always has an extra null code point appended.

New in version 3.3.

Deprecated Py_UNICODE APIs

Deprecated since version 3.3, will be removed in version 4.0.

These API functions are deprecated with the implementation of **PEP 393**. Extension modules can continue using them, as they will not be removed in Python 3.x, but need to be aware that their use can now cause performance and memory hits.

PyObject* **`PyUnicode_FromUnicode`**(const Py_UNICODE **u*, Py_ssize_t *size*)

Return value: New reference.

Create a Unicode object from the Py_UNICODE buffer *u* of the given size. *u* may be *NULL* which causes the contents to be undefined. It is the user's responsibility to fill in the needed data. The buffer is copied into the new object.

If the buffer is not *NULL*, the return value might be a shared object. Therefore, modification of the resulting Unicode object is only allowed when *u* is *NULL*.

If the buffer is *NULL*, `PyUnicode_READY()` must be called once the string content has been filled before using any of the access macros such as `PyUnicode_KIND()`.

Please migrate to using `PyUnicode_FromKindAndData()`, `PyUnicode_FromWideChar()` or `PyUnicode_New()`.

Py_UNICODE* **PyUnicode_AsUnicode**(PyObject **unicode*)

Return a read-only pointer to the Unicode object's internal `Py_UNICODE` buffer, or *NULL* on error. This will create the `Py_UNICODE*` representation of the object if it is not yet available. The buffer is always terminated with an extra null code point. Note that the resulting `Py_UNICODE` string may also contain embedded null code points, which would cause the string to be truncated when used in most C functions.

Please migrate to using `PyUnicode_AsUCS4()`, `PyUnicode_AsWideChar()`, `PyUnicode_ReadChar()` or similar new APIs.

PyObject* **PyUnicode_TransformDecimalToASCII**(Py_UNICODE **s*, Py_ssize_t *size*)

Create a Unicode object by replacing all decimal digits in `Py_UNICODE` buffer of the given *size* by ASCII digits 0–9 according to their decimal value. Return *NULL* if an exception occurs.

Py_UNICODE* **PyUnicode_AsUnicodeAndSize**(PyObject **unicode*, Py_ssize_t **size*)

Like `PyUnicode_AsUnicode()`, but also saves the `Py_UNICODE()` array length (excluding the extra null terminator) in *size*. Note that the resulting `Py_UNICODE*` string may contain embedded null code points, which would cause the string to be truncated when used in most C functions.

New in version 3.3.

Py_UNICODE* **PyUnicode_AsUnicodeCopy**(PyObject **unicode*)

Create a copy of a Unicode string ending with a null code point. Return *NULL* and raise a `MemoryError` exception on memory allocation failure, otherwise return a new allocated buffer (use `PyMem_Free()` to free the buffer). Note that the resulting `Py_UNICODE*` string may contain embedded null code points, which would cause the string to be truncated when used in most C functions.

New in version 3.2.

Please migrate to using `PyUnicode_AsUCS4Copy()` or similar new APIs.

Py_ssize_t **`PyUnicode_GetSize`**(PyObject **unicode*)

Return the size of the deprecated `Py_UNICODE` representation, in code units (this includes surrogate pairs as 2 units).

Please migrate to using `PyUnicode_GetLength()`.

PyObject* **`PyUnicode_FromObject`**(PyObject **obj*)

Return value: New reference.

Copy an instance of a Unicode subtype to a new true Unicode object if necessary. If *obj* is already a true Unicode object (not a subtype), return the reference with incremented refcount.

Objects other than Unicode or its subtypes will cause a `TypeError`.

Locale Encoding

The current locale encoding can be used to decode text from the operating system.

PyObject* **`PyUnicode_DecodeLocaleAndSize`**(const char **str*, Py_ssize_t *len*, const char **errors*)

Decode a string from the current locale encoding. The supported error handlers are `"strict"` and `"surrogateescape"` (**PEP 383**). The decoder uses `"strict"` error handler if *errors* is `NULL`. *str* must end with a null character but cannot contain embedded null characters.

Use `PyUnicode_DecodeFSDefaultAndSize()` to decode a string from `Py_FileSystemDefaultEncoding` (the locale encoding read at Python startup).

See also: The `Py_DecodeLocale()` function.

New in version 3.3.

Changed in version 3.6.5: The function now also uses the current locale encoding for the `surrogateescape` error handler. Previously, `Py_DecodeLocale()` was used for the `surrogateescape`, and the current locale encoding was used for `strict`.

