`!locale` --- Internationalization services

- Background, details, hints, tips and caveats
- Locale names
- For extension writers and programs that embed Python
- Access to message catalogs

locale

Source code: `Lib/locale.py`

The `!locale` module opens access to the POSIX locale database and functionality. The POSIX locale mechanism allows programmers to deal with certain cultural issues in an application, without requiring the programmer to know all the specifics of each country where the software is executed.

pair: module; locale

The `!locale` module is implemented on top of the `!_locale` module, which in turn uses an ANSI C locale implementation if available.

The `!locale` module defines the following exception and functions:

Error

Exception raised when the locale passed to `setlocale` is not recognized.

setlocale(category, locale=None)

If *locale* is given and not `None`, `setlocale` modifies the locale setting for the *category*. The available categories are listed in the data description below. *locale* may be a `string <locale_name>`, or a pair, language code and encoding. An empty string specifies the user's default settings. If the modification of the locale fails, the exception `Error` is raised. If successful, the new locale setting is returned.

If *locale* is a pair, it is converted to a locale name using the locale aliasing engine. The language code has the same format as a `locale name <locale_name>`, but without encoding. The language code and encoding can be `None`.

If *locale* is omitted or `None`, the current setting for *category* is returned.

Example:

```
>>> import locale
>>> loc = locale.setlocale(locale.LC_ALL)  # get current locale
# use German locale; name and availability varies with platform
>>> locale.setlocale(locale.LC_ALL, 'de_DE.UTF-8')
>>> locale.strcoll('f\xe4n', 'foo')  # compare a string containing an umlaut
>>> locale.setlocale(locale.LC_ALL, '')   # use user's preferred locale
>>> locale.setlocale(locale.LC_ALL, 'C')  # use default (C) locale
>>> locale.setlocale(locale.LC_ALL, loc)  # restore saved locale
```

`setlocale` is not thread-safe on most systems. Applications typically start with a call of:

```
import locale
locale.setlocale(locale.LC_ALL, '')
```

This sets the locale for all categories to the user's default setting (typically specified in the `LANG` environment variable). If the locale is not changed thereafter, using multithreading should not cause problems.

3.15 Support language codes with `@`-modifiers.

localeconv()

Returns the database of the local conventions as a dictionary. This dictionary has the following strings as keys:

|l|l|L|

Category	Key	Meaning
`LC_NUMERIC`	`'decimal_point'`	Decimal point character.
	`'grouping'`	Sequence of numbers specifying which relative positions the `'thousands_sep'` is expected. If the sequence is terminated with `CHAR_MAX`, no further grouping is performed. If the sequence terminates with a `0`, the last group size is repeatedly used.

Category	Key	Meaning
	`'thousands_sep'`	Character used between groups.
`LC_MONETARY`	`'int_curr_symbol'`	International currency symbol.
	`'currency_symbol'`	Local currency symbol.
	`'p_cs_precedes/n_cs_precedes'`	Whether the currency symbol precedes the value (for positive resp. negative values).
	`'p_sep_by_space/n_sep_by_space'`	Whether the currency symbol is separated from the value by a space (for positive resp. negative values).
	`'mon_decimal_point'`	Decimal point used for monetary values.
	`'frac_digits'`	Number of fractional digits used in local formatting of monetary values.
	`'int_frac_digits'`	Number of fractional digits used in international formatting of monetary values.
	`'mon_thousands_sep'`	Group separator used for monetary values.
	`'mon_grouping'`	Equivalent to `'grouping'`, used for monetary values.
	`'positive_sign'`	Symbol used to annotate a positive monetary value.
	`'negative_sign'`	Symbol used to annotate a negative monetary value.
	`'p_sign_posn/n_sign_posn'`	The position of the sign (for positive resp. negative values), see below.

All numeric values can be set to `CHAR_MAX` to indicate that there is no value specified in this locale.

The possible values for `'p_sign_posn'` and `'n_sign_posn'` are given below.

