Chapter 13. Time Operations

A Python program can handle time in several ways. Time *intervals* are floating-point numbers in units of seconds (a fraction of a second is the fractional part of the interval): all standard library functions accepting an argument that expresses a time interval in seconds accept a float as the value of that argument. *Instants* in time are expressed in seconds since a reference instant, known as the *epoch*. (Although epochs vary per language and per platform, on all platforms, Python's epoch is midnight, UTC, January 1, 1970.) Time instants often also need to be expressed as a mixture of units of measurement (e.g., years, months, days, hours, minutes, and seconds), particularly for I/O purposes. I/O, of course, also requires the ability to format times and dates into human-readable strings, and parse them back from string formats.

The time Module

The `time` module is somewhat dependent on the underlying system's C library, which sets the range of dates that the `time` module can handle. On older Unix systems, the years 1970 and 2038 were typical cutoff points[1] (a limitation avoided by using `datetime`, discussed in the following section). Time instants are normally specified in UTC (Coordinated Universal Time, once known as GMT, or Greenwich Mean Time). The `time` module also supports local time zones and daylight savings time (DST), but only to the extent the underlying C system library does.[2]

As an alternative to seconds since the epoch, a time instant can be represented by a tuple of nine integers, called a *timetuple* (covered in **Table 13-1**.) All the items are integers: timetuples don't keep track of fractions of a second. A timetuple is an instance of `struct_time`. You can use it as a tuple; you can also, more usefully, access the items as the read-only at-

tributes *x*.tm_year, *x*.tm_mon, and so on, with the attribute names listed in **Table 13-1**. Wherever a function requires a timetuple argument, you can pass an instance of struct_time or any other sequence whose items are nine integers in the right ranges (all ranges in the table include both lower and upper bounds, both inclusive).

Table 13-1. Tuple form of time representation

Item	Meaning	Field name	Range	Notes
0	Year	tm_year	1970–2038	0001–9999 on some platforms
1	Month	tm_mon	1–12	1 is January; 12 is December
2	Day	tm_mday	1–31	
3	Hour	tm_hour	0–23	0 is midnight; 12 is noon
4	Minute	tm_min	0–59	
5	Second	tm_sec	0–61	60 and 61 for leap seconds
6	Weekday	tm_wday	0–6	0 is Monday; 6 is Sunday
7	Year day	tm_yday	1–366	Day number within the year
8	DST flag	tm_isdst	-1–1	-1 means the library determines DST

To translate a time instant from a “seconds since the epoch” floating-point value into a timetuple, pass the floating-point value to a function (e.g., `localtime`) that returns a timetuple with all nine items valid. When you convert in the other direction, `mktime` ignores redundant items `6` (`tm_wday`) and `7` (`tm_yday`) of the tuple. In this case, you normally set item `8` (`tm_isdst`) to `-1` so that `mktime` itself determines whether to apply DST.

`time` supplies the functions and attributes listed in **Table 13-2**.

Table 13-2. Functions and attributes of the *time* module

`asctime`	`asctime([tupletime])` Accepts a timetuple and returns a readable 24-characte `'Sun Jan 8 14:41:06 2017'`. Calling `asctime()` withou is like calling `asctime(time.localtime())` (formats cu local time).
`ctime`	`ctime([secs])` Like `asctime(localtime(secs))`, accepts an instant ex seconds since the epoch and returns a readable 24-cha form of that instant, in local time. Calling `ctime()` with arguments is like calling `asctime()` (formats current ti time).
`gmtime`	`gmtime([secs])` Accepts an instant expressed in seconds since the epoc returns a timetuple *t* with the UTC time (*t*`.tm_isdst` is Calling `gmtime()` without arguments is like calling `gmti` (returns the timetuple for the current time instant).

`localtime`	`localtime([`*`secs`*`])` Accepts an instant expressed in seconds since the epoc returns a timetuple *`t`* with the local time (`t.tm_isdst` i depending on whether DST applies to instant *`secs`* by l Calling `localtime()` without arguments is like calling `localtime(time())` (returns the timetuple for the curr instant).
`mktime`	`mktime(`*`tupletime`*`)` Accepts an instant expressed as a timetuple in local tim returns a floating-point value with the instant expresse since the epoch (only accepts the limited epoch dates b 1970–2038, not the extended range, even on 64-bit mac DST flag, the last item in *`tupletime`*, is meaningful: set standard time, to `1` to get DST, or to `-1` to let `mktime` con whether DST is in effect at the given instant.
`monotonic`	`monotonic()` Like `time()`, returns the current time instant, a `float` v since the epoch; however, the time value is guaranteed backward between calls, even when the system clock is (e.g., due to leap seconds or at the moment of switching DST).
`perf_counter`	`perf_counter()` For determining elapsed time between successive calls stopwatch), `perf_counter` returns a time value in fract seconds using the highest-resolution clock available to for short durations. It is system-wide and *includes* time during `sleep`. Use only the difference between successi there is no defined reference point.
`process_time`	`process_time()` Like `perf_counter`; however, the returned time value i wide and *doesn't* include time elapsed during `sleep`. Us