PyObject* **`PyUnicode_DecodeLocale`**(const char **str*, const char **errors*)

Similar to `PyUnicode_DecodeLocaleAndSize()`, but compute the string length using `strlen()`.

New in version 3.3.

PyObject* **`PyUnicode_EncodeLocale`**(PyObject **unicode*, const char **errors*)

Encode a Unicode object to the current locale encoding. The supported error handlers are `"strict"` and `"surrogateescape"` (**PEP 383**). The encoder uses `"strict"` error handler if *errors* is `NULL`. Return a `bytes` object. *unicode* cannot contain embedded null characters.

Use `PyUnicode_EncodeFSDefault()` to encode a string to `Py_FileSystemDefaultEncoding` (the locale encoding read at Python startup).

See also: The `Py_EncodeLocale()` function.

New in version 3.3.

Changed in version 3.6.5: The function now also uses the current locale encoding for the `surrogateescape` error handler. Previously, `Py_EncodeLocale()` was used for the `surrogateescape`, and the current locale encoding was used for `strict`.

File System Encoding

To encode and decode file names and other environment strings, `Py_FileSystemDefaultEncoding` should be used as the encoding, and `Py_FileSystemDefaultEncodeErrors` should be used as the error handler (**PEP 383** and **PEP 529**). To encode file names to `bytes` during argument parsing, the `"O&"` converter should be used, passing `PyUnicode_FSConverter()` as the conversion function:

int **`PyUnicode_FSConverter`**(PyObject* *obj*, void* *result*)

ParseTuple converter: encode `str` objects – obtained directly or through the `os.PathLike` interface – to `bytes` using `PyUnicode_EncodeFSDefault()`; `bytes` objects are output as-is. *result* must be a `PyBytesObject*` which must be released when it is no longer used.

New in version 3.1.

Changed in version 3.6: Accepts a path-like object.

To decode file names to `str` during argument parsing, the `"O&"` converter should be used, passing `PyUnicode_FSDecoder()` as the conversion function:

int **PyUnicode_FSDecoder**(PyObject* *obj*, void* *result*)

ParseTuple converter: decode `bytes` objects – obtained either directly or indirectly through the `os.PathLike` interface – to `str` using `PyUnicode_DecodeFSDefaultAndSize()`; `str` objects are output as-is. *result* must be a `PyUnicodeObject*` which must be released when it is no longer used.

New in version 3.2.

Changed in version 3.6: Accepts a path-like object.

PyObject* **PyUnicode_DecodeFSDefaultAndSize**(const char **s*, Py_ssize_t *size*)

Decode a string using `Py_FileSystemDefaultEncoding` and the `Py_FileSystemDefaultEncodeErrors` error handler.

If `Py_FileSystemDefaultEncoding` is not set, fall back to the locale encoding.

`Py_FileSystemDefaultEncoding` is initialized at startup from the locale encoding and cannot be modified later. If you need to decode a string from the current locale encoding, use `PyUnicode_DecodeLocaleAndSize()`.

See also: The `Py_DecodeLocale()` function.

Changed in version 3.6: Use `Py_FileSystemDefaultEncodeErrors` error handler.

PyObject* **PyUnicode_DecodeFSDefault**(const char **s*)

Decode a null-terminated string using `Py_FileSystemDefaultEncoding` and the `Py_FileSystemDefaultEncodeErrors` error handler.

If `Py_FileSystemDefaultEncoding` is not set, fall back to the locale encoding.

Use `PyUnicode_DecodeFSDefaultAndSize()` if you know the string length.

Changed in version 3.6: Use `Py_FileSystemDefaultEncodeErrors` error handler.

PyObject* **PyUnicode_EncodeFSDefault**(PyObject **unicode*)

Encode a Unicode object to `Py_FileSystemDefaultEncoding` with the `Py_FileSystemDefaultEncodeErrors` error handler, and return `bytes`. Note that the resulting `bytes` object may contain null bytes.

If `Py_FileSystemDefaultEncoding` is not set, fall back to the locale encoding.

`Py_FileSystemDefaultEncoding` is initialized at startup from the locale encoding and cannot be modified later. If you need to encode a string to the current locale encoding, use `PyUnicode_EncodeLocale()`.

See also: The `Py_EncodeLocale()` function.

New in version 3.2.

Changed in version 3.6: Use `Py_FileSystemDefaultEncodeErrors` error handler.

wchar_t Support

`wchar_t` support for platforms which support it:

PyObject* **`PyUnicode_FromWideChar`**(const wchar_t **w*, Py_ssize_t *size*)

Return value: New reference.