Value	Explanation
`0`	Currency and value are surrounded by parentheses.
`1`	The sign should precede the value and currency symbol.
`2`	The sign should follow the value and currency symbol.
`3`	The sign should immediately precede the value.
`4`	The sign should immediately follow the value.
`CHAR_MAX`	Nothing is specified in this locale.

The function temporarily sets the `LC_CTYPE` locale to the `LC_NUMERIC` locale or the `LC_MONETARY` locale if locales are different and numeric or monetary strings are non-ASCII. This temporary change affects other threads.

3.7 The function now temporarily sets the `LC_CTYPE` locale to the `LC_NUMERIC` locale in some cases.

nl_langinfo(option)

Return some locale-specific information as a string. This function is not available on all systems, and the set of possible options might also vary across platforms. The possible argument values are numbers, for which symbolic constants are available in the locale module.

The `nl_langinfo` function accepts one of the following keys. Most descriptions are taken from the corresponding description in the GNU C library.

CODESET

Get a string with the name of the character encoding used in the selected locale.

D_T_FMT

Get a string that can be used as a format string for `time.strftime` to represent date and time in a locale-specific way.

D_FMT

Get a string that can be used as a format string for `time.strftime` to represent a date in a locale-specific way.

T_FMT

Get a string that can be used as a format string for `time.strftime` to represent a time in a locale-specific way.

T_FMT_AMPM

Get a format string for `time.strftime` to represent time in the am/pm format.

DAY_1 DAY_2 DAY_3 DAY_4 DAY_5 DAY_6 DAY_7

Get the name of the n-th day of the week.

> [!NOTE] This follows the US convention of `DAY_1` being Sunday, not the international convention (ISO 8601) that Monday is the first day of the week.

ABDAY_1 ABDAY_2 ABDAY_3 ABDAY_4 ABDAY_5 ABDAY_6 ABDAY_7

Get the abbreviated name of the n-th day of the week.

MON_1 MON_2 MON_3 MON_4 MON_5 MON_6 MON_7 MON_8 MON_9 MON_10 MON_11 MON_12

Get the name of the n-th month.

ABMON_1 ABMON_2 ABMON_3 ABMON_4 ABMON_5 ABMON_6 ABMON_7 ABMON_8 ABMON_9 ABMON_10 ABMON_11 ABMON_12

Get the abbreviated name of the n-th month.

RADIXCHAR

Get the radix character (decimal dot, decimal comma, etc.).

THOUSEP

Get the separator character for thousands (groups of three digits).

YESEXPR

Get a regular expression that can be used with the regex function to recognize a positive response to a yes/no question.

NOEXPR

Get a regular expression that can be used with the `regex(3)` function to recognize a negative response to a yes/no question.

> [!NOTE] The regular expressions for `YESEXPR` and `NOEXPR` use syntax suitable for the `regex` function from the C library, which might differ from the syntax used in `re`.

CRNCYSTR

Get the currency symbol, preceded by "-" if the symbol should appear before the value, "+" if the symbol should appear after the value, or "." if the symbol should replace the radix character.

ERA

Get a string which describes how years are counted and displayed for each era in a locale.

Most locales do not define this value. An example of a locale which does define this value is the Japanese one. In Japan, the traditional representation of dates includes the name of the era corresponding to the then-emperor's reign.

Normally it should not be necessary to use this value directly. Specifying the `E` modifier in their format strings causes the `time.strftime` function to use this information. The format of the returned string is specified in *The Open Group Base Specifications Issue 8*, paragraph 7.3.5.2 LC_TIME C-Language Access.

ERA_D_T_FMT

Get a format string for `time.strftime` to represent date and time in a locale-specific era-based way.

ERA_D_FMT

Get a format string for `time.strftime` to represent a date in a locale-specific era-based way.

ERA_T_FMT

Get a format string for `time.strftime` to represent a time in a locale-specific era-based way.