	difference between successive calls, as there is no defi reference point.
`sleep`	`sleep(`*`secs`*`)` Suspends the calling thread for *`secs`* seconds. The calli may start executing again before *`secs`* seconds (when i thread and some signal wakes it up) or after a longer s (depending on system scheduling of processes and thre can call sleep with *`secs`* set to `0` to offer other threads a run, incurring no significant delay if the current threa one ready to run.
`strftime`	`strftime(`*`fmt`*`[,` *`tupletime`*`])` Accepts an instant expressed as a timetuple in local tim returns a string representing the instant as specified b If you omit *`tupletime`*, `strftime` uses `localtime(time(` the current time instant). The syntax of *`fmt`* is similar t covered in **"Legacy String Formatting with %"**, thoug conversion characters are different, as shown in **Table** to the time instant specified by *`tupletime`*; the format c width and precision. For example, you can obtain dates just as formatted by (e.g., `'Tue Dec 10 18:07:14 2002'`) with the format st `%d %H:%M:%S %Y'`. You can obtain dates compliant with RFC 822 (e.g., `'Tue 2002 18:07:14 EST'`) with the format string `'%a, %d %` `%H:%M:%S %Z'`. These strings can also be used for `datetime` formatting mechanisms discussed in **"Formatting of User-Coded** allowing you to equivalently write, for a `datetime.dat` `d`, either `f'{d:%Y/%m/%d}'` or `'{:%Y/%m/%d}'.format(d)` which give a result such as '2022/04/17'. For ISO 8601–f `datetimes`, see the `isoformat()` and `fromisoformat()` covered in **"The date Class"**.

`strptime`	`strptime(str, fmt)`
	Parses *`str`* according to format string *`fmt`* (a string such
	`%d %H:%M:%S %Y'`, as covered in the discussion of `strft`
	returns the instant as a timetuple. If no time values are
	defaults to midnight. If no date values are provided, de
	January 1, 1900. For example:
	
	`>>> print(time.strptime("Sep 20, 2022", '%b %`
	
	`time.struct_time(tm_year=2022, tm_mon=9, tm_m`
	`tm_hour=0, tm_min=0, tm_sec=0, tm_wday=1,`
	`tm_yday=263, tm_isdst=-1)`

`time`	`time()` Returns the current time instant, a `float` with seconds epoch. On some (mostly older) platforms, the precision is as low as one second. May return a lower value in a s call if the system clock is adjusted backward between c due to leap seconds).
`timezone`	The offset in seconds of the local time zone (without DS (`<0` in the Americas; `>=0` in most of Europe, Asia, and A
`tzname`	A pair of locale-dependent strings, which are the name time zone without and with DST, respectively.

a `mktime`'s result's fractional part is always `0`, since its timetuple argument does r
for fractions of a second.

Table 13-3. Conversion characters for `strftime`

Type char	Meaning	Special notes
`a`	Weekday name, abbreviated	Depends on locale
`A`	Weekday name, full	Depends on locale
`b`	Month name, abbreviated	Depends on locale
`B`	Month name, full	Depends on locale
`c`	Complete date and time representation	Depends on locale
`d`	Day of the month	Between 1 and 31
`f`	Microsecond as decimal, zero-padded to six digits	One to six digits
`G`	ISO 8601:2000 standard week-based year number	
`H`	Hour (24-hour clock)	Between 0 and 23
`I`	Hour (12-hour clock)	Between 1 and 12
`j`	Day of the year	Between 1 and 366
`m`	Month number	Between 1 and 12
`M`	Minute number	Between 0 and 59
`p`	A.M. or P.M. equivalent	Depends on locale

Type char	Meaning	Special notes
`S`	Second number	Between 0 and 61
`u`	Day of week	Monday is 1, up to 7
`U`	Week number (Sunday first weekday)	Between 0 and 53
`V`	ISO 8601:2000 standard week-based week number	
`w`	Weekday number	0 is Sunday, up to 6
`W`	Week number (Monday first weekday)	Between 0 and 53
`x`	Complete date representation	Depends on locale
`X`	Complete time representation	Depends on locale
`y`	Year number within century	Between 0 and 99
`Y`	Year number	1970 to 2038, or wider
`z`	UTC offset as a string: ±HHMM[SS[.ffffff]]	
`Z`	Name of time zone	Empty if no time zone exists
`%`	A literal % character	Encoded as %%

