Reentrancy

Binary buffered objects (instances of *BufferedReader*, *BufferedWriter*, *BufferedRandom* and *BufferedRWPair*) are not reentrant. While reentrant calls will not happen in normal situations, they can arise from doing I/O in a *signal* handler. If a thread tries to re-enter a buffered object which it is already accessing, a *RuntimeError* is raised. Note this doesn't prohibit a different thread from entering the buffered object.

The above implicitly extends to text files, since the *open()* function will wrap a buffered object inside a *TextIOWrapper*. This includes standard streams and therefore affects the built-in *print()* function as well.

16.3 `time` — Time access and conversions

This module provides various time-related functions. For related functionality, see also the *datetime* and *calendar* modules.

Although this module is always available, not all functions are available on all platforms. Most of the functions defined in this module call platform C library functions with the same name. It may sometimes be helpful to consult the platform documentation, because the semantics of these functions varies among platforms.

An explanation of some terminology and conventions is in order.

- The *epoch* is the point where the time starts, the return value of `time.gmtime(0)`. It is January 1, 1970, 00:00:00 (UTC) on all platforms.
- The term *seconds since the epoch* refers to the total number of elapsed seconds since the epoch, typically excluding leap seconds. Leap seconds are excluded from this total on all POSIX-compliant platforms.
- The functions in this module may not handle dates and times before the *epoch* or far in the future. The cut-off point in the future is determined by the C library; for 32-bit systems, it is typically in 2038.
- Function *strptime()* can parse 2-digit years when given `%y` format code. When 2-digit years are parsed, they are converted according to the POSIX and ISO C standards: values 69–99 are mapped to 1969–1999, and values 0–68 are mapped to 2000–2068.
- UTC is Coordinated Universal Time (formerly known as Greenwich Mean Time, or GMT). The acronym UTC is not a mistake but a compromise between English and French.
- DST is Daylight Saving Time, an adjustment of the timezone by (usually) one hour during part of the year. DST rules are magic (determined by local law) and can change from year to year. The C library has a table containing the local rules (often it is read from a system file for flexibility) and is the only source of True Wisdom in this respect.
- The precision of the various real-time functions may be less than suggested by the units in which their value or argument is expressed. E.g. on most Unix systems, the clock "ticks" only 50 or 100 times a second.
- On the other hand, the precision of *time()* and *sleep()* is better than their Unix equivalents: times are expressed as floating point numbers, *time()* returns the most accurate time available (using Unix `gettimeofday()` where available), and *sleep()* will accept a time with a nonzero fraction (Unix `select()` is used to implement this, where available).
- The time value as returned by *gmtime()*, *localtime()*, and *strptime()*, and accepted by *asctime()*, *mktime()* and *strftime()*, is a sequence of 9 integers. The return values of *gmtime()*, *localtime()*, and *strptime()* also offer attribute names for individual fields.

  See *struct_time* for a description of these objects.

  Changed in version 3.3: The *struct_time* type was extended to provide the `tm_gmtoff` and `tm_zone` attributes when platform supports corresponding `struct tm` members.

  Changed in version 3.6: The *struct_time* attributes `tm_gmtoff` and `tm_zone` are now available on all platforms.

- Use the following functions to convert between time representations:

From	To	Use
seconds since the epoch	*struct_time* in UTC	*gmtime()*
seconds since the epoch	*struct_time* in local time	*localtime()*
struct_time in UTC	seconds since the epoch	*calendar.timegm()*
struct_time in local time	seconds since the epoch	*mktime()*

16.3.1 Functions

time.**asctime**([*t*])

Convert a tuple or *struct_time* representing a time as returned by *gmtime()* or *localtime()* to a string of the following form: `'Sun Jun 20 23:21:05 1993'`. The day field is two characters long and is space padded if the day is a single digit, e.g.: `'Wed Jun 9 04:26:40 1993'`.

If *t* is not provided, the current time as returned by *localtime()* is used. Locale information is not used by *asctime()*.

Note: Unlike the C function of the same name, *asctime()* does not add a trailing newline.

time.**pthread_getcpuclockid**(*thread_id*)

Return the *clk_id* of the thread-specific CPU-time clock for the specified *thread_id*.