ALT_DIGITS

Get a string consisting of up to 100 semicolon-separated symbols used to represent the values 0 to 99 in a locale-specific way. In most locales this is an empty string.

The function temporarily sets the `LC_CTYPE` locale to the locale of the category that determines the requested value (`LC_TIME`, `LC_NUMERIC`, `LC_MONETARY` or `LC_MESSAGES`) if locales are different and the resulting string is non-ASCII. This temporary change affects other threads.

3.14 The function now temporarily sets the `LC_CTYPE` locale in some cases.

3.16 On glibc, the `LC_TIME` items (except `ERA`) are now decoded independently of the `LC_CTYPE` encoding.

getdefaultlocale([envvars])

Tries to determine the default locale settings and returns them as a tuple of the form `(language code, encoding)`.

According to POSIX, a program which has not called `setlocale(LC_ALL, '')` runs using the portable `'C'` locale. Calling `setlocale(LC_ALL, '')` lets it use the default locale as defined by the `LANG` variable. Since we do not want to interfere with the current locale setting we thus emulate the behavior in the way described above.

To maintain compatibility with other platforms, not only the `LANG` variable is tested, but a list of variables given as envvars parameter. The first found to be defined will be used. *envvars* defaults to the search path used in GNU gettext; it must always contain the variable name `'LANG'`. The GNU gettext search path contains `'LC_ALL'`, `'LC_CTYPE'`, `'LANG'` and `'LANGUAGE'`, in that order.

The language code has the same format as a `locale name <locale_name>`, but without encoding and `@`-modifier. The language code and encoding may be `None` if their values cannot be determined. The "C" locale is represented as `(None, None)`.

getlocale(category=LC_CTYPE)

Returns the current setting for the given locale category as a tuple containing the language code and encoding. *category* may be one of the `!LC_\*` values except `LC_ALL`. It defaults to `LC_CTYPE`.

The language code has the same format as a `locale name <locale_name>`, but without encoding. The language code and encoding may be `None` if their values cannot be determined. The "C" locale is represented as `(None, None)`.

3.15 `@`-modifier are no longer silently removed, but included in the language code.

getpreferredencoding(do_setlocale=True)

Return the `locale encoding` used for text data, according to user preferences. User preferences are expressed differently on different systems, and might not be available programmatically on some systems, so this function only returns a guess.

On some systems, it is necessary to invoke `setlocale` to obtain the user preferences, so this function is not thread-safe. If invoking setlocale is not necessary or desired, *do_setlocale* should be set to `False`.

On Android or if the `Python UTF-8 Mode <utf8-mode>` is enabled, always return `'utf-8'`, the `locale encoding` and the *do_setlocale* argument are ignored.

The `Python preinitialization <c-preinit>` configures the LC_CTYPE locale. See also the `filesystem encoding and error handler`.

3.7 The function now always returns `"utf-8"` on Android or if the `Python UTF-8 Mode <utf8-mode>` is enabled.

getencoding()

Get the current `locale encoding`:

- On Android and VxWorks, return `"utf-8"`.
- On Unix, return the encoding of the current `LC_CTYPE` locale. Return `"utf-8"` if `nl_langinfo(CODESET)` returns an empty string: for example, if the current LC_CTYPE locale

is not supported.
- On Windows, return the ANSI code page.

The `Python preinitialization <c-preinit>` configures the LC_CTYPE locale. See also the `filesystem encoding and error handler`.

This function is similar to `getpreferredencoding(False) <getpreferredencoding>` except this function ignores the `Python UTF-8 Mode <utf8-mode>`.

3.11

normalize(localename)

Returns a normalized locale code for the given locale name. The returned locale code is formatted for use with `setlocale`. If normalization fails, the original name is returned unchanged.