The datetime Module

`datetime` provides classes for modeling date and time objects, which can be either *aware* of time zones or *naive* (the default). The class `tzinfo`, whose instances model a time zone, is abstract: the `datetime` module supplies only one simple implementation, `datetime.timezone` (for all the gory details, see the **online docs**). The `zoneinfo` module, covered in the following section, offers a richer concrete implementation of `tzinfo`, which lets you easily create time zone-aware `datetime` objects. All types in `datetime` have immutable instances: attributes are read-only, instances can be keys in a `dict` or items in a `set`, and all functions and methods return new objects, never altering objects passed as arguments.

The date Class

Instances of the `date` class represent a date (no time of day in particular within that date) between `date.min <= d <= date.max`, are always naive, and assume the Gregorian calendar was always in effect. `date` instances have three read-only integer attributes: *`year`*, *`month`*, and *`day`*. The constructor for this class has the signature:

`date` **`class`** `date(`*`year, month, day`*`)`
: Returns a date object for the given *`year`*, *`month`*, and *`day`* arguments, in the valid ranges `1 <=` *`year`* `<= 9999`, `1 <=` *`month`* `<= 12`, and `1 <=` *`day`* `<=` *`n`*, where *`n`* is the number of days for the given month and year. Raises `ValueError` if invalid values are given.

The `date` class also supplies three class methods usable as alternative constructors, listed in **Table 13-4**.

Table 13-4. Alternative `date` constructors

`fromordinal` `date.fromordinal(`*`ordinal`*`)`
: Returns a `date` object corresponding to the

	proleptic Gregorian ordinal *ordinal*, where a value of 1 corresponds to the first day of year 1 CE.
`fromtimestamp`	`date.fromtimestamp(`*timestamp*`)` Returns a `date` object corresponding to the instant *timestamp* expressed in seconds since the epoch.
`today`	`date.today()` Returns a `date` representing today's date.

Instances of the `date` class support some arithmetic. The difference between `date` instances is a `timedelta` instance; you can add or subtract a `timedelta` to or from a `date` instance to make another `date` instance. You can also compare any two instances of the `date` class (the later one is greater).

An instance *d* of the class `date` supplies the methods listed in **Table 13-5**.

Table 13-5. Methods of an instance `d` of class `date`

`ctime`	*d*`.ctime()` Returns a string representing the date *d* in the same 24-character format as `time.ctime` (with the time of day set to 00:00:00, midnight).
`isocalendar`	*d*`.isocalendar()` Returns a tuple with three integers (ISO year, ISO week number, and ISO weekday). See the **ISO 8601 standard** for more details about the ISO (International Standards Organization) calendar.
`isoformat`	*d*`.isoformat()` Returns a string representing date *d* in the format `'YYYY-MM-DD'`; same as `str(`*d*`)`.
`isoweekday`	*d*`.isoweekday()` Returns the day of the week of date *d* as an integer, 1

	for Monday through `7` for Sunday; like `d.weekday() + 1`.
`replace`	*d*`.replace(year=`**`None`**`, month=`**`None`**`, day=`**`None`**`)` Returns a new `date` object, like *d* except for those attributes explicitly specified as arguments, which get replaced. For example: `date(x,y,z).replace(month=m) == date(x,m,z)`
`strftime`	*d*`.strftime(`*fmt*`)` Returns a string representing date *d* as specified by string *fmt*, like: `time.strftime(fmt, d.timetuple())`
`timetuple`	*d*`.timetuple()` Returns a timetuple corresponding to date *d* at time `00:00:00` (midnight).
`toordinal`	*d*`.toordinal()` Returns the proleptic Gregorian ordinal for date *d*. For example: `date(1,1,1).toordinal() == 1`
`weekday`	*d*`.weekday()` Returns the day of the week of date *d* as an integer, `0`

for Monday through 6 for Sunday; like `d.isoweekday() - 1`.

The time Class

Instances of the `time` class represent a time of day (of no particular date), may be naive or aware regarding time zones, and always ignore leap seconds. They have five attributes: four read-only integers (`hour`, `minute`, `second`, and `microsecond`) and an optional read-only `tzinfo` (**None** for naive instances). The constructor for the `time` class has the signature:

`time`
: **class** `time(hour=0, minute=0, second=0, microsecond=0, tzinfo=`**None**`)`
Instances of the class `time` do not support arithmetic. You can compare two instances of `time` (the one that's later in the day is greater), but only if they are either both aware or both naive.

