9.1. `datetime` — Basic date and time types¶

New in version 2.3.

The `datetime` module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. For related functionality, see also the `time` and `calendar` modules.

There are two kinds of date and time objects: "naive" and "aware". This distinction refers to whether the object has any notion of time zone, daylight saving time, or other kind of algorithmic or political time adjustment. Whether a naive `datetime` object represents Coordinated Universal Time (UTC), local time, or time in some other timezone is purely up to the program, just like it's up to the program whether a particular number represents metres, miles, or mass. Naive `datetime` objects are easy to understand and to work with, at the cost of ignoring some aspects of reality.

For applications requiring more, `datetime` and `time` objects have an optional time zone information member, `tzinfo`, that can contain an instance of a subclass of the abstract `tzinfo` class. These `tzinfo` objects capture information about the offset from UTC time, the time zone name, and whether Daylight Saving Time is in effect. Note that no concrete `tzinfo` classes are supplied by the `datetime` module. Supporting timezones at whatever level of detail is required is up to the application. The rules for time adjustment across the world are more political than rational, and there is no standard suitable for every application.

The `datetime` module exports the following constants:

`datetime.MINYEAR`¶

The smallest year number allowed in a `date` or `datetime` object. `MINYEAR` is `1`.

`datetime.MAXYEAR`¶

The largest year number allowed in a `date` or `datetime` object. `MAXYEAR` is `9999`.

See also

Module `calendar`

General calendar related functions.

Module `time`

Time access and conversions.

9.1.1. Available Types¶

class `datetime.date`

An idealized naive date, assuming the current Gregorian calendar always was, and always will be, in effect. Attributes: `year`, `month`, and `day`.

class `datetime.time`

An idealized time, independent of any particular day, assuming that every day has exactly 24*60*60 seconds (there is no notion of "leap seconds" here). Attributes: `hour`, `minute`, `second`, `microsecond`, and `tzinfo`.

class `datetime.datetime`

A combination of a date and a time. Attributes: `year`, `month`, `day`, `hour`, `minute`, `second`, `microsecond`, and `tzinfo`.

class `datetime.timedelta`

A duration expressing the difference between two `date`, `time`, or `datetime` instances to microsecond resolution.

class `datetime.tzinfo`¶

An abstract base class for time zone information objects. These are used by the `datetime` and `time` classes to provide a customizable notion of time adjustment (for example, to account for time zone and/or daylight saving time).

Objects of these types are immutable.

Objects of the `date` type are always naive.

An object *d* of type `time` or `datetime` may be naive or aware. *d* is aware if `d.tzinfo` is not `None` and `d.tzinfo.utcoffset(d)` does not return `None`. If `d.tzinfo` is `None`, or if `d.tzinfo` is not `None` but `d.tzinfo.utcoffset(d)` returns `None`, *d* is naive.

The distinction between naive and aware doesn't apply to `timedelta` objects.

Subclass relationships:

```
object
   timedelta
   tzinfo
   time
   date
       datetime
```

9.1.2. `timedelta` Objects¶

A `timedelta` object represents a duration, the difference between two dates or times.

class `datetime.timedelta`([*days*[, *seconds*[, *microseconds*[, *milliseconds*[, *minutes*[, *hours*[, *weeks*]]]]]]])¶

All arguments are optional and default to `0`. Arguments may be ints, longs, or floats, and may be positive or negative.

Only *days*, *seconds* and *microseconds* are stored internally. Arguments are converted to those units:

- A millisecond is converted to 1000 microseconds.
- A minute is converted to 60 seconds.
- An hour is converted to 3600 seconds.
- A week is converted to 7 days.

and days, seconds and microseconds are then normalized so that the representation is unique, with

- `0 <= microseconds < 1000000`
- `0 <= seconds < 3600*24` (the number of seconds in one day)
- `-999999999 <= days <= 999999999`

If any argument is a float and there are fractional microseconds, the fractional microseconds left over from all arguments are combined and their sum is rounded to the nearest microsecond. If no argument is a float, the conversion and normalization processes are exact (no information is lost).

If the normalized value of days lies outside the indicated range, `OverflowError` is raised.

Note that normalization of negative values may be surprising at first. For example,

```
>>> from datetime import timedelta
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)
```

Class attributes are:

`timedelta.min`¶
The most negative `timedelta` object, `timedelta(-999999999)`.

`timedelta.max`¶
The most positive `timedelta` object, `timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)`.

`timedelta.resolution`¶
The smallest possible difference between non-equal `timedelta` objects, `timedelta(microseconds=1)`.