Create a Unicode object from the `wchar_t` buffer *w* of the given *size*. Passing `-1` as the *size* indicates that the function must itself compute the length, using wcslen. Return *NULL* on failure.

Py_ssize_t **`PyUnicode_AsWideChar`**(PyUnicodeObject **unicode*, wchar_t **w*, Py_ssize_t *size*)

Copy the Unicode object contents into the `wchar_t` buffer *w*. At most *size* `wchar_t` characters are copied (excluding a possibly trailing null termination character). Return the number of `wchar_t` characters copied or `-1` in case of an error. Note that the resulting `wchar_t*` string may or may not be null-terminated. It is the responsibility of the caller to make sure that the `wchar_t*` string is null-terminated in case this is required by the application. Also, note that the `wchar_t*` string might contain null characters, which would cause the string to be truncated when used with most C functions.

wchar_t* **`PyUnicode_AsWideCharString`**(PyObject **unicode*, Py_ssize_t **size*)

Convert the Unicode object to a wide character string. The output string always ends with a null character. If *size* is not *NULL*, write the number of wide characters (excluding the trailing null termination character) into **size*.

Returns a buffer allocated by `PyMem_Alloc()` (use `PyMem_Free()` to free it) on success. On error, returns *NULL*, **size* is undefined and raises a `MemoryError`. Note that the resulting `wchar_t` string might contain null characters, which would cause the string to be truncated when used with most C functions.

New in version 3.2.

Built-in Codecs

Python provides a set of built-in codecs which are written in C for speed. All of these codecs are directly usable via the following functions.

Many of the following APIs take two arguments encoding and errors, and they have the same semantics as the ones of the built-in `str()` string object constructor.

Setting encoding to *NULL* causes the default encoding to be used which is ASCII. The file system calls should use `PyUnicode_FSConverter()` for encoding file names. This uses the variable `Py_FileSystemDefaultEncoding` internally. This variable should be treated as read-only: on some systems, it will be a pointer to a static string, on others, it will change at run-time (such as when the application invokes setlocale).

Error handling is set by errors which may also be set to *NULL* meaning to use the default handling defined for the codec. Default error handling for all built-in codecs is "strict" (`ValueError` is raised).

The codecs all use a similar interface. Only deviation from the following generic ones are documented for simplicity.

Generic Codecs

These are the generic codec APIs:

PyObject* **`PyUnicode_Decode`**(const char **s*, Py_ssize_t *size*, const char **encoding*, const char **errors*)

> *Return value: New reference.*
> Create a Unicode object by decoding *size* bytes of the encoded string *s*. *encoding* and *errors* have the same meaning as the parameters of the same name in the `str()` built-in function. The codec to be used is looked up using the Python codec registry. Return *NULL* if an exception was raised by the codec.

PyObject* **`PyUnicode_AsEncodedString`**(PyObject **unicode*, const char **encoding*, const char **errors*)

> *Return value: New reference.*
> Encode a Unicode object and return the result as Python bytes object. *encoding* and *errors* have the same meaning as the parameters of the same name in the Unicode `encode()` method. The codec to be used is looked up using the Python codec registry. Return *NULL* if an exception was raised by the codec.

PyObject* **`PyUnicode_Encode`**(const Py_UNICODE **s*, Py_ssize_t *size*, const char **encoding*, const char **errors*)

Return value: New reference.
Encode the `Py_UNICODE` buffer *s* of the given *size* and return a Python bytes object. *encoding* and *errors* have the same meaning as the parameters of the same name in the Unicode `encode()` method. The codec to be used is looked up using the Python codec registry. Return *NULL* if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsEncodedString()`.

UTF-8 Codecs

These are the UTF-8 codec APIs:

PyObject* **PyUnicode_DecodeUTF8**(const char **s*, Py_ssize_t *size*, const char **errors*)

Return value: New reference.
Create a Unicode object by decoding *size* bytes of the UTF-8 encoded string *s*. Return *NULL* if an exception was raised by the codec.

PyObject* **PyUnicode_DecodeUTF8Stateful**(const char **s*, Py_ssize_t *size*, const char **errors*, Py_ssize_t **consumed*)

Return value: New reference.
If *consumed* is *NULL*, behave like `PyUnicode_DecodeUTF8()`. If *consumed* is not *NULL*, trailing incomplete UTF-8 byte sequences will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject* **PyUnicode_AsUTF8String**(PyObject **unicode*)

Return value: New reference.
Encode a Unicode object using UTF-8 and return the result as Python bytes object. Error handling is “strict”. Return *NULL* if an exception was raised by the codec.

char* **PyUnicode_AsUTF8AndSize**(PyObject **unicode*, Py_ssize_t **size*)

Return a pointer to the UTF-8 encoding of the Unicode object, and store the size of the encoded representation (in bytes) in *size*. The *size* argument can be *NULL*; in this case no size will be stored. The returned buffer always has an extra null byte appended (not included in *size*), regardless of whether there are any other null code points.