If the given encoding is not known, the function defaults to the default encoding for the locale code just like `setlocale`.

strcoll(string1, string2)

Compares two strings according to the current `LC_COLLATE` setting. As any other compare function, returns a negative, or a positive value, or `0`, depending on whether *string1* collates before or after *string2* or is equal to it.

strxfrm(string)

Transforms a string to one that can be used in locale-aware comparisons. For example, `strxfrm(s1) < strxfrm(s2)` is equivalent to `strcoll(s1, s2) < 0`. This function can be used when the same string is compared repeatedly, e.g. when collating a sequence of strings.

format_string(format, val, grouping=False, monetary=False)

Formats a number *val* according to the current `LC_NUMERIC` setting. The format follows the conventions of the `%` operator. For floating-point values, the decimal point is modified if appropriate. If *grouping* is `True`, also takes the grouping into account.

If *monetary* is true, the conversion uses monetary thousands separator and grouping strings.

Processes formatting specifiers as in `format % val`, but takes the current locale settings into account.

3.7 The *monetary* keyword parameter was added.

currency(val, symbol=True, grouping=False, international=False)

Formats a number *val* according to the current `LC_MONETARY` settings.

The returned string includes the currency symbol if *symbol* is true, which is the default. If *grouping* is `True` (which is not the default), grouping is done with the value. If *international* is `True` (which is not the default), the international currency symbol is used.

> [!NOTE] This function will not work with the 'C' locale, so you have to set a locale via `setlocale` first.

str(float)

Formats a floating-point number using the same format as the built-in function `str(float)`, but takes the decimal point into account.

delocalize(string)

Converts a string into a normalized number string, following the `LC_NUMERIC` settings.

3.5

localize(string, grouping=False, monetary=False)

Converts a normalized number string into a formatted string following the `LC_NUMERIC` settings.

3.10

atof(string, func=float)

Converts a string to a number, following the `LC_NUMERIC` settings, by calling *func* on the result of calling `delocalize` on *string*.

atoi(string)

Converts a string to an integer, following the `LC_NUMERIC` conventions.

LC_CTYPE

Locale category for the character type functions. Most importantly, this category defines the text encoding, i.e. how bytes are interpreted as Unicode codepoints. See `538` and `540` for how this variable might be automatically coerced to `C.UTF-8` to avoid issues created by invalid settings in containers or incompatible settings passed over remote SSH connections.

Python doesn't internally use locale-dependent character transformation functions from `ctype.h`. Instead, `pyctype.h` provides locale-independent equivalents like `Py_TOLOWER`.

LC_COLLATE

Locale category for sorting strings. The functions `strcoll` and `strxfrm` of the `!locale` module are affected.

LC_TIME

Locale category for the formatting of time. The function `time.strftime` follows these conventions.

LC_MONETARY

Locale category for formatting of monetary values. The available options are available from the `localeconv` function.

LC_MESSAGES

Locale category for message display. Python currently does not support application specific locale-aware messages. Messages displayed by the operating system, like those returned by `os.strerror` might be affected by this category.

This value may not be available on operating systems not conforming to the POSIX standard, most notably Windows.

LC_NUMERIC

Locale category for formatting numbers. The functions `format_string`, `atoi`, `atof` and `.str` of the `!locale` module are affected by that category. All other numeric formatting operations are not affected.

LC_ALL

Combination of all locale settings. If this flag is used when the locale is changed, setting the locale for all categories is attempted. If that fails for any category, no category is changed at all. When the locale is retrieved using this flag, a string indicating the setting for all categories is returned. This string can be later used to restore the settings.

CHAR_MAX

This is a symbolic constant used for different values returned by `localeconv`.

Background, details, hints, tips and caveats

The C standard defines the locale as a program-wide property that may be relatively expensive to change. On top of that, some implementations are broken in such a way that frequent locale changes may cause core dumps. This makes the locale somewhat painful to use correctly.

Initially, when a program is started, the locale is the `C` locale, no matter what the user's preferred locale is. There is one exception: the `LC_CTYPE` category is changed at startup to set the current locale encoding to the user's preferred locale encoding. The program must explicitly say that it wants the user's preferred locale settings for other categories by calling `setlocale(LC_ALL, '')`.

