23.2. `locale` — Internationalization services

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.

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:

exception `locale.`**`Error`**

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

`locale.`**`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, or an iterable of two strings (language code and encoding). If it's an iterable, it's converted to a locale name using the locale aliasing engine. 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 omitted or `None`, the current setting for *category* is returned.

`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.

`locale.`**`localeconv`**()

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

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.
	'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'	

Category	Key	Meaning
		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 sets temporarily the `LC_CTYPE` locale to the `LC_NUMERIC` locale to decode `decimal_point` and `thousands_sep` byte strings if they are non-ASCII or longer than 1 byte, and the `LC_NUMERIC` locale is different than the `LC_CTYPE` locale. This temporary change affects other threads.

Changed in version 3.6.5: The function now sets temporarily the `LC_CTYPE` locale to the `LC_NUMERIC` locale in some cases.

`locale.`**`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.

`locale.`**`CODESET`**

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

`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.

`locale.`**`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.

`locale.`**`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.

`locale.`**`T_FMT_AMPM`**

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

`DAY_1 ... 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_7`

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

`MON_1 ... MON_12`

Get the name of the n-th month.

ABMON_1 ... ABMON_12

Get the abbreviated name of the n-th month.

`locale.`**RADIXCHAR**

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

`locale.`**THOUSEP**

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

`locale.`**YESEXPR**

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

> **Note:** The expression is in the syntax suitable for the `regex()` function from the C library, which might differ from the syntax used in `re`.

`locale.`**NOEXPR**

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

`locale.`**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.

`locale.`**ERA**

Get a string that represents the era used in the current 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 not specified, and therefore you should not assume knowledge of it on different systems.

`locale.`**ERA_D_T_FMT**

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

`locale.`**ERA_D_FMT**

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

`locale.`**ERA_T_FMT**

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

`locale.`**ALT_DIGITS**

Get a representation of up to 100 values used to represent the values 0 to 99.

`locale.`**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.

Except for the code `'C'`, the language code corresponds to **RFC 1766**. *language code* and *encoding* may be `None` if their values cannot be determined.

`locale.`**getlocale**(*category=LC_CTYPE*)

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

Except for the code `'C'`, the language code corresponds to **RFC 1766**. *language code* and *encoding* may be `None` if their values cannot be determined.

`locale.`**getpreferredencoding**(*do_setlocale=True*)

Return the 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`.

`locale.`**`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()`.

`locale.`**`resetlocale`**(*category=LC_ALL*)

Sets the locale for *category* to the default setting.

The default setting is determined by calling `getdefaultlocale()`. *category* defaults to `LC_ALL`.

`locale.`**`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.

`locale.`**`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.

`locale.`**`format`**(*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.

Please note that this function will only work for exactly one %char specifier. For whole format strings, use `format_string()`.

`locale.`**`format_string`**(*format*, *val*, *grouping=False*)

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

`locale.`**`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 that this function will not work with the 'C' locale, so you have to set a locale via `setlocale()` first.

`locale.`**`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.

`locale.`**`delocalize`**(*string*)

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

New in version 3.5.

`locale.`**`atof`**(*string*)

Converts a string to a floating point number, following the `LC_NUMERIC` settings.

`locale.`**`atoi`**(*string*)

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

`locale.`**`LC_CTYPE`**

Locale category for the character type functions. Depending on the settings of this category, the functions of module `string` dealing with case change their behaviour.

`locale.`**`LC_COLLATE`**

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

`locale.`**`LC_TIME`**

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

`locale.`**`LC_MONETARY`**

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

`locale.`**`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.

`locale.`**LC_NUMERIC**

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

`locale.`**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.

`locale.`**CHAR_MAX**

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

Example:

```
>>> import locale
>>> loc = locale.getlocale()  # get current locale
# use German locale; name might vary with platform
>>> locale.setlocale(locale.LC_ALL, 'de_DE')
>>> locale.strcoll('f\xe4n', 'foo')  # compare a string containing an
>>> locale.setlocale(locale.LC_ALL, '')   # use user's preferred local
>>> locale.setlocale(locale.LC_ALL, 'C')  # use default (C) locale
>>> locale.setlocale(locale.LC_ALL, loc)  # restore saved locale
```

23.2.1. 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 implementation 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()`, `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.

23.2.2. 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.

23.2.3. Access to message catalogs

`locale.`**`gettext`**(*msg*)

`locale.`**`dgettext`**(*domain*, *msg*)

`locale.`**`dcgettext`**(*domain*, *msg*, *category*)

`locale.`**`textdomain`**(*domain*)

`locale.`**`bindtextdomain`**(*domain*, *dir*)

The locale module exposes the C library's gettext interface on systems that provide this interface. It consists of the functions `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 `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.