In the case of an error, *NULL* is returned with an exception set and no *size* is stored.

This caches the UTF-8 representation of the string in the Unicode object, and subsequent calls will return a pointer to the same buffer. The caller is not responsible for deallocating the buffer.

New in version 3.3.

char* **PyUnicode_AsUTF8**(PyObject **unicode*)

As `PyUnicode_AsUTF8AndSize()`, but does not store the size.

New in version 3.3.

PyObject* **PyUnicode_EncodeUTF8**(const Py_UNICODE **s*, Py_ssize_t *size*, const char **errors*)

Return value: New reference.

Encode the `Py_UNICODE` buffer *s* of the given *size* using UTF-8 and return a Python bytes object. Return *NULL* if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsUTF8String()`, `PyUnicode_AsUTF8AndSize()` or `PyUnicode_AsEncodedString()`.

UTF-32 Codecs

These are the UTF-32 codec APIs:

PyObject* **PyUnicode_DecodeUTF32**(const char **s*, Py_ssize_t *size*, const char **errors*, int **byteorder*)

Decode *size* bytes from a UTF-32 encoded buffer string and return the corresponding Unicode object. *errors* (if non-*NULL*) defines the error handling. It defaults to “strict”.

If *byteorder* is non-*NULL*, the decoder starts decoding using the given byte order:

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

If `*byteorder` is zero, and the first four bytes of the input data are a byte order mark (BOM), the decoder switches to this byte order and the BOM is not copied into the resulting Unicode string. If `*byteorder` is `-1` or `1`, any byte order mark is copied to the output.

After completion, **byteorder* is set to the current byte order at the end of input data.

If *byteorder* is *NULL*, the codec starts in native order mode.

Return *NULL* if an exception was raised by the codec.

PyObject* **`PyUnicode_DecodeUTF32Stateful`**(const char **s*, Py_ssize_t *size*, const char **errors*, int **byteorder*, Py_ssize_t **consumed*)

If *consumed* is *NULL*, behave like `PyUnicode_DecodeUTF32()`. If *consumed* is not *NULL*, `PyUnicode_DecodeUTF32Stateful()` will not treat trailing incomplete UTF-32 byte sequences (such as a number of bytes not divisible by four) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject* **`PyUnicode_AsUTF32String`**(PyObject **unicode*)

Return a Python byte string using the UTF-32 encoding in native byte order. The string always starts with a BOM mark. Error handling is “strict”. Return *NULL* if an exception was raised by the codec.

PyObject* **`PyUnicode_EncodeUTF32`**(const Py_UNICODE **s*, Py_ssize_t *size*, const char **errors*, int *byteorder*)

Return a Python bytes object holding the UTF-32 encoded value of the Unicode data in *s*. Output is written according to the following byte order:

```
byteorder == -1: little endian
byteorder == 0:  native byte order (writes a BOM mark)
byteorder == 1:  big endian
```

If byteorder is `0`, the output string will always start with the Unicode BOM mark (U+FEFF). In the other two modes, no BOM mark is prepended.

If *Py_UNICODE_WIDE* is not defined, surrogate pairs will be output as a single code point.

Return *NULL* if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsUTF32String()` or `PyUnicode_AsEncodedString()`.

UTF-16 Codecs

These are the UTF-16 codec APIs:

PyObject* **`PyUnicode_DecodeUTF16`**(const char **s*, Py_ssize_t *size*, const char **errors*, int **byteorder*)

Return value: New reference.

Decode *size* bytes from a UTF-16 encoded buffer string and return the corresponding Unicode object. *errors* (if non-*NULL*) defines the error handling. It defaults to “strict”.

If *byteorder* is non-*NULL*, the decoder starts decoding using the given byte order:

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

If `*byteorder` is zero, and the first two bytes of the input data are a byte order mark (BOM), the decoder switches to this byte order and the BOM is not copied into the resulting Unicode string. If `*byteorder` is `-1` or `1`, any byte order mark is copied to the output (where it will result in either a `\ufeff` or a `\ufffe` character).

After completion, **byteorder* is set to the current byte order at the end of input data.

If *byteorder* is *NULL*, the codec starts in native order mode.

