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. On January 1st of that year, at 0 hours, the "time since the epoch" is zero. For Unix, the epoch is 1970. To find out what the epoch is, look at `gmtime(0)`.

The functions in this module do 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 Unix, it is typically in 2038.

Year 2000 (Y2K) issues: Python depends on the platform's C library, which generally doesn't have year 2000 issues, since all dates and times are represented internally as seconds since the epoch. Functions accepting a `struct_time` (see below) generally require a 4-digit year. For backward compatibility, 2-digit years are supported if the module variable `accept2dyear` is a non-zero integer; this variable is initialized to `1` unless the environment variable **PYTHONY2K** is set to a non-empty string, in which case it is initialized to `0`. Thus, you can set **PYTHONY2K** to a non-empty string in the environment to require 4-digit years for all year input. When 2-digit years are accepted, they are converted according to the POSIX or X/Open standard: values 69-99 are mapped to 1969-1999, and values 0–68 are mapped to 2000–2068. Values 100–1899 are always illegal. Note that this is new as of Python 1.5.2(a2); earlier versions, up to Python 1.5.1 and 1.5.2a1, would add 1900 to year values below 1900.

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()`, may be considered as a sequence of 9 integers. The return values of `gmtime()`, `localtime()`, and `strptime()` also offer attribute names for individual fields.

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 **(1)** 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

Note that unlike the C structure, the month value is a range of 1-12, not 0-11. A year value will be handled as described under "Year 2000 (Y2K) issues" above. A `-1` argument as the daylight savings flag, passed to `mktime()` will usually result in the correct daylight savings state to be 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.

Changed in version 2.2: The time value sequence was changed from a tuple to a `struct_time`, with the addition of attribute names for the fields.

Use the following functions to convert between time representations:

From	To	
seconds since the epoch	struct_time in UTC	gmti
seconds since the epoch	struct_time in local time	loca
struct_time in UTC	seconds since the epoch	cale
struct_time in local time	seconds since the epoch	mkti

The module defines the following functions and data items:

`time.accept2dyear`¶

Boolean value indicating whether two-digit year values will be accepted. This is true by default, but will be set to false if the environment variable **PYTHONY2K** has been set to a non-empty string. It may also be modified at run time.

`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.

`time.asctime`([*t*])¶

Convert a tuple or `struct_time` representing a time as returned by `gmtime()` or `localtime()` to a 24-character string of the following form: `'Sun Jun 20 23:21:05 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, there is no trailing newline.

Changed in version 2.1: Allowed *t* to be omitted.

`time.clock`()¶

On Unix, return the current processor time as a floating point number expressed in seconds. The precision, and in fact the very definition of the meaning of “processor time”, depends on that of the C function of the same name, but in any case, this is the function to use for benchmarking Python or timing algorithms.

On Windows, this function returns wall-clock seconds elapsed since the first call to this function, as a floating point number, based on the Win32 function `QueryPerformanceCounter()`. The resolution is typically better than one microsecond.

`time.ctime`([*secs*])¶

Convert a time expressed in seconds since the epoch to a string representing local time. 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()`.

Changed in version 2.1: Allowed *secs* to be omitted.

Changed in version 2.4: If *secs* is `None`, the current time is used.

`time.daylight`¶

Nonzero if a DST timezone is defined.

`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.

Changed in version 2.1: Allowed *secs* to be omitted.

Changed in version 2.4: If *secs* is `None`, the current time is used.

`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.

Changed in version 2.1: Allowed *secs* to be omitted.

Changed in version 2.4: If *secs* is `None`, the current time is used.

`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.sleep`(*secs*)¶

Suspend execution for the given number of seconds. The argument may be a floating point number to indicate a more precise sleep time. The actual suspension time may be less than that requested because any caught signal will terminate the `sleep()` following execution of that signal’s catching routine. Also, the

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

`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.

Changed in version 2.1: Allowed *t* to be omitted.

Changed in version 2.4: `ValueError` raised if a field in *t* is out of range.

Changed in version 2.5: 0 is now 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 name (no characters if no time zone exists).	
%%	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`; this accounts for leap seconds and the (very rare) double leap seconds.
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.

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)`.

For example:

```
>>> import time
>>> time.strptime("30 Nov 00", "%d %b %y")   # doctest: +NORMALIZE_WHITESPACE
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.

`time.struct_time`¶

The type of the time value sequence returned by `gmtime()`, `localtime()`, and `strptime()`.

New in version 2.2.

`time.time`()¶

Return the time as a floating point number expressed in seconds since the epoch, in UTC. 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.

`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).

`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.

`time.tzset`()¶

Resets the time conversion rules used by the library routines. The environment variable **TZ** specifies how this is done.

New in version 2.3.

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:

`J`*n*

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.

`M`*m*`.`*n*`.`*d*

The *d*'th day (0 <= *d* <= 6) or 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 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')
>>> os.environ['TZ'] = 'Egypt'
>>> time.tzset()
>>> time.tzname
('EET', 'EEST')
```

See also

Module `datetime`

More object-oriented interface to dates and times.

Module `locale`

Internationalization services. The locale settings can affect the return values for some of the functions in the `time` module.

Module `calendar`

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

Footnotes

[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. The 4-digit year has been mandated by **RFC 2822**, which obsoletes **RFC 822**.

Previous topic

16.2. `io` — Core tools for working with streams

Next topic

16.4. `optparse` — More powerful command line option parser

This Page

- Show Source

Navigation

- index
- modules |
- next |
- previous |
- Python v2.6.4 documentation »
- The Python Standard Library »
- 16. Generic Operating System Services »