It is generally a bad idea to call `setlocale` in some library routine, since as a side effect it affects the entire program. Saving and restoring it is almost as bad: it is expensive and affects other threads that happen to run before the settings have been restored.

If, when coding a module for general use, you need a locale independent version of an operation that is affected by the locale (such as certain formats used with `time.strftime`), you will have to find a way to do it without using the standard library routine. Even better is convincing yourself that using locale settings is okay. Only as a last resort should you document that your module is not compatible with non-`C` locale settings.

The only way to perform numeric operations according to the locale is to use the special functions defined by this module: `atof`, `atoi`, `format_string`, `.str`.

There is no way to perform case conversions and character classifications according to the locale. For (Unicode) text strings these are done according to the character value only, while for byte strings, the conversions and classifications are done according to the ASCII value of the byte, and bytes whose high bit is set (i.e., non-ASCII bytes) are never converted or considered part of a character class such as letter or whitespace.

Locale names

The format of the locale name is platform dependent, and the set of supported locales can depend on the system configuration.

On Posix platforms, it usually has the format[^1]:

locale_name : language ["_" territory] ["." charset] ["@" modifier]

where *language* is a two- or three-letter language code from ISO 639, *territory* is a two-letter country or region code from ISO 3166, *charset* is a locale encoding, and *modifier* is a script name, a language subtag, a sort order identifier, or other locale modifier (for example, "latin", "valencia", "stroke" and "euro").

On Windows, several formats are supported.[^2][^3] A subset of IETF BCP 47 tags:

locale_name : language ["-" script] ["-" territory] ["." charset] : language ["-" script] "-" territory "-" modifier

where *language* and *territory* have the same meaning as in Posix, *script* is a four-letter script code from ISO 15924, and *modifier* is a language subtag, a sort order identifier or custom modifier (for example, "valencia", "stroke" or "x-python"). Both hyphen (`'-'`) and underscore (`'_'`) separators are supported. Only UTF-8 encoding is allowed for BCP 47 tags.

Windows also supports locale names in the format:

locale_name : language ["_" territory] ["." charset]

where *language* and *territory* are full names, such as "English" and "United States", and *charset* is either a code page number (for example, "1252") or UTF-8. Only the underscore separator is supported in this format.

The "C" locale is supported on all platforms.

For extension writers and programs that embed Python

Extension modules should never call `setlocale`, except to find out what the current locale is. But since the return value can only be used portably to restore it, that is not very useful (except perhaps to find out whether or not the locale is `C`).

When Python code uses the `!locale` module to change the locale, this also affects the embedding application. If the embedding application doesn't want this to happen, it should remove the `!_locale` extension module (which does all the work) from the table of built-in modules in the `config.c` file, and make sure that the `!_locale` module is not accessible as a shared library.

Access to message catalogs

gettext(msg)

dgettext(domain, msg)

dcgettext(domain, msg, category)

textdomain(domain)

bindtextdomain(domain, dir)

bind_textdomain_codeset(domain, codeset)

The locale module exposes the C library's gettext interface on systems that provide this interface. It consists of the functions `~locale.gettext`, `dgettext`, `dcgettext`, `textdomain`, `bindtextdomain`, and `bind_textdomain_codeset`. These are similar to the same functions in the `gettext` module, but use the C library's binary format for message catalogs, and the C library's search algorithms for locating message catalogs.

Python applications should normally find no need to invoke these functions, and should use `gettext` instead. A known exception to this rule are applications that link with additional C libraries which internally invoke C functions `gettext` or `dcgettext`. For these applications, it may be necessary to bind the text domain, so that the libraries can properly locate their message catalogs.

[^1]: IEEE Std 1003.1-2024; 8.2 Internationalization Variables

[^2]: UCRT Locale names, Languages, and Country/Region strings

[^3]: Locale Names

□□□ #2
□ □□□ □□□ 2026-10-31 17:13:43 UTC
□ □□□ □□□ 2026-10-31 17:13:54 UTC