Return *NULL* if an exception was raised by the codec.

PyObject* **`PyUnicode_DecodeUTF16Stateful`**(const char **s*, Py_ssize_t *size*, const char **errors*, int **byteorder*, Py_ssize_t **consumed*)

Return value: New reference.

If *consumed* is *NULL*, behave like `PyUnicode_DecodeUTF16()`. If *consumed* is not *NULL*, `PyUnicode_DecodeUTF16Stateful()` will not treat trailing incomplete UTF-16 byte sequences (such as an odd number of bytes or a split surrogate pair) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject* **`PyUnicode_AsUTF16String`**(PyObject **unicode*)

Return value: New reference.

Return a Python byte string using the UTF-16 encoding in native byte order. The string always starts with a BOM mark. Error handling is “strict”. Return *NULL* if an exception was raised by the codec.

PyObject* **`PyUnicode_EncodeUTF16`**(const Py_UNICODE **s*, Py_ssize_t *size*, const char **errors*, int *byteorder*)

Return value: New reference.

Return a Python bytes object holding the UTF-16 encoded value of the Unicode data in *s*. Output is written according to the following byte order:

```
byteorder == -1: little endian
byteorder == 0:  native byte order (writes a BOM mark)
byteorder == 1:  big endian
```

If byteorder is `0`, the output string will always start with the Unicode BOM mark (U+FEFF). In the other two modes, no BOM mark is prepended.

If *Py_UNICODE_WIDE* is defined, a single `Py_UNICODE` value may get represented as a surrogate pair. If it is not defined, each `Py_UNICODE` values is interpreted as a UCS-2 character.

Return *NULL* if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsUTF16String()` or `PyUnicode_AsEncodedString()`.

UTF-7 Codecs

These are the UTF-7 codec APIs:

PyObject* **PyUnicode_DecodeUTF7**(const char **s*, Py_ssize_t *size*, const char **errors*)

Create a Unicode object by decoding *size* bytes of the UTF-7 encoded string *s*. Return *NULL* if an exception was raised by the codec.

PyObject* **PyUnicode_DecodeUTF7Stateful**(const char **s*, Py_ssize_t *size*, const char **errors*, Py_ssize_t **consumed*)

If *consumed* is *NULL*, behave like `PyUnicode_DecodeUTF7()`. If *consumed* is not *NULL*, trailing incomplete UTF-7 base-64 sections will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject* **PyUnicode_EncodeUTF7**(const Py_UNICODE **s*, Py_ssize_t *size*, int *base64SetO*, int *base64WhiteSpace*, const char **errors*)

Encode the `Py_UNICODE` buffer of the given size using UTF-7 and return a Python bytes object. Return *NULL* if an exception was raised by the codec.

If *base64SetO* is nonzero, “Set O” (punctuation that has no otherwise special meaning) will be encoded in base-64. If *base64WhiteSpace* is nonzero, whitespace will be encoded in base-64. Both are set to zero for the Python “utf-7” codec.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsEncodedString()`.

Unicode-Escape Codecs

These are the “Unicode Escape” codec APIs:

PyObject* **`PyUnicode_DecodeUnicodeEscape`**(const char *s, Py_ssize_t *size*, const char **errors*)

Return value: New reference.
Create a Unicode object by decoding *size* bytes of the Unicode-Escape encoded string *s*. Return *NULL* if an exception was raised by the codec.

PyObject* **`PyUnicode_AsUnicodeEscapeString`**(PyObject **unicode*)

Return value: New reference.
Encode a Unicode object using Unicode-Escape and return the result as a bytes object. Error handling is “strict”. Return *NULL* if an exception was raised by the codec.

PyObject* **`PyUnicode_EncodeUnicodeEscape`**(const Py_UNICODE **s*, Py_ssize_t *size*)

Return value: New reference.
Encode the `Py_UNICODE` buffer of the given *size* using Unicode-Escape and return a bytes object. Return *NULL* if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsUnicodeEscapeString()`.

Raw-Unicode-Escape Codecs

These are the “Raw Unicode Escape” codec APIs:

PyObject* **`PyUnicode_DecodeRawUnicodeEscape`**(const char **s*, Py_ssize_t *size*, const char **errors*)

Return value: New reference.
Create a Unicode object by decoding *size* bytes of the Raw-Unicode-Escape encoded string *s*. Return *NULL* if an exception was raised by the codec.

PyObject* **`PyUnicode_AsRawUnicodeEscapeString`**(PyObject **unicode*)

Return value: New reference.