Use *threading.get_ident()* or the *ident* attribute of *threading.Thread* objects to get a suitable value for *thread_id*.

Warning: Passing an invalid or expired *thread_id* may result in undefined behavior, such as segmentation fault.

Availability: Unix

See the man page for *pthread_getcpuclockid(3)* for further information.

New in version 3.7.

time.**clock_getres**(*clk_id*)

Return the resolution (precision) of the specified clock *clk_id*. Refer to *Clock ID Constants* for a list of accepted values for *clk_id*.

Availability: Unix.

New in version 3.3.

time.**clock_gettime**(*clk_id*) → *float*

Return the time of the specified clock *clk_id*. Refer to *Clock ID Constants* for a list of accepted values for *clk_id*.

Use *clock_gettime_ns()* to avoid the precision loss caused by the *float* type.

Availability: Unix.

New in version 3.3.

time.**clock_gettime_ns**(*clk_id*) → *int*

Similar to *clock_gettime()* but return time as nanoseconds.

Availability: Unix.

New in version 3.7.

time.**clock_settime**(*clk_id*, *time:* float)

Set the time of the specified clock *clk_id*. Currently, `CLOCK_REALTIME` is the only accepted value for *clk_id*.

Use `clock_settime_ns()` to avoid the precision loss caused by the `float` type.

Availability: Unix.

New in version 3.3.

time.**clock_settime_ns**(*clk_id*, *time:* int)

Similar to `clock_settime()` but set time with nanoseconds.

Availability: Unix.

New in version 3.7.

time.**ctime**([*secs*])

Convert a time expressed in seconds since the *epoch* to a string of a form: `'Sun Jun 20 23:21:05 1993'` representing local time. The day field is two characters long and is space padded if the day is a single digit, e.g.: `'Wed Jun 9 04:26:40 1993'`.

If *secs* is not provided or `None`, the current time as returned by `time()` is used. `ctime(secs)` is equivalent to `asctime(localtime(secs))`. Locale information is not used by `ctime()`.

time.**get_clock_info**(*name*)

Get information on the specified clock as a namespace object. Supported clock names and the corresponding functions to read their value are:

- `'monotonic'`: `time.monotonic()`
- `'perf_counter'`: `time.perf_counter()`
- `'process_time'`: `time.process_time()`
- `'thread_time'`: `time.thread_time()`
- `'time'`: `time.time()`

The result has the following attributes:

- *adjustable*: `True` if the clock can be changed automatically (e.g. by a NTP daemon) or manually by the system administrator, `False` otherwise
- *implementation*: The name of the underlying C function used to get the clock value. Refer to *Clock ID Constants* for possible values.
- *monotonic*: `True` if the clock cannot go backward, `False` otherwise
- *resolution*: The resolution of the clock in seconds (`float`)

New in version 3.3.

time.**gmtime**([*secs*])

Convert a time expressed in seconds since the *epoch* to a `struct_time` in UTC in which the dst flag is always zero. If *secs* is not provided or `None`, the current time as returned by `time()` is used. Fractions of a second are ignored. See above for a description of the `struct_time` object. See `calendar.timegm()` for the inverse of this function.

time.**localtime**([*secs*])

Like `gmtime()` but converts to local time. If *secs* is not provided or `None`, the current time as returned by `time()` is used. The dst flag is set to `1` when DST applies to the given time.

`localtime()` may raise `OverflowError`, if the timestamp is outside the range of values supported by the platform C `localtime()` or `gmtime()` functions, and `OSError` on `localtime()` or `gmtime()` failure. It's common for this to be restricted to years between 1970 and 2038.

time.**mktime**(*t*)

This is the inverse function of *localtime()*. Its argument is the *struct_time* or full 9-tuple (since the dst flag is needed; use `-1` as the dst flag if it is unknown) which expresses the time in *local* time, not UTC. It returns a floating point number, for compatibility with *time()*. If the input value cannot be represented as a valid time, either *OverflowError* or *ValueError* will be raised (which depends on whether the invalid value is caught by Python or the underlying C libraries). The earliest date for which it can generate a time is platform-dependent.

time.**monotonic**() → *float*

Return the value (in fractional seconds) of a monotonic clock, i.e. a clock that cannot go backwards. The clock is not affected by system clock updates. The reference point of the returned value is undefined, so that only the difference between the results of two calls is valid.