An instance *t* of the class `time` supplies the methods listed in **Table 13-6**.

Table 13-6. Methods of an instance `t` of class `time`

`isoformat`
: `t.isoformat()`
Returns a string representing time *t* in the format `'HH:MM:SS'`; same as `str(t)`. If `t.microsecond != 0`, the resulting string is longer: `'HH:MM:SS.mmmmmm'`. If *t* is aware, six more characters, `'+HH:MM'`, are added at the end to represent the time zone's offset from UTC. In other words, this formatting operation follows the **ISO 8601 standard**.

`replace`
: `t.replace(hour=`**None**`, minute=`**None**`, second=`**None**`, microsecond=`**None**`[, tzinfo])`
Returns a new `time` object, like *t* except for those

attributes explicitly specified as arguments, which get replaced. For example:

```
time(x,y,z).replace(minute=m) == time(x,m,z)
```

`strftime` `t.strftime(fmt)`
Returns a string representing time `t` as specified by the string `fmt`.

An instance `t` of the class `time` also supplies methods `dst`, `tzname`, and `utcoffset`, which accept no arguments and delegate to `t.tzinfo`, returning **`None`** when `t.tzinfo` is **`None`**.

The datetime Class

Instances of the `datetime` class represent an instant (a date, with a specific time of day within that date), may be naive or aware of time zones, and always ignore leap seconds. `datetime` extends `date` and adds `time`'s attributes; its instances have read-only integer attributes `year`, `month`, `day`, `hour`, `minute`, `second`, and `microsecond`, and an optional `tzinfo` attribute (**`None`** for naive instances). In addition, `datetime` instances have a read-only `fold` attribute to distinguish between ambiguous timestamps during a rollback of the clock (such as the "fall back" at the end of daylight savings time, which creates duplicate naive times between 1 A.M. and 2 A.M.). `fold` has the value `0` or `1` `0` corresponds to the time *before* the rollback; `1` to the time *after* the rollback.

Instances of `datetime` support some arithmetic: the difference between `datetime` instances (both aware, or both naive) is a `timedelta` instance, and you can add or subtract a `timedelta` instance to or from a `datetime` instance to construct another `datetime` instance. You can compare two instances of the `datetime` class (the later one is greater) as long as they're both aware or both naive. The constructor for this class has the signature:

`datetime`	**`class`** `datetime(`*`year`*`, `*`month`*`, `*`day`*`, hour=0, minute=0, second=0, microsecond=0, tzinfo=`**`None`**`, *, fold=0)` Returns a `datetime` object following similar constraints as the `date` class constructor. `fold` is an `int` with the value `0` or `1`, as described previously.

`datetime` also supplies some class methods usable as alternative constructors, covered in **Table 13-7**.

Table 13-7. Alternative `datetime` constructors

`combine`	`datetime.combine(`*`date`*`, `*`time`*`)`
	Returns a `datetime` object with the date attributes taken from *`date`* and the time attributes (including `tzinfo`) taken from *`time`*. `datetime.combine(`*`d, t`*`)` is like:
	`datetime(d.year, d.month, d.day, t.hour, t.minute, t.second, t.microsecond, t.tzinfo)`

`fromordinal`	`datetime.fromordinal(`*`ordinal`*`)` Returns a `datetime` object for the date given proleptic Gregorian ordinal *`ordinal`*, where a value of `1` means the first day of year 1 CE, at midnight.
`fromtimestamp`	`datetime.fromtimestamp(`*`timestamp`*`, tz=`**`None`**`)` Returns a `datetime` object corresponding to the instant *`timestamp`* expressed in seconds since the epoch, in local time. When `tz` is not **`None`**, returns an aware `datetime` object with the given `tzinfo` instance `tz`.

`now`	`datetime.now(tz=None)` Returns a naive `datetime` object for the current local date and time. When `tz` is not **`None`**, returns an aware `datetime` object with the given `tzinfo` instance `tz`.
`strptime`	`datetime.strptime(str, fmt)` Returns a `datetime` representing *`str`* as specified by string *`fmt`*. When `%z` is present in *`fmt`*, the resulting `datetime` object is time zone–aware.
`today`	`datetime.today()` Returns a naive `datetime` object representing the current local date and time; same as the `now` class method but does not accept optional argument *`tz`*.
`utcfromtimestamp`	`datetime.utcfromtimestamp(timestamp)` Returns a naive `datetime` object corresponding to the instant *`timestamp`* expressed in seconds since the epoch, in UTC.
`utcnow`	`datetime.utcnow()` Returns a naive `datetime` object representing the current date and time, in UTC.

An instance *`d`* of `datetime` also supplies the methods listed in **Table 13-8**.