Encode a Unicode object using Raw-Unicode-Escape and return the result as a bytes object. Error handling is “strict”. Return *NULL* if an exception was raised by the codec.

PyObject* **`PyUnicode_EncodeRawUnicodeEscape`**(const Py_UNICODE **s*, Py_ssize_t *size*, const char **errors*)

Return value: New reference.

Encode the `Py_UNICODE` buffer of the given *size* using Raw-Unicode-Escape and return a bytes object. Return *NULL* if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsRawUnicodeEscapeString()` or `PyUnicode_AsEncodedString()`.

Latin-1 Codecs

These are the Latin-1 codec APIs: Latin-1 corresponds to the first 256 Unicode ordinals and only these are accepted by the codecs during encoding.

PyObject* **`PyUnicode_DecodeLatin1`**(const char **s*, Py_ssize_t *size*, const char **errors*)

Return value: New reference.

Create a Unicode object by decoding *size* bytes of the Latin-1 encoded string *s*. Return *NULL* if an exception was raised by the codec.

PyObject* **`PyUnicode_AsLatin1String`**(PyObject **unicode*)

Return value: New reference.

Encode a Unicode object using Latin-1 and return the result as Python bytes object. Error handling is “strict”. Return *NULL* if an exception was raised by the codec.

PyObject* **`PyUnicode_EncodeLatin1`**(const Py_UNICODE **s*, Py_ssize_t *size*, const char **errors*)

Return value: New reference.

Encode the `Py_UNICODE` buffer of the given *size* using Latin-1 and return a Python bytes object. Return *NULL* if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsLatin1String()` or `PyUnicode_AsEncodedString()`.

ASCII Codecs

These are the ASCII codec APIs. Only 7-bit ASCII data is accepted. All other codes generate errors.

PyObject* **PyUnicode_DecodeASCII**(const char **s*, Py_ssize_t *size*, const char **errors*)

> *Return value: New reference.*
>
> Create a Unicode object by decoding *size* bytes of the ASCII encoded string *s*. Return *NULL* if an exception was raised by the codec.

PyObject* **PyUnicode_AsASCIIString**(PyObject **unicode*)

> *Return value: New reference.*
>
> Encode a Unicode object using ASCII and return the result as Python bytes object. Error handling is “strict”. Return *NULL* if an exception was raised by the codec.

PyObject* **PyUnicode_EncodeASCII**(const Py_UNICODE **s*, Py_ssize_t *size*, const char **errors*)

> *Return value: New reference.*
>
> Encode the `Py_UNICODE` buffer of the given *size* using ASCII and return a Python bytes object. Return *NULL* if an exception was raised by the codec.
>
> *Deprecated since version 3.3, will be removed in version 4.0:* Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsASCIIString()` or `PyUnicode_AsEncodedString()`.

Character Map Codecs

This codec is special in that it can be used to implement many different codecs (and this is in fact what was done to obtain most of the standard codecs included in the `encodings` package). The codec uses mapping to encode and decode characters. The mapping objects provided must support the `__getitem__()` mapping interface; dictionaries and sequences work well.

These are the mapping codec APIs:

PyObject* **PyUnicode_DecodeCharmap**(const char **data*, Py_ssize_t *size*, PyObject **mapping*, const char **errors*)

> *Return value: New reference.*
>
> Create a Unicode object by decoding *size* bytes of the encoded string *s* using the given *mapping* object. Return *NULL* if an exception was raised by the codec.

If *mapping* is *NULL*, Latin-1 decoding will be applied. Else *mapping* must map bytes ordinals (integers in the range from 0 to 255) to Unicode strings, integers (which are then interpreted as Unicode ordinals) or `None`. Unmapped data bytes – ones which cause a `LookupError`, as well as ones which get mapped to `None`, `0xFFFE` or `'\ufffe'`, are treated as undefined mappings and cause an error.

PyObject* **`PyUnicode_AsCharmapString`**(PyObject **unicode*, PyObject **mapping*)

Return value: New reference.

Encode a Unicode object using the given *mapping* object and return the result as a bytes object. Error handling is “strict”. Return *NULL* if an exception was raised by the codec.

The *mapping* object must map Unicode ordinal integers to bytes objects, integers in the range from 0 to 255 or `None`. Unmapped character ordinals (ones which cause a `LookupError`) as well as mapped to `None` are treated as “undefined mapping” and cause an error.

PyObject* **`PyUnicode_EncodeCharmap`**(const Py_UNICODE **s*, Py_ssize_t *size*, PyObject **mapping*, const char **errors*)

Return value: New reference.