Note that, because of normalization, `timedelta.max` > `-timedelta.min`. `-timedelta.max` is not representable as a `timedelta` object.

Instance attributes (read-only):

Attribute	Value
`days`	Between -999999999 and 999999999 inclusive
`seconds`	Between 0 and 86399 inclusive
`microseconds`	Between 0 and 999999 inclusive

Supported operations:

Operation	Result
`t1 = t2 + t3`	Sum of *t2* and *t3*. Afterwards *t1*-*t2* == *t3* and *t1*-*t3* == *t2* are true. (1)
`t1 = t2 - t3`	Difference of *t2* and *t3*. Afterwards *t1* == *t2* - *t3* and *t2* == *t1* + *t3* are true. (1)
`t1 = t2 * i or t1 = i * t2`	Delta multiplied by an integer or long. Afterwards *t1* // i == *t2* is true, provided `i != 0`.
	In general, *t1* * i == *t1* * (i-1) + *t1* is true. (1)
`t1 = t2 // i`	The floor is computed and the remainder (if any) is thrown away. (3)
`+t1`	Returns a `timedelta` object with the same value. (2)

`-t1`	equivalent to `timedelta`(*-t1.days*, *-t1.seconds*, *-t1.microseconds*), and to *t1** -1. (1)(4)
`abs(t)`	equivalent to +*t* when `t.days >= 0`, and to -*t* when `t.days < 0`. (2)

Notes:

1. This is exact, but may overflow.
2. This is exact, and cannot overflow.
3. Division by 0 raises `ZeroDivisionError`.
4. *-timedelta.max* is not representable as a `timedelta` object.

In addition to the operations listed above `timedelta` objects support certain additions and subtractions with `date` and `datetime` objects (see below).

Comparisons of `timedelta` objects are supported with the `timedelta` object representing the smaller duration considered to be the smaller timedelta. In order to stop mixed-type comparisons from falling back to the default comparison by object address, when a `timedelta` object is compared to an object of a different type, `TypeError` is raised unless the comparison is `==` or `!=`. The latter cases return `False` or `True`, respectively.

`timedelta` objects are *hashable* (usable as dictionary keys), support efficient pickling, and in Boolean contexts, a `timedelta` object is considered to be true if and only if it isn't equal to `timedelta(0)`.

Example usage:

```
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> another_year = timedelta(weeks=40, days=84, hours=23,
...                          minutes=50, seconds=600)  # adds up to 365 days
>>> year == another_year
True
>>> ten_years = 10 * year
>>> ten_years, ten_years.days // 365
(datetime.timedelta(3650), 10)
>>> nine_years = ten_years - year
>>> nine_years, nine_years.days // 365
(datetime.timedelta(3285), 9)
>>> three_years = nine_years // 3;
>>> three_years, three_years.days // 365
(datetime.timedelta(1095), 3)
>>> abs(three_years - ten_years) == 2 * three_years + year
True
```

9.1.3. `date` Objects¶

A `date` object represents a date (year, month and day) in an idealized calendar, the current Gregorian calendar indefinitely extended in both directions. January 1 of year 1 is called day number 1, January 2 of year 1 is called day number 2, and so on. This matches the definition of the "proleptic Gregorian" calendar in Dershowitz and Reingold's book Calendrical Calculations, where it's the base calendar for all computations. See the book for algorithms for converting between proleptic Gregorian ordinals and many other calendar systems.

class `datetime.date`(*year*, *month*, *day*)¶

All arguments are required. Arguments may be ints or longs, in the following ranges:

- `MINYEAR <= year <= MAXYEAR`
- `1 <= month <= 12`
- `1 <= day <= number of days in the given month and year`

If an argument outside those ranges is given, `ValueError` is raised.

Other constructors, all class methods:

`date.today`()¶
Return the current local date. This is equivalent to `date.fromtimestamp(time.time())`.

`date.fromtimestamp`(*timestamp*)¶
Return the local date corresponding to the POSIX timestamp, such as is returned by `time.time()`. This may raise `ValueError`, if the timestamp is out of the range of values supported by the platform C `localtime()` function. It's common for this to be restricted to years from 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by `fromtimestamp()`.

`date.fromordinal`(*ordinal*)¶
Return the date corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1. `ValueError` is raised unless `1 <= ordinal <= date.max.toordinal()`. For any date *d*, `date.fromordinal(d.toordinal()) == d`.

Class attributes:

`date.min`¶

The earliest representable date, `date(MINYEAR, 1, 1)`.