Use *monotonic_ns()* to avoid the precision loss caused by the *float* type.

New in version 3.3.

Changed in version 3.5: The function is now always available and always system-wide.

Changed in version 3.10: On macOS, the function is now system-wide.

time.**monotonic_ns**() → *int*

Similar to *monotonic()*, but return time as nanoseconds.

New in version 3.7.

time.**perf_counter**() → *float*

Return the value (in fractional seconds) of a performance counter, i.e. a clock with the highest available resolution to measure a short duration. It does include time elapsed during sleep and is system-wide. The reference point of the returned value is undefined, so that only the difference between the results of two calls is valid.

Use *perf_counter_ns()* to avoid the precision loss caused by the *float* type.

New in version 3.3.

Changed in version 3.10: On Windows, the function is now system-wide.

time.**perf_counter_ns**() → *int*

Similar to *perf_counter()*, but return time as nanoseconds.

New in version 3.7.

time.**process_time**() → *float*

Return the value (in fractional seconds) of the sum of the system and user CPU time of the current process. It does not include time elapsed during sleep. It is process-wide by definition. The reference point of the returned value is undefined, so that only the difference between the results of two calls is valid.

Use *process_time_ns()* to avoid the precision loss caused by the *float* type.

New in version 3.3.

time.**process_time_ns**() → *int*

Similar to *process_time()* but return time as nanoseconds.

New in version 3.7.

time.**sleep**(*secs*)

Suspend execution of the calling thread for the given number of seconds. The argument may be a floating point number to indicate a more precise sleep time.

If the sleep is interrupted by a signal and no exception is raised by the signal handler, the sleep is restarted with a recomputed timeout.

The suspension time may be longer than requested by an arbitrary amount, because of the scheduling of other activity in the system.

On Windows, if *secs* is zero, the thread relinquishes the remainder of its time slice to any other thread that is ready to run. If there are no other threads ready to run, the function returns immediately, and the thread continues execution. On Windows 8.1 and newer the implementation uses a high-resolution timer which provides resolution of 100 nanoseconds. If *secs* is zero, `Sleep(0)` is used.

Unix implementation:

- Use `clock_nanosleep()` if available (resolution: 1 nanosecond);
- Or use `nanosleep()` if available (resolution: 1 nanosecond);
- Or use `select()` (resolution: 1 microsecond).

Changed in version 3.11: On Unix, the `clock_nanosleep()` and `nanosleep()` functions are now used if available. On Windows, a waitable timer is now used.

Changed in version 3.5: The function now sleeps at least *secs* even if the sleep is interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale).

`time.`**`strftime`**(*format*[, *t*])

Convert a tuple or *struct_time* representing a time as returned by *gmtime()* or *localtime()* to a string as specified by the *format* argument. If *t* is not provided, the current time as returned by *localtime()* is used. *format* must be a string. *ValueError* is raised if any field in *t* is outside of the allowed range.

0 is a legal argument for any position in the time tuple; if it is normally illegal the value is forced to a correct one.

The following directives can be embedded in the *format* string. They are shown without the optional field width and precision specification, and are replaced by the indicated characters in the *strftime()* result:

Directive	Meaning	Notes
`%a`	Locale's abbreviated weekday name.	
`%A`	Locale's full weekday name.	
`%b`	Locale's abbreviated month name.	
`%B`	Locale's full month name.	
`%c`	Locale's appropriate date and time representation.	
`%d`	Day of the month as a decimal number [01,31].	
`%H`	Hour (24-hour clock) as a decimal number [00,23].	
`%I`	Hour (12-hour clock) as a decimal number [01,12].	
`%j`	Day of the year as a decimal number [001,366].	
`%m`	Month as a decimal number [01,12].	
`%M`	Minute as a decimal number [00,59].	
`%p`	Locale's equivalent of either AM or PM.	(1)
`%S`	Second as a decimal number [00,61].	(2)
`%U`	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.	(3)
`%w`	Weekday as a decimal number [0(Sunday),6].	
`%W`	Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.	(3)
`%x`	Locale's appropriate date representation.	
`%X`	Locale's appropriate time representation.	
`%y`	Year without century as a decimal number [00,99].	
`%Y`	Year with century as a decimal number.	
`%z`	Time zone offset indicating a positive or negative time difference from UTC/GMT of the form +HHMM or -HHMM, where H represents decimal hour digits and M represents decimal minute digits [-23:59, +23:59].[1]	
`%Z`	Time zone name (no characters if no time zone exists). Deprecated.[Page 674, 1]	
`%%`	A literal `'%'` character.	