Encode the `Py_UNICODE` buffer of the given *size* using the given *mapping* object and return the result as a bytes object. Return *NULL* if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsCharmapString()` or `PyUnicode_AsEncodedString()`.

The following codec API is special in that maps Unicode to Unicode.

PyObject* **`PyUnicode_Translate`**(PyObject **unicode*, PyObject **mapping*, const char **errors*)

Return value: New reference.

Translate a Unicode object using the given *mapping* object and return the resulting Unicode object. Return *NULL* if an exception was raised by the codec.

The *mapping* object must map Unicode ordinal integers to Unicode strings, integers (which are then interpreted as Unicode ordinals) or `None` (causing deletion of the character). Unmapped character ordinals (ones which cause a `LookupError`) are left untouched and are copied as-is.

PyObject* **`PyUnicode_TranslateCharmap`**(const Py_UNICODE **s*, Py_ssize_t *size*, PyObject **mapping*, const char **errors*)

Return value: New reference.

Translate a `Py_UNICODE` buffer of the given *size* by applying a character *mapping* table to it and return the resulting Unicode object. Return *NULL* when an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_Translate()`. or generic codec based API

MBCS codecs for Windows

These are the MBCS codec APIs. They are currently only available on Windows and use the Win32 MBCS converters to implement the conversions. Note that MBCS (or DBCS) is a class of encodings, not just one. The target encoding is defined by the user settings on the machine running the codec.

PyObject* **`PyUnicode_DecodeMBCS`**(const char **s*, Py_ssize_t *size*, const char **errors*)

Return value: New reference.

Create a Unicode object by decoding *size* bytes of the MBCS encoded string *s*. Return *NULL* if an exception was raised by the codec.

PyObject* **`PyUnicode_DecodeMBCSStateful`**(const char **s*, int *size*, const char **errors*, int **consumed*)

If *consumed* is *NULL*, behave like `PyUnicode_DecodeMBCS()`. If *consumed* is not *NULL*, `PyUnicode_DecodeMBCSStateful()` will not decode trailing lead byte and the number of bytes that have been decoded will be stored in *consumed*.

PyObject* **`PyUnicode_AsMBCSString`**(PyObject **unicode*)

Return value: New reference.

Encode a Unicode object using MBCS and return the result as Python bytes object. Error handling is “strict”. Return *NULL* if an exception was raised by the codec.

PyObject* **`PyUnicode_EncodeCodePage`**(int *code_page*, PyObject **unicode*, const char **errors*)

Encode the Unicode object using the specified code page and return a Python bytes object. Return *NULL* if an exception was raised by the codec. Use `CP_ACP` code page to get the MBCS encoder.

New in version 3.3.

PyObject* **`PyUnicode_EncodeMBCS`**(const Py_UNICODE **s*, Py_ssize_t *size*, const char **errors*)

Return value: New reference.
Encode the `Py_UNICODE` buffer of the given *size* using MBCS and return a Python bytes object. Return *NULL* if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsMBCSString()`, `PyUnicode_EncodeCodePage()` or `PyUnicode_AsEncodedString()`.

Methods & Slots

Methods and Slot Functions

The following APIs are capable of handling Unicode objects and strings on input (we refer to them as strings in the descriptions) and return Unicode objects or integers as appropriate.

They all return *NULL* or `-1` if an exception occurs.

PyObject* **`PyUnicode_Concat`**(PyObject **left*, PyObject **right*)
Return value: New reference.
Concat two strings giving a new Unicode string.

PyObject* **`PyUnicode_Split`**(PyObject **s*, PyObject **sep*, Py_ssize_t *maxsplit*)

Return value: New reference.
Split a string giving a list of Unicode strings. If *sep* is *NULL*, splitting will be done at all whitespace substrings. Otherwise, splits occur at the given separator. At most *maxsplit* splits will be done. If negative, no limit is set. Separators are not included in the resulting list.

PyObject* **`PyUnicode_Splitlines`**(PyObject **s*, int *keepend*)
Return value: New reference.
Split a Unicode string at line breaks, returning a list of Unicode strings. CRLF is considered to be one line break. If *keepend* is `0`, the Line break characters are not included in the resulting strings.

PyObject* **`PyUnicode_Translate`**(PyObject **str*, PyObject **table*, const char **errors*)
Translate a string by applying a character mapping table to it and return the resulting Unicode object.

The mapping table must map Unicode ordinal integers to Unicode ordinal integers or `None` (causing deletion of the character).

Mapping tables need only provide the `__getitem__()` interface; dictionaries and sequences work well. Unmapped character ordinals (ones which cause a `LookupError`) are left untouched and are copied as-is.

errors has the usual meaning for codecs. It may be *NULL* which indicates to use the default error handling.