Table 13-8. Methods of an instance `d` of `datetime`

`astimezone`	*`d`*`.astimezone(`*`tz`*`)` Returns a new aware `datetime` object, like *`d`*, except that date and time are converted along with the time zone to one in `tzinfo` object *`tz`*.[a] *`d`* must be aware, to avoid pote bugs. Passing a naive *`d`* may lead to unexpected results.
`ctime`	*`d`*`.ctime()` Returns a string representing date and time *`d`* in the sam

	character format as `time.ctime`.
`date`	*d*`.date()` Returns a date object representing the same date as *d*.
`isocalendar`	*d*`.isocalendar()` Returns a tuple with three integers (ISO year, ISO week number, and ISO weekday) for *d*'s date.
`isoformat`	*d*`.isoformat(sep='T')` Returns a string representing *d* in the format `'YYYY-MM-DDxHH:MM:SS'`, where *x* is the value of argument `sep` (mu string of length 1). If *d*`.microsecond != 0`, seven charac `'.mmmmmm'`, are added after the `'SS'` part of the string. If aware, six more characters, `'+HH:MM'`, are added at the e represent the time zone's offset from UTC. In other word formatting operation follows the ISO 8601 standard. `str` the same as *d*`.isoformat(sep=' ')`.
`isoweekday`	*d*`.isoweekday()` Returns the day of the week of *d*'s date as an integer, `1` fo Monday through `7` for Sunday.

`replace`	*d*`.replace(year=`**`None`**`, month=`**`None`**`, day=`**`None`**`, hour=`**`No`**
	`minute=`**`None`**`, second=`**`None`**`, microsecond=`**`None`**`,`
	`tzinfo=`**`None`**`,*, fold=0)`
	Returns a new `datetime` object, like *d* except for those attributes specified as arguments, which get replaced (b does *no* time zone conversion—use `astimezone` if you w time converted). You can also use `replace` to create an a `datetime` object from a naive one. For example:
	
	`# create datetime replacing just month with no`
	`# other changes (== datetime(x,m,z))`
	`datetime(x,y,z).replace(month=m)`

```
# create aware datetime from naive datetime.nc
d = datetime.now().replace(tzinfo=ZoneInfo(
                    "US/Pacific"))
```

`strftime`	`d.strftime(fmt)` Returns a string representing *d* as specified by the forma string *fmt*.
`time`	`d.time()` Returns a naive `time` object representing the same time as *d*.
`timestamp`	`d.timestamp()` Returns a float with the seconds since the epoch. Naive instances are assumed to be in the local time zone.
`timetuple`	`d.timetuple()` Returns a timetuple corresponding to instant *d*.
`timetz`	`d.timetz()` Returns a `time` object representing the same time of day with the same `tzinfo`.

`toordinal`	`d.toordinal()`
	Returns the proleptic Gregorian ordinal for *d*'s date. For example:
	`datetime(1, 1, 1).toordinal() == 1`

`utctime tuple`	`d.utctimetuple()` Returns a timetuple corresponding to instant *d*, normali UTC if *d* is aware.
`weekday`	`d.weekday()` Returns the day of the week of *d*'s date as an integer, `0` f Monday through `6` for Sunday.

a Note that *d*.astimezone(*tz*) is quite different from *d*.replace(tzinfo=*tz*): repl does no time zone conversion, but rather just copies all of *d*'s attributes except *d*.tzinfo.

An instance *d* of the class datetime also supplies the methods dst, tzname, and utcoffset, which accept no arguments and delegate to *d*.tzinfo, returning **None** when *d*.tzinfo is **None** (i.e., when *d* is naive).

The timedelta Class

Instances of the timedelta class represent time intervals with three read-only integer attributes: days, seconds, and microseconds. The constructor for this class has the signature:

timedelta — timedelta(days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0)
Converts all units with the obvious factors (a week is 7 days, an hour is 3,600 seconds, and so on) and normalizes everything to the three integer attributes, ensuring that 0 <= seconds < 24 * 60 * 60 and 0 <= microseconds < 1000000. For example:

```
>>> print(repr(timedelta(minutes=0.5)))
```

```
datetime.timedelta(days=0, seconds=30)
```

```
>>> print(repr(timedelta(minutes=-0.5)))
```

```
datetime.timedelta(days=-1, seconds=86370)
```

Instances of `timedelta` support arithmetic: `+` and `-` between themselves and with instances of the classes `date` and `datetime`; `*` with integers; `/` with integers and `timedelta` instances (floor division, true division, `divmod`, `%`). They also support comparisons between themselves.