Notes:

(1) When used with the *strptime()* function, the %p directive only affects the output hour field if the %I directive is used to parse the hour.

(2) The range really is 0 to 61; value 60 is valid in timestamps representing leap seconds and value 61 is supported for historical reasons.

(3) When used with the *strptime()* function, %U and %W are only used in calculations when the day of the week and the year are specified.

Here is an example, a format for dates compatible with that specified in the **RFC 2822** Internet email standard.[1]

```
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Thu, 28 Jun 2001 14:17:15 +0000'
```

Additional directives may be supported on certain platforms, but only the ones listed here have a meaning standardized by ANSI C. To see the full set of format codes supported on your platform, consult the *strftime(3)* documentation.

On some platforms, an optional field width and precision specification can immediately follow the initial '%' of a directive in the following order; this is also not portable. The field width is normally 2 except for %j where it is 3.

time.**strptime**(*string*[, *format*])

Parse a string representing a time according to a format. The return value is a *struct_time* as returned by *gmtime()* or *localtime()*.

The *format* parameter uses the same directives as those used by *strftime()*; it defaults to "%a %b %d %H:%M:%S %Y" which matches the formatting returned by *ctime()*. If *string* cannot be parsed according to *format*, or if it has excess data after parsing, *ValueError* is raised. The default values used to fill in any missing data when more accurate values cannot be inferred are (1900, 1, 1, 0, 0, 0, 0, 1, -1). Both *string* and *format* must be strings.

For example:

```
>>> import time
>>> time.strptime("30 Nov 00", "%d %b %y")
time.struct_time(tm_year=2000, tm_mon=11, tm_mday=30, tm_hour=0, tm_min=0,
                 tm_sec=0, tm_wday=3, tm_yday=335, tm_isdst=-1)
```

Support for the %Z directive is based on the values contained in tzname and whether daylight is true. Because of this, it is platform-specific except for recognizing UTC and GMT which are always known (and are considered to be non-daylight savings timezones).

Only the directives specified in the documentation are supported. Because strftime() is implemented per platform it can sometimes offer more directives than those listed. But strptime() is independent of any platform and thus does not necessarily support all directives available that are not documented as supported.

class time.**struct_time**

The type of the time value sequence returned by *gmtime()*, *localtime()*, and *strptime()*. It is an object with a *named tuple* interface: values can be accessed by index and by attribute name. The following values are present:

[1] The use of %Z is now deprecated, but the %z escape that expands to the preferred hour/minute offset is not supported by all ANSI C libraries. Also, a strict reading of the original 1982 **RFC 822** standard calls for a two-digit year (%y rather than %Y), but practice moved to 4-digit years long before the year 2000. After that, **RFC 822** became obsolete and the 4-digit year has been first recommended by **RFC 1123** and then mandated by **RFC 2822**.

Index	Attribute	Values
0	tm_year	(for example, 1993)
1	tm_mon	range [1, 12]
2	tm_mday	range [1, 31]
3	tm_hour	range [0, 23]
4	tm_min	range [0, 59]
5	tm_sec	range [0, 61]; see **(2)** in *strftime()* description
6	tm_wday	range [0, 6], Monday is 0
7	tm_yday	range [1, 366]
8	tm_isdst	0, 1 or -1; see below
N/A	tm_zone	abbreviation of timezone name
N/A	tm_gmtoff	offset east of UTC in seconds

Note that unlike the C structure, the month value is a range of [1, 12], not [0, 11].

In calls to *mktime()*, tm_isdst may be set to 1 when daylight savings time is in effect, and 0 when it is not. A value of -1 indicates that this is not known, and will usually result in the correct state being filled in.