`date.max`¶

The latest representable date, `date(MAXYEAR, 12, 31)`.

`date.resolution`¶

The smallest possible difference between non-equal date objects, `timedelta(days=1)`.

Instance attributes (read-only):

`date.year`¶

Between `MINYEAR` and `MAXYEAR` inclusive.

`date.month`¶

Between 1 and 12 inclusive.

`date.day`¶

Between 1 and the number of days in the given month of the given year.

Supported operations:

Operation	Result
`date2 = date1 + timedelta`	*date2* is `timedelta.days` days removed from *date1*. (1)
`date2 = date1 - timedelta`	Computes *date2* such that `date2 + timedelta == date1`. (2)
`timedelta = date1 - date2`	(3)
`date1 < date2`	*date1* is considered less than *date2* when *date1* precedes *date2* in time. (4)

Notes:

1. *date2* is moved forward in time if `timedelta.days > 0`, or backward if `timedelta.days < 0`. Afterward `date2 - date1 == timedelta.days`. `timedelta.seconds` and `timedelta.microseconds` are ignored. `OverflowError` is raised if `date2.year` would be smaller than `MINYEAR` or larger than `MAXYEAR`.
2. This isn't quite equivalent to date1 + (-timedelta), because -timedelta in isolation can overflow in cases where date1 - timedelta does not. `timedelta.seconds` and `timedelta.microseconds` are ignored.
3. This is exact, and cannot overflow. timedelta.seconds and timedelta.microseconds are 0, and date2 + timedelta == date1 after.
4. In other words, `date1 < date2` if and only if `date1.toordinal() < date2.toordinal()`. In order to stop comparison from falling back to the default scheme of comparing object addresses, date comparison normally raises `TypeError` if the other comparand isn't also a `date` object. However, `NotImplemented` is returned instead if the other comparand has a `timetuple()` attribute. This hook gives other kinds of date objects a chance at implementing mixed-type comparison. If not, when a `date` object is compared to an object of a different type, `TypeError` is raised unless the comparison is == or `!=`. The latter cases return `False` or `True`, respectively.

Dates can be used as dictionary keys. In Boolean contexts, all `date` objects are considered to be true.

Instance methods:

`date.replace`(*year*, *month*, *day*)¶

Return a date with the same value, except for those members given new values by whichever keyword arguments are specified. For example, if `d == date(2002, 12, 31)`, then `d.replace(day=26) == date(2002, 12, 26)`.

`date.timetuple`()¶

Return a `time.struct_time` such as returned by `time.localtime()`. The hours, minutes and seconds are 0, and the DST flag is -1. `d.timetuple()` is equivalent to `time.struct_time((d.year, d.month, d.day, 0, 0, 0, d.weekday(), d.toordinal() - date(d.year, 1, 1).toordinal() + 1, -1))`

`date.toordinal`()¶

Return the proleptic Gregorian ordinal of the date, where January 1 of year 1 has ordinal 1. For any `date` object *d*, `date.fromordinal(d.toordinal()) == d`.

`date.weekday`()¶

Return the day of the week as an integer, where Monday is 0 and Sunday is 6. For example, `date(2002, 12, 4).weekday() == 2`, a Wednesday. See also `isoweekday()`.

`date.isoweekday`()¶

Return the day of the week as an integer, where Monday is 1 and Sunday is 7. For example, `date(2002, 12, 4).isoweekday() == 3`, a Wednesday. See also `weekday()`, `isocalendar()`.

`date.isocalendar`()¶

Return a 3-tuple, (ISO year, ISO week number, ISO weekday).

The ISO calendar is a widely used variant of the Gregorian calendar. See http://www.phys.uu.nl/~vgent/calendar/isocalendar.htm for a good explanation.

The ISO year consists of 52 or 53 full weeks, and where a week starts on a Monday and ends on a Sunday. The first week of an ISO year is the first (Gregorian) calendar week of a year containing a Thursday. This is called week number 1, and the ISO year of that Thursday is the same as its Gregorian year.

For example, 2004 begins on a Thursday, so the first week of ISO year 2004 begins on Monday, 29 Dec 2003 and ends on Sunday, 4 Jan 2004, so that `date(2003, 12, 29).isocalendar() == (2004, 1, 1)` and `date(2004, 1, 4).isocalendar() == (2004, 1, 7)`.

`date.isoformat()`¶

Return a string representing the date in ISO 8601 format, 'YYYY-MM-DD'. For example, `date(2002, 12, 4).isoformat() == '2002-12-04'`.