While `timedelta` instances can be created using this constructor, they are more often created by subtracting two `date`, `time`, or `datetime` instances, such that the resulting `timedelta` represents an elapsed time period. An instance *`td`* of `timedelta` supplies a method *`td`*`.total_seconds()` that returns a float representing the total seconds of a `timedelta` instance.

The tzinfo Abstract Class

The `tzinfo` class defines the abstract class methods listed in **Table 13-9**, to support creation and usage of aware `datetime` and `time` objects.

Table 13-9. Methods of the `tzinfo` class

`dst`	`dst(`*`dt`*`)` Returns the daylight savings offset of a given `datetime`, as a `timedelta` object
`tzname`	`tzname(`*`dt`*`)` Returns the abbreviation for the time zone of a given `datetime`
`utcoffset`	`utcoffset(`*`dt`*`)` Returns the offset from UTC of a given `datetime`, as a `timedelta` object

`tzinfo` also defines a `fromutc` abstract instance method, primarily for internal use by the `datetime.astimezone` method.

The timezone Class

The `timezone` class is a concrete implementation of the `tzinfo` class. You construct a `timezone` instance using a `timedelta` representing the time offset from UTC. `timezone` supplies one class property, `utc`, a `timezone` representing the UTC time zone (equivalent to `timezone(timedelta(0))`).

The zoneinfo Module

3.9+ The `zoneinfo` module is a concrete implementation of timezones for use with datetime's `tzinfo`.[3] `zoneinfo` uses the system's time zone data by default, with **`tzdata`** as a fallback. (On Windows, you may need to **`pip install tzdata`**; once installed, you don't import `tzdata` in your program—rather, `zoneinfo` uses it automatically.)

`zoneinfo` provides one class: `ZoneInfo`, a concrete implementation of the `datetime.tzinfo` abstract class. You can assign it to `tzinfo` during construction of an aware `datetime` instance, or use it with the `datetime.replace` or `datetime.astimezone` methods. To construct a `ZoneInfo`, use one of the defined IANA time zone names, such as `"America/Los_Angeles"` or `"Asia/Tokyo"`. You can get a list of these time zone names by calling `zoneinfo.available_timezones()`. More details on each time zone (such as offset from UTC and daylight savings information) can be found **on Wikipedia**.

Here are some examples using `ZoneInfo`. We'll start by getting the current local date and time in California:

```
>>> from datetime import datetime
>>> from zoneinfo import ZoneInfo
>>> d=datetime.now(tz=ZoneInfo("America/Los_Angeles"))
>>> d
```

```
datetime.datetime(2021,10,21,16,32,23,96782,tzinfo=zoneinfo.ZoneInfo(key
```

```
='America/Los_Angeles'))
```

We can now update the time zone to a different one *without* changing other attributes (i.e., without converting the time to the new time zone):

```
>>> dny=d.replace(tzinfo=ZoneInfo("America/New_York"))
>>> dny
```

```
datetime.datetime(2021,10,21,16,32,23,96782,tzinfo=zoneinfo.ZoneInfo(key
='America/New_York'))
```

Convert a `datetime` instance to UTC:

```
>>> dutc=d.astimezone(tz=ZoneInfo("UTC"))
>>> dutc
```

```
datetime.datetime(2021,10,21,23,32,23,96782,tzinfo=zoneinfo.ZoneInfo(key
='UTC'))
```

Get an *aware* timestamp of the current time in UTC:

```
>>> daware=datetime.utcnow().replace(tzinfo=ZoneInfo("UTC"))
>>> daware
```

```
datetime.datetime(2021,10,21,23,32,23,96782,tzinfo=zoneinfo.ZoneInfo(key
='UTC'))
```

Display the `datetime` instance in a different time zone:

```
>>> dutc.astimezone(ZoneInfo("Asia/Katmandu")) # offset +5h 45m
```

```
datetime.datetime(2021,10,22,5,17,23,96782,tzinfo=zoneinfo.ZoneInfo(key
='Asia/Katmandu'))
```

Get the local time zone:

```
>>> tz_local=datetime.now().astimezone().tzinfo
>>> tz_local
```

```
datetime.timezone(datetime.timedelta(days=-1, seconds=61200), 'Pacific
Daylight Time')
```

Convert the UTC `datetime` instance back into the local time zone:

```
>>> dt_loc=dutc.astimezone(tz_local)
>>> dt_loc
```

```
datetime.datetime(2021, 10, 21, 16, 32, 23, 96782, tzinfo=datetime.time
(datetime.timedelta(days=-1, seconds=61200), 'Pacific Daylight Time'))
```

```
>>> d==dt_local
```

```
True
```

And get a sorted list of all available time zones:

```
>>> tz_list=zoneinfo.available_timezones()
>>> sorted(tz_list)[0],sorted(tz_list)[-1]
```

```
('Africa/Abidjan', 'Zulu')
```

ALWAYS USE THE UTC TIME ZONE INTERNALLY

The best way to program around the traps and pitfalls of time zones is to always use the UTC time zone internally, converting from other time zones on input, and use `datetime.astimezone` only for display purposes.