When a tuple with an incorrect length is passed to a function expecting a *struct_time*, or having elements of the wrong type, a *TypeError* is raised.

time.**time**() → *float*

Return the time in seconds since the *epoch* as a floating point number. The handling of leap seconds is platform dependent. On Windows and most Unix systems, the leap seconds are not counted towards the time in seconds since the *epoch*. This is commonly referred to as Unix time.

Note that even though the time is always returned as a floating point number, not all systems provide time with a better precision than 1 second. While this function normally returns non-decreasing values, it can return a lower value than a previous call if the system clock has been set back between the two calls.

The number returned by *time()* may be converted into a more common time format (i.e. year, month, day, hour, etc…) in UTC by passing it to *gmtime()* function or in local time by passing it to the *localtime()* function. In both cases a *struct_time* object is returned, from which the components of the calendar date may be accessed as attributes.

Use *time_ns()* to avoid the precision loss caused by the *float* type.

time.**time_ns**() → *int*

Similar to *time()* but returns time as an integer number of nanoseconds since the *epoch*.

New in version 3.7.

time.**thread_time**() → *float*

Return the value (in fractional seconds) of the sum of the system and user CPU time of the current thread. It does not include time elapsed during sleep. It is thread-specific by definition. The reference point of the returned value is undefined, so that only the difference between the results of two calls in the same thread is valid.

Use *thread_time_ns()* to avoid the precision loss caused by the *float* type.

Availability: Linux, Unix, Windows.

Unix systems supporting CLOCK_THREAD_CPUTIME_ID.

New in version 3.7.

time.**thread_time_ns**() → *int*

Similar to *thread_time()* but return time as nanoseconds.

New in version 3.7.

time.**tzset**()

Reset the time conversion rules used by the library routines. The environment variable TZ specifies how this is done. It will also set the variables tzname (from the TZ environment variable), timezone (non-DST seconds West of UTC), altzone (DST seconds west of UTC) and daylight (to 0 if this timezone does not have any daylight saving time rules, or to nonzero if there is a time, past, present or future when daylight saving time applies).

Availability: Unix.

Note: Although in many cases, changing the TZ environment variable may affect the output of functions like *localtime()* without calling *tzset()*, this behavior should not be relied on.

The TZ environment variable should contain no whitespace.

The standard format of the TZ environment variable is (whitespace added for clarity):

```
std offset [dst [offset [,start[/time], end[/time]]]]
```

Where the components are:

std and dst Three or more alphanumerics giving the timezone abbreviations. These will be propagated into time.tzname

offset The offset has the form: ± hh[:mm[:ss]]. This indicates the value added the local time to arrive at UTC. If preceded by a '-', the timezone is east of the Prime Meridian; otherwise, it is west. If no offset follows dst, summer time is assumed to be one hour ahead of standard time.

start[/time], end[/time] Indicates when to change to and back from DST. The format of the start and end dates are one of the following:

Jn The Julian day *n* (1 <= *n* <= 365). Leap days are not counted, so in all years February 28 is day 59 and March 1 is day 60.

n The zero-based Julian day (0 <= *n* <= 365). Leap days are counted, and it is possible to refer to February 29.

Mm.n.d The *d*'th day (0 <= *d* <= 6) of week *n* of month *m* of the year (1 <= *n* <= 5, 1 <= *m* <= 12, where week 5 means "the last *d* day in month *m*" which may occur in either the fourth or the fifth week). Week 1 is the first week in which the *d*'th day occurs. Day zero is a Sunday.

time has the same format as offset except that no leading sign ('-' or '+') is allowed. The default, if time is not given, is 02:00:00.

```
>>> os.environ['TZ'] = 'EST+05EDT,M4.1.0,M10.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'02:07:36 05/08/03 EDT'
>>> os.environ['TZ'] = 'AEST-10AEDT-11,M10.5.0,M3.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'16:08:12 05/08/03 AEST'
```

On many Unix systems (including *BSD, Linux, Solaris, and Darwin), it is more convenient to use the system's zoneinfo (*tzfile(5)*) database to specify the timezone rules. To do this, set the TZ environment variable to the path of the required timezone datafile, relative to the root of the systems 'zoneinfo' timezone database, usually located at /usr/share/zoneinfo. For example, 'US/Eastern', 'Australia/Melbourne', 'Egypt' or 'Europe/Amsterdam'.

```
>>> os.environ['TZ'] = 'US/Eastern'
>>> time.tzset()
>>> time.tzname
('EST', 'EDT')
```