PyObject* **`PyUnicode_Join`**(PyObject **separator*, PyObject **seq*)

Return value: New reference.

Join a sequence of strings using the given *separator* and return the resulting Unicode string.

Py_ssize_t **`PyUnicode_Tailmatch`**(PyObject **str*, PyObject **substr*, Py_ssize_t *start*, Py_ssize_t *end*, int *direction*)

Return `1` if *substr* matches `str[start:end]` at the given tail end (*direction* == `-1` means to do a prefix match, *direction* == `1` a suffix match), `0` otherwise. Return `-1` if an error occurred.

Py_ssize_t **`PyUnicode_Find`**(PyObject **str*, PyObject **substr*, Py_ssize_t *start*, Py_ssize_t *end*, int *direction*)

Return the first position of *substr* in `str[start:end]` using the given *direction* (*direction* == `1` means to do a forward search, *direction* == `-1` a backward search). The return value is the index of the first match; a value of `-1` indicates that no match was found, and `-2` indicates that an error occurred and an exception has been set.

Py_ssize_t **`PyUnicode_FindChar`**(PyObject **str*, Py_UCS4 *ch*, Py_ssize_t *start*, Py_ssize_t *end*, int *direction*)

Return the first position of the character *ch* in `str[start:end]` using the given *direction* (*direction* == `1` means to do a forward search, *direction* == `-1` a backward search). The return value is the index of the first match; a value of `-1` indicates that no match was found, and `-2` indicates that an error occurred and an exception has been set.

New in version 3.3.

Py_ssize_t **`PyUnicode_Count`**(PyObject **str*, PyObject **substr*, Py_ssize_t *start*, Py_ssize_t *end*)

Return the number of non-overlapping occurrences of *substr* in `str[start:end]`. Return `-1` if an error occurred.

PyObject* **`PyUnicode_Replace`**(PyObject **str*, PyObject **substr*, PyObject **replstr*, Py_ssize_t *maxcount*)

Return value: New reference.

Replace at most *maxcount* occurrences of *substr* in *str* with *replstr* and return the resulting Unicode object. *maxcount* == `-1` means replace all occurrences.

int **`PyUnicode_Compare`**(PyObject **left*, PyObject **right*)

Compare two strings and return `-1`, `0`, `1` for less than, equal, and greater than, respectively.

This function returns `-1` upon failure, so one should call `PyErr_Occurred()` to check for errors.

int **`PyUnicode_CompareWithASCIIString`**(PyObject **uni*, const char **string*)

Compare a unicode object, *uni*, with *string* and return `-1`, `0`, `1` for less than, equal, and greater than, respectively. It is best to pass only ASCII-encoded strings, but the function interprets the input string as ISO-8859-1 if it contains non-ASCII characters.

This function does not raise exceptions.

PyObject* **`PyUnicode_RichCompare`**(PyObject **left*, PyObject **right*, int *op*)

Rich compare two unicode strings and return one of the following:

- `NULL` in case an exception was raised
- `Py_True` or `Py_False` for successful comparisons
- `Py_NotImplemented` in case the type combination is unknown

Possible values for *op* are `Py_GT`, `Py_GE`, `Py_EQ`, `Py_NE`, `Py_LT`, and `Py_LE`.

PyObject* **`PyUnicode_Format`**(PyObject **format*, PyObject **args*)

Return value: New reference.

Return a new string object from *format* and *args*; this is analogous to `format % args`.

int **`PyUnicode_Contains`**(PyObject **container*, PyObject **element*)

Check whether *element* is contained in *container* and return true or false accordingly.

element has to coerce to a one element Unicode string. `-1` is returned if there was an error.

void **`PyUnicode_InternInPlace`**(PyObject ***string*)

Intern the argument **string* in place. The argument must be the address of a pointer variable pointing to a Python unicode string object. If there is an existing interned string that is the same as **string*, it sets **string* to it (decrementing the reference count of the old string object and incrementing the reference count of the interned string object), otherwise it leaves **string* alone and interns it (incrementing its reference count). (Clarification: even though there is a lot of talk about reference counts, think of this function as reference-count-neutral; you own the object after the call if and only if you owned it before the call.)

PyObject* **`PyUnicode_InternFromString`**(const char **v*)

A combination of `PyUnicode_FromString()` and `PyUnicode_InternInPlace()`, returning either a new unicode string object that has been interned, or a new ("owned") reference to an earlier interned string object with the same value.