This tip applies even if your application runs only in your own location, with no intention of ever using time data from other time zones. If your application runs continuously for days or weeks at a time, and the time zone configured for your system observes daylight savings time, you *will* run into time zone-related issues if you don't work in UTC internally.

The dateutil Module

The third-party package **`dateutil`** (which you can install with **`pip install python-dateutil`**) offers modules to manipulate dates in many ways. **Table 13-10** lists the main modules it provides, in addition to those for time zone–related operations (now best performed with `zoneinfo`, discussed in the previous section).

Table 13-10. `dateutil` modules

`easter`	`easter.easter(`*`year`*`)` Returns the `datetime.date` object for Easter of the giv *`year`*. For example:

```
>>> from dateutil import easter
>>> print(easter.easter(2023))
```

```
2023-04-09
```

`parser`	`parser.parse(s)`
	Returns the `datetime.datetime` object denoted by str
	with very permissive (or "fuzzy") parsing rules. For
	example:
	
	`>>> from dateutil import parser`
	`>>> print(parser.parse('Saturday, January 28`
	`' 2006, at 11:15pm'))`
	
	`2006-01-28 23:15:00`

`relativedelta`	`relativedelta.relativedelta(...)` Provides, among other things, an easy way to find "ne Monday," "last year," etc. `dateutil`'s **docs** offer detail explanations of the rules defining the inevitably complicated behavior of `relativedelta` instances.
`rrule`	`rrule.rrule(freq, ...)` Implements **RFC 2445** (also known as the `iCalendar` F in all the glory of its 140+ pages. `rrule` allows you to d with recurring events, providing such methods as `aft` `before`, `between`, and `count`.

See the **documentation** for complete details on the `dateutil` module's rich functionality.

The sched Module

The `sched` module implements an event scheduler, letting you easily deal with events that may be scheduled in either a "real" or a "simulated" time scale. This event scheduler is safe to use in single and multithreaded environments. `sched` supplies a `scheduler` class that takes two optional arguments, `timefunc` and `delayfunc`.

`scheduler` **`class`** `scheduler(timefunc=time.monotonic, delayfunc=time.sleep)`
The optional argument `timefunc` must be callable without arguments to get the current time instant (in any unit of measure); for example, you can pass `time.time`. The optional `delayfunc` is callable with one argument (a time duration, in the same units as `timefunc`) to delay the current thread for that time. `scheduler` calls `delayfunc(0)` after each event to give other threads a chance; this is compatible with `time.sleep`. By taking functions as arguments, `scheduler` lets you use whatever "simulated time" or "pseudotime" fits your application's needs[a].
If monotonic time (time that cannot go backward even if the system clock is adjusted backward between calls, e.g., due to leap seconds) is critical to your application, use the default `time.monotonic` for your scheduler.

[a] A great example of the **dependency injection design pattern** for purposes not necessarily related to testing.

A `scheduler` instance *`s`* supplies the methods detailed in **Table 13-11**.

Table 13-11. Methods of an instance `s` of `scheduler`

`cancel` *`s`*`.cancel(`*`event_token`*`)`
Removes an event from *`s`*'s queue. *`event_token`* must be the

result of a previous call to `s.enter` or `s.enterabs`, and the event must not yet have happened; otherwise, `cancel` raises `RuntimeError`.

`empty` — `s.empty()`

Returns **`True`** when *`s`*'s queue is currently empty; otherwise, returns **`False`**.

`enter` — `s.enter(delay, priority, func, argument=(), kwargs={})`

Like `enterabs`, except that *`delay`* is a relative time (a positive difference forward from the current instant), while `enterabs` argument *`when`* is an absolute time (a future instant). To schedule an event for *repeated* execution, use a little wrapper function; for example:

```
def enter_repeat(s, first_delay, period, priority,
        func, args):
    def repeating_wrapper():
        s.enter(period, priority,
                repeating_wrapper, ())
        func(*args)
    s.enter(first_delay, priority,
        repeating_wrapper, args)
```

`enterabs` — `s.enterabs(when, priority, func, argument=(), kwargs=`

Schedules a future event (a callback to *`func`*(*`args`*, *`kwargs`*) at time *`when`*. *`when`* is in the units used by the time functions of *`s`*. Should several events be scheduled for the same time, *`s`* executes them in increasing order of *`priority`*. `enterabs` returns an event token *`t`*, which you may later pass to `s.cancel` to cancel this event.