(continues on next page)

(continued from previous page)

```
>>> os.environ['TZ'] = 'Egypt'
>>> time.tzset()
>>> time.tzname
('EET', 'EEST')
```

16.3.2 Clock ID Constants

These constants are used as parameters for *clock_getres()* and *clock_gettime()*.

time.**CLOCK_BOOTTIME**

Identical to *CLOCK_MONOTONIC*, except it also includes any time that the system is suspended.

This allows applications to get a suspend-aware monotonic clock without having to deal with the complications of *CLOCK_REALTIME*, which may have discontinuities if the time is changed using `settimeofday()` or similar.

Availability: Linux >= 2.6.39.

New in version 3.7.

time.**CLOCK_HIGHRES**

The Solaris OS has a `CLOCK_HIGHRES` timer that attempts to use an optimal hardware source, and may give close to nanosecond resolution. `CLOCK_HIGHRES` is the nonadjustable, high-resolution clock.

Availability: Solaris.

New in version 3.3.

time.**CLOCK_MONOTONIC**

Clock that cannot be set and represents monotonic time since some unspecified starting point.

Availability: Unix.

New in version 3.3.

time.**CLOCK_MONOTONIC_RAW**

Similar to *CLOCK_MONOTONIC*, but provides access to a raw hardware-based time that is not subject to NTP adjustments.

Availability: Linux >= 2.6.28, macOS >= 10.12.

New in version 3.3.

time.**CLOCK_PROCESS_CPUTIME_ID**

High-resolution per-process timer from the CPU.

Availability: Unix.

New in version 3.3.

time.**CLOCK_PROF**

High-resolution per-process timer from the CPU.

Availability: FreeBSD, NetBSD >= 7, OpenBSD.

New in version 3.7.

time.**CLOCK_TAI**

International Atomic Time

The system must have a current leap second table in order for this to give the correct answer. PTP or NTP software can maintain a leap second table.

Availability: Linux.

New in version 3.9.

time.**CLOCK_THREAD_CPUTIME_ID**
: Thread-specific CPU-time clock.

Availability: Unix.

New in version 3.3.

time.**CLOCK_UPTIME**
: Time whose absolute value is the time the system has been running and not suspended, providing accurate uptime measurement, both absolute and interval.

Availability: FreeBSD, OpenBSD >= 5.5.

New in version 3.7.

time.**CLOCK_UPTIME_RAW**
: Clock that increments monotonically, tracking the time since an arbitrary point, unaffected by frequency or time adjustments and not incremented while the system is asleep.

Availability: macOS >= 10.12.

New in version 3.8.

The following constant is the only parameter that can be sent to *clock_settime()*.

time.**CLOCK_REALTIME**
: System-wide real-time clock. Setting this clock requires appropriate privileges.

Availability: Unix.

New in version 3.3.

16.3.3 Timezone Constants

time.**altzone**
: The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if `daylight` is nonzero. See note below.

time.**daylight**
: Nonzero if a DST timezone is defined. See note below.

time.**timezone**
: The offset of the local (non-DST) timezone, in seconds west of UTC (negative in most of Western Europe, positive in the US, zero in the UK). See note below.

time.**tzname**
: A tuple of two strings: the first is the name of the local non-DST timezone, the second is the name of the local DST timezone. If no DST timezone is defined, the second string should not be used. See note below.

Note: For the above Timezone constants (*altzone*, *daylight*, *timezone*, and *tzname*), the value is determined by the timezone rules in effect at module load time or the last time *tzset()* is called and may be incorrect for times in the past. It is recommended to use the `tm_gmtoff` and `tm_zone` results from *localtime()* to obtain timezone information.

See also:

Module *datetime* More object-oriented interface to dates and times.

Module *locale* Internationalization services. The locale setting affects the interpretation of many format specifiers in *strftime()* and *strptime()*.

Module *calendar* General calendar-related functions. *timegm()* is the inverse of *gmtime()* from this module.