`date.__str__()`¶

For a date *d*, `str(d)` is equivalent to `d.isoformat()`.

`date.ctime()`¶

Return a string representing the date, for example `date(2002, 12, 4).ctime() == 'Wed Dec 4 00:00:00 2002'`. `d.ctime()` is equivalent to `time.ctime(time.mktime(d.timetuple()))` on platforms where the native C `ctime()` function (which `time.ctime()` invokes, but which `date.ctime()` does not invoke) conforms to the C standard.

`date.strftime(`*format*`)`¶

Return a string representing the date, controlled by an explicit format string. Format codes referring to hours, minutes or seconds will see 0 values. See section *strftime() Behavior*.

Example of counting days to an event:

```
>>> import time
>>> from datetime import date
>>> today = date.today()
>>> today
datetime.date(2007, 12, 5)
>>> today == date.fromtimestamp(time.time())
True
>>> my_birthday = date(today.year, 6, 24)
>>> if my_birthday < today:
...     my_birthday = my_birthday.replace(year=today.year + 1)
>>> my_birthday
datetime.date(2008, 6, 24)
>>> time_to_birthday = abs(my_birthday - today)
>>> time_to_birthday.days
202
```

Example of working with `date`:

```
>>> from datetime import date
>>> d = date.fromordinal(730920) # 730920th day after 1. 1. 0001
>>> d
datetime.date(2002, 3, 11)
>>> t = d.timetuple()
>>> for i in t:
...     print i
2002                # year
3                   # month
11                  # day
0
0
0
0                   # weekday (0 = Monday)
70                  # 70th day in the year
-1
>>> ic = d.isocalendar()
>>> for i in ic:
...     print i
2002                # ISO year
11                  # ISO week number
1                   # ISO day number ( 1 = Monday )
>>> d.isoformat()
'2002-03-11'
>>> d.strftime("%d/%m/%y")
'11/03/02'
>>> d.strftime("%A %d. %B %Y")
'Monday 11. March 2002'
```

9.1.4. `datetime` Objects¶

A `datetime` object is a single object containing all the information from a `date` object and a `time` object. Like a `date` object, `datetime` assumes the current Gregorian calendar extended in both directions; like a time object, `datetime` assumes there are exactly 3600*24 seconds in every day.

Constructor:

class `datetime.datetime`(*year*, *month*, *day*[, *hour*[, *minute*[, *second*[, *microsecond*[, *tzinfo*]]]]])¶

The year, month and day arguments are required. *tzinfo* may be `None`, or an instance of a `tzinfo` subclass. The remaining arguments may be ints or longs, in the following ranges:

- `MINYEAR <= year <= MAXYEAR`
- `1 <= month <= 12`
- `1 <= day <= number of days in the given month and year`
- `0 <= hour < 24`
- `0 <= minute < 60`
- `0 <= second < 60`
- `0 <= microsecond < 1000000`

If an argument outside those ranges is given, `ValueError` is raised.

Other constructors, all class methods:

`datetime.today()`¶
Return the current local datetime, with `tzinfo` `None`. This is equivalent to `datetime.fromtimestamp(time.time())`. See also `now()`, `fromtimestamp()`.

`datetime.now`([*tz*])¶

Return the current local date and time. If optional argument *tz* is `None` or not specified, this is like `today()`, but, if possible, supplies more precision than can be gotten from going through a `time.time()` timestamp (for example, this may be possible on platforms supplying the C `gettimeofday()` function).

Else *tz* must be an instance of a class `tzinfo` subclass, and the current date and time are converted to *tz*'s time zone. In this case the result is equivalent to `tz.fromutc(datetime.utcnow().replace(tzinfo=tz))`. See also `today()`, `utcnow()`.

`datetime.utcnow()`¶
Return the current UTC date and time, with `tzinfo` `None`. This is like `now()`, but returns the current UTC date and time, as a naive `datetime` object. See also `now()`.

`datetime.fromtimestamp`(*timestamp*[, *tz*])¶

Return the local date and time corresponding to the POSIX timestamp, such as is returned by `time.time()`. If optional argument *tz* is `None` or not specified, the timestamp is converted to the platform's local date and time, and the returned `datetime` object is naive.

Else *tz* must be an instance of a class `tzinfo` subclass, and the timestamp is converted to *tz*'s time zone. In this case the result is equivalent to `tz.fromutc(datetime.utcfromtimestamp(timestamp).replace(tzinfo=tz))`.

`fromtimestamp()` may raise `ValueError`, if the timestamp is out of the range of values supported by the platform C `localtime()` or `gmtime()` functions. It's common for this to be restricted to years in 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by `fromtimestamp()`, and then it's possible to have two timestamps differing by a second that yield identical `datetime` objects. See also `utcfromtimestamp()`.

`datetime.utcfromtimestamp`(*timestamp*)¶
Return the UTC `datetime` corresponding to the POSIX timestamp, with `tzinfo` `None`. This may raise `ValueError`, if the timestamp is out of the range of values supported by the platform C `gmtime()` function. It's common for this to be restricted to years in 1970 through 2038. See also `fromtimestamp()`.

`datetime.fromordinal`(*ordinal*)¶
Return the `datetime` corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1. `ValueError` is raised unless `1 <= ordinal <= datetime.max.toordinal()`. The hour, minute, second and microsecond of the result are all 0, and `tzinfo` is `None`.

`datetime.combine`(*date*, *time*)¶
Return a new `datetime` object whose date members are equal to the given `date` object's, and whose time and `tzinfo` members are equal to the given `time` object's. For any `datetime` object *d*, `d == datetime.combine(d.date(), d.timetz())`. If date is a `datetime` object, its time and `tzinfo` members are ignored.

`datetime.strptime`(*date_string*, *format*)¶

Return a `datetime` corresponding to *date_string*, parsed according to *format*. This is equivalent to `datetime(*(time.strptime(date_string, format)[0:6]))`. `ValueError` is raised if the date_string and format can't be parsed by `time.strptime()` or if it returns a value which isn't a time tuple.

New in version 2.5.

Class attributes:

`datetime.min`¶

The earliest representable datetime, `datetime(MINYEAR, 1, 1, tzinfo=None)`.

`datetime.max`¶

The latest representable datetime, `datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None)`.

`datetime.resolution`¶

The smallest possible difference between non-equal datetime objects, `timedelta(microseconds=1)`.

Instance attributes (read-only):

`datetime.year`¶

Between MINYEAR and MAXYEAR inclusive.

`datetime.month`¶

Between 1 and 12 inclusive.

`datetime.day`¶

Between 1 and the number of days in the given month of the given year.

`datetime.hour`¶

In `range(24)`.

`datetime.minute`¶

In `range(60)`.

`datetime.second`¶

In `range(60)`.

`datetime.microsecond`¶

In `range(1000000)`.

`datetime.tzinfo`¶

The object passed as the *tzinfo* argument to the datetime constructor, or `None` if none was passed.

Supported operations:

Operation	Result
`datetime2 = datetime1 + timedelta`	(1)
`datetime2 = datetime1 - timedelta`	(2)
`timedelta = datetime1 - datetime2`	(3)
`datetime1 < datetime2`	Compares datetime to datetime. (4)

datetime2 is a duration of timedelta removed from datetime1, moving forward in time if `timedelta.days` > 0, or backward if `timedelta.days` < 0. The result has the same tzinfo member as the input datetime, and datetime2 - datetime1 == timedelta after. OverflowError is raised if datetime2.year would be smaller than MINYEAR or larger than MAXYEAR. Note that no time zone adjustments are done even if the input is an aware object.

Computes the datetime2 such that datetime2 + timedelta == datetime1. As for addition, the result has the same tzinfo member as the input datetime, and no time zone adjustments are done even if the input is aware. This isn't quite equivalent to datetime1 + (-timedelta), because -timedelta in isolation can overflow in cases where datetime1 - timedelta does not.

Subtraction of a datetime from a datetime is defined only if both operands are naive, or if both are aware. If one is aware and the other is naive, TypeError is raised.

If both are naive, or both are aware and have the same tzinfo member, the tzinfo members are ignored, and the result is a timedelta object *t* such that `datetime2 + t == datetime1`. No time zone adjustments are done in this case.

If both are aware and have different tzinfo members, `a-b` acts as if *a* and *b* were first converted to naive UTC datetimes first. The result is `(a.replace(tzinfo=None) - a.utcoffset()) - (b.replace(tzinfo=None) - b.utcoffset())` except that the implementation never overflows.

datetime1 is considered less than *datetime2* when *datetime1* precedes *datetime2* in time.

If one comparand is naive and the other is aware, TypeError is raised. If both comparands are aware, and have the same tzinfo member, the common tzinfo member is ignored and the base datetimes are compared. If both comparands are aware and have different tzinfo members, the comparands are first adjusted by subtracting their UTC offsets (obtained from `self.utcoffset()`).

Note

In order to stop comparison from falling back to the default scheme of comparing object addresses, datetime comparison normally raises TypeError if the other comparand isn't also a datetime object. However, `NotImplemented` is returned instead if the other comparand has a `timetuple()` attribute. This hook gives other kinds of date objects a chance at implementing mixed-type comparison. If not, when a datetime object is compared to an object of a different type, TypeError is raised unless the comparison is == or `!=`. The latter cases return False or True, respectively.

datetime objects can be used as dictionary keys. In Boolean contexts, all datetime objects are considered to be true.

Instance methods:

`datetime.date()`¶
Return `date` object with same year, month and day.

`datetime.time()`¶
Return `time` object with same hour, minute, second and microsecond. `tzinfo` is `None`. See also method `timetz()`.

`datetime.timetz()`¶
Return `time` object with same hour, minute, second, microsecond, and tzinfo members. See also method `time()`.

`datetime.replace(`[*year*[, *month*[, *day*[, *hour*[, *minute*[, *second*[, *microsecond*[, *tzinfo*]]]]]]]`)`¶
Return a datetime with the same members, except for those members given new values by whichever keyword arguments are specified. Note that `tzinfo=None` can be specified to create a naive datetime from an aware datetime with no conversion of date and time members.

`datetime.astimezone(`*tz*`)`¶

Return a `datetime` object with new `tzinfo` member *tz*, adjusting the date and time members so the result is the same UTC time as *self*, but in *tz*'s local time.

tz must be an instance of a `tzinfo` subclass, and its `utcoffset()` and `dst()` methods must not return `None`. *self* must be aware (`self.tzinfo` must not be `None`, and `self.utcoffset()` must not return `None`).

If `self.tzinfo` is *tz*, `self.astimezone(tz)` is equal to *self*: no adjustment of date or time members is performed. Else the result is local time in time zone *tz*, representing the same UTC time as *self*: after `astz = dt.astimezone(tz)`, `astz - astz.utcoffset()` will usually have the same date and time members as `dt - dt.utcoffset()`. The discussion of class `tzinfo` explains the cases at Daylight Saving Time transition boundaries where this cannot be achieved (an issue only if *tz* models both standard and daylight time).

If you merely want to attach a time zone object *tz* to a datetime *dt* without adjustment of date and time members, use `dt.replace(tzinfo=tz)`. If you merely want to remove the time zone object from an aware datetime *dt* without conversion of date and time members, use `dt.replace(tzinfo=None)`.

Note that the default `tzinfo.fromutc()` method can be overridden in a `tzinfo` subclass to affect the result returned by `astimezone()`. Ignoring error cases, `astimezone()` acts like:

```
def astimezone(self, tz):
   if self.tzinfo is tz:
       return self
   # Convert self to UTC, and attach the new time zone object.
   utc = (self - self.utcoffset()).replace(tzinfo=tz)
   # Convert from UTC to tz's local time.
   return tz.fromutc(utc)
```

`datetime.utcoffset()`¶
If `tzinfo` is `None`, returns `None`, else returns `self.tzinfo.utcoffset(self)`, and raises an exception if the latter doesn't return `None`, or a `timedelta` object representing a whole number of minutes with magnitude less than one day.

`datetime.dst()`¶
If `tzinfo` is `None`, returns `None`, else returns `self.tzinfo.dst(self)`, and raises an exception if the latter doesn't return `None`, or a `timedelta` object representing a whole number of minutes with magnitude less than one day.

`datetime.tzname()`¶
If `tzinfo` is `None`, returns `None`, else returns `self.tzinfo.tzname(self)`, raises an exception if the latter doesn't return `None` or a string object,

`datetime.timetuple()`¶
Return a `time.struct_time` such as returned by `time.localtime()`. `d.timetuple()` is equivalent to `time.struct_time((d.year, d.month, d.day, d.hour, d.minute, d.second, d.weekday(), d.toordinal() - date(d.year, 1, 1).toordinal() + 1, dst))` The `tm_isdst` flag of the result is set according to the `dst()` method: `tzinfo` is `None` or `dst()` returns `None`, `tm_isdst` is set to `-1`; else if `dst()` returns a non-zero value, `tm_isdst` is set to `1`; else `tm_isdst` is set to `0`.

`datetime.utctimetuple()`¶

If `datetime` instance *d* is naive, this is the same as `d.timetuple()` except that `tm_isdst` is forced to 0 regardless of what `d.dst()` returns. DST is never in effect for a UTC time.

If *d* is aware, *d* is normalized to UTC time, by subtracting `d.utcoffset()`, and a `time.struct_time` for the normalized time is returned. `tm_isdst` is forced to 0. Note that the result's `tm_year` member may be `MINYEAR`-1 or `MAXYEAR`+1, if *d*.year was `MINYEAR` or `MAXYEAR` and UTC adjustment spills over a year boundary.

`datetime.toordinal()`¶
Return the proleptic Gregorian ordinal of the date. The same as `self.date().toordinal()`.

`datetime.weekday()`¶
Return the day of the week as an integer, where Monday is 0 and Sunday is 6. The same as `self.date().weekday()`. See also `isoweekday()`.

`datetime.isoweekday()`¶
Return the day of the week as an integer, where Monday is 1 and Sunday is 7. The same as `self.date().isoweekday()`. See also `weekday()`, `isocalendar()`.

`datetime.isocalendar()`¶
Return a 3-tuple, (ISO year, ISO week number, ISO weekday). The same as `self.date().isocalendar()`.

`datetime.isoformat`([*sep*])¶

Return a string representing the date and time in ISO 8601 format, YYYY-MM-DDTHH:MM:SS.mmmmmm or, if `microsecond` is 0, YYYY-MM-DDTHH:MM:SS

If `utcoffset()` does not return `None`, a 6-character string is appended, giving the UTC offset in (signed) hours and minutes: YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM or, if `microsecond` is 0 YYYY-MM-DDTHH:MM:SS+HH:MM

The optional argument *sep* (default `'T'`) is a one-character separator, placed between the date and time portions of the result. For example,

```
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     def utcoffset(self, dt): return timedelta(minutes=-399)
...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
```

`datetime.__str__`()¶
For a `datetime` instance *d*, `str(d)` is equivalent to `d.isoformat(' ')`.

`datetime.ctime`()¶
Return a string representing the date and time, for example `datetime(2002, 12, 4, 20, 30, 40).ctime() == 'Wed Dec  4 20:30:40 2002'`. `d.ctime()` is equivalent to `time.ctime(time.mktime(d.timetuple()))` on platforms where the native C `ctime()` function (which `time.ctime()` invokes, but which `datetime.ctime()` does not invoke) conforms to the C standard.

`datetime.strftime`(*format*)¶
Return a string representing the date and time, controlled by an explicit format string. See section *strftime() Behavior*.

Examples of working with datetime objects:

```
>>> from datetime import datetime, date, time
>>> # Using datetime.combine()
>>> d = date(2005, 7, 14)
>>> t = time(12, 30)
>>> datetime.combine(d, t)
datetime.datetime(2005, 7, 14, 12, 30)
>>> # Using datetime.now() or datetime.utcnow()
>>> datetime.now()
datetime.datetime(2007, 12, 6, 16, 29, 43, 79043)   # GMT +1
>>> datetime.utcnow()
datetime.datetime(2007, 12, 6, 15, 29, 43, 79060)
>>> # Using datetime.strptime()
>>> dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
>>> dt
datetime.datetime(2006, 11, 21, 16, 30)
>>> # Using datetime.timetuple() to get tuple of all attributes
>>> tt = dt.timetuple()
>>> for it in tt:
...     print it
...
2006    # year
11      # month
21      # day
16      # hour
30      # minute
0       # second
1       # weekday (0 = Monday)
325     # number of days since 1st January
-1      # dst - method tzinfo.dst() returned None
>>> # Date in ISO format
>>> ic = dt.isocalendar()
>>> for it in ic:
...     print it
...
2006    # ISO year
47      # ISO week
2       # ISO weekday
>>> # Formatting datetime
>>> dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'
```

Using datetime with tzinfo:

```
>>> from datetime import timedelta, datetime, tzinfo
>>> class GMT1(tzinfo):
...     def __init__(self):         # DST starts last Sunday in March
...         d = datetime(dt.year, 4, 1)   # ends last Sunday in October
...         self.dston = d - timedelta(days=d.weekday() + 1)
...         d = datetime(dt.year, 11, 1)
...         self.dstoff = d - timedelta(days=d.weekday() + 1)
...     def utcoffset(self, dt):
...         return timedelta(hours=1) + self.dst(dt)
...     def dst(self, dt):
...         if self.dston <=  dt.replace(tzinfo=None) < self.dstoff:
...             return timedelta(hours=1)
...         else:
...             return timedelta(0)
...     def tzname(self,dt):
...          return "GMT +1"
...
>>> class GMT2(tzinfo):
...     def __init__(self):
...         d = datetime(dt.year, 4, 1)
...         self.dston = d - timedelta(days=d.weekday() + 1)
...         d = datetime(dt.year, 11, 1)
...         self.dstoff = d - timedelta(days=d.weekday() + 1)
...     def utcoffset(self, dt):
...         return timedelta(hours=1) + self.dst(dt)
...     def dst(self, dt):
...         if self.dston <=  dt.replace(tzinfo=None) < self.dstoff:
...             return timedelta(hours=2)
...         else:
...             return timedelta(0)
...     def tzname(self,dt):
...         return "GMT +2"
...
>>> gmt1 = GMT1()
>>> # Daylight Saving Time
>>> dt1 = datetime(2006, 11, 21, 16, 30, tzinfo=gmt1)
>>> dt1.dst()
datetime.timedelta(0)
>>> dt1.utcoffset()
datetime.timedelta(0, 3600)
>>> dt2 = datetime(2006, 6, 14, 13, 0, tzinfo=gmt1)
>>> dt2.dst()
datetime.timedelta(0, 3600)
>>> dt2.utcoffset()
datetime.timedelta(0, 7200)
>>> # Convert datetime to another time zone
>>> dt3 = dt2.astimezone(GMT2())
>>> dt3     # doctest: +ELLIPSIS
datetime.datetime(2006, 6, 14, 14, 0, tzinfo=<GMT2 object at 0x...>)
>>> dt2     # doctest: +ELLIPSIS
datetime.datetime(2006, 6, 14, 13, 0, tzinfo=<GMT1 object at 0x...>)
>>> dt2.utctimetuple() == dt3.utctimetuple()
True
```

9.1.5. `time` Objects¶

A time object represents a (local) time of day, independent of any particular day, and subject to adjustment via a `tzinfo` object.

class `datetime.time`(*hour*[, *minute*[, *second*[, *microsecond*[, *tzinfo*]]]])¶

All arguments are optional. *tzinfo* may be `None`, or an instance of a `tzinfo` subclass. The remaining arguments may be ints or longs, in the following ranges:

- `0 <= hour < 24`
- `0 <= minute < 60`
- `0 <= second < 60`
- `0 <= microsecond < 1000000`.

If an argument outside those ranges is given, `ValueError` is raised. All default to `0` except *tzinfo*, which defaults to `None`.

Class attributes:

`time.min`¶

The earliest representable time, `time(0, 0, 0, 0)`.

`time.max`¶

The latest representable time, `time(23, 59, 59, 999999)`.

`time.resolution`¶

The smallest possible difference between non-equal time objects, `timedelta(microseconds=1)`, although note that arithmetic on time objects is not supported.

Instance attributes (read-only):

`time.hour`¶

In `range(24)`.

`time.minute`¶

In `range(60)`.

`time.second`¶

In `range(60)`.

`time.microsecond`¶

In `range(1000000)`.

`time.tzinfo`¶

The object passed as the tzinfo argument to the time constructor, or `None` if none was passed.

Supported operations:

- comparison of time to time, where *a* is considered less than *b* when *a* precedes *b* in time. If one comparand is naive and the other is aware, `TypeError` is raised. If both comparands are aware, and have the same tzinfo member, the common tzinfo member is ignored and the base times are compared. If both comparands are aware and have different tzinfo members, the comparands are first adjusted by subtracting their UTC offsets (obtained from `self.utcoffset()`). In order to stop mixed-type comparisons from falling back to the default comparison by object address, when a time object is compared to an object of a different type, `TypeError` is raised unless the comparison is `==` or `!=`. The latter cases return `False` or `True`, respectively.
- hash, use as dict key
- efficient pickling
- in Boolean contexts, a time object is considered to be true if and only if, after converting it to minutes and subtracting `utcoffset()` (or `0` if that's `None`), the result is non-zero.

Instance methods:

`time.replace`([*hour*[, *minute*[, *second*[, *microsecond*[, *tzinfo*]]]]])¶

Return a time with the same value, except for those members given new values by whichever keyword arguments are specified. Note that `tzinfo=None` can be specified to create a naive time from an aware time, without conversion of the time members.

`time.isoformat`()¶

Return a string representing the time in ISO 8601 format, HH:MM:SS.mmmmmm or, if self.microsecond is 0, HH:MM:SS If `utcoffset()` does not return `None`, a 6-character string is appended, giving the UTC offset in (signed) hours and minutes