`run` — `s.run(blocking=True)`

Runs scheduled events. If `blocking` is **`True`**, `s.run` loops until `s.empty` returns **`True`**, using the `delayfunc` passed on *`s`*'s

initialization to wait for each scheduled event. If `blocking` i
`False`, executes any soon-to-expire events, then returns the
next event's deadline (if any). When a callback *`func`* raises a
exception, *`s`* propagates it, but *`s`* keeps its own state, removi
the event from the schedule. If a callback *`func`* runs longer t
the time available before the next scheduled event, *`s`* falls
behind but keeps executing scheduled events in order, neve
dropping any. Call `s.cancel` to drop an event explicitly if th
event is no longer of interest.

The calendar Module

The `calendar` module supplies calendar-related functions, including functions to print a text calendar for a given month or year. By default, `calendar` takes Monday as the first day of the week and Sunday as the last one. To change this, call `calendar.setfirstweekday`. `calendar` handles years in module `time`'s range, typically (at least) 1970 to 2038.

The `calendar` module supplies the functions listed in **Table 13-12**.

Table 13-12. Functions of the `calendar` module

`calendar`	`calendar(`*`year`*`, w=2, li=1, c=6)` Returns a multiline string with a calendar for year *`year`* formatted into three columns separated by `c` spaces. `w` is the width in characters of each date; each line has length `21*w+18+2*c`. `li` is the number of lines for each week.
`firstweekday`	`firstweekday()` Returns the current setting for the weekday that starts each week. By default, when `calendar` is first imported, this is `0` (meaning Monday).

`isleap`	`isleap(`*`year`*`)` Returns **`True`** if *`year`* is a leap year; otherwise, returns **`False`**.
`leapdays`	`leapdays(`*`y1`*`, `*`y2`*`)` Returns the total number of leap days in the years within `range(`*`y1`*`, `*`y2`*`)` (remember, this means that *`y2`* is excluded).
`month`	`month(`*`year`*`, `*`month`*`, w=2, li=1)` Returns a multiline string with a calendar for month *`month`* of year *`year`*, one line per week plus two header lines. `w` is the width in characters of each date; each line has length `7*w+6`. `li` is the number of lines for each week.
`monthcalendar`	`monthcalendar(`*`year`*`, `*`month`*`)` Returns a list of lists of `int`s. Each sublist denotes a week. Days outside month *`month`* of year *`year`* are set to `0`; days within the month are set to their day of month, `1` and up.
`monthrange`	`monthrange(`*`year`*`, `*`month`*`)` Returns two integers. The first one is the code of the weekday for the first day of the month *`month`* in year *`year`*; the second one is the number of days in the month. Weekday codes are `0` (Monday) to `6` (Sunday); month numbers are `1` to `12`.
`prcal`	`prcal(`*`year`*`, w=2, li=1, c=6)` Like `print(calendar.calendar(`*`year`*`, `*`w`*`, `*`li`*`, `*`c`*`))`.

`prmonth`	`prmonth(`*`year`*`, `*`month`*`, w=2, li=1)` Like `print(calendar.month(`*`year`*`, `*`month`*`, `*`w`*`, `*`li`*`))`.
`setfirstweekday`	`setfirstweekday(`*`weekday`*`)` Sets the first day of each week to weekday code *`weekday`*. Weekday codes are `0` (Monday) to `6` (Sunday). `calendar` supplies the attributes `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY`, and `SUNDAY`, whose values are the integers `0` to `6`. Use these attributes when you mean weekdays (e.g., `calendar.FRIDAY` instead of `4`) to make your code clearer and more readable.
`timegm`	`timegm(`*`tupletime`*`)` Just like `time.mktime`: accepts a time instant in timetuple form and returns that instant as a `float` number of seconds since the epoch.
`weekday`	`weekday(`*`year`*`, `*`month`*`, `*`day`*`)` Returns the weekday code for the given date. Weekday codes are `0` (Monday) to `6` (Sunday); month numbers are `1` (January) to `12` (December).

`python -m calendar` offers a useful command-line interface to the module's functionality: run **`python -m calendar -h`** to get a brief help message.

1 On older Unix systems, 1970-01-01 is the start of the epoch, and 2038-01-19 is when 32-bit time wraps back to the epoch. Most modern systems now use 64-bit time, and many `time` methods can accept a year from 0001 to 9999, but some methods, or old systems (especially embedded ones), may still be limited.

2 `time` and `datetime` don't account for leap seconds, since their schedule is not known for the future.

3 Pre-3.9, use instead the third-party module **`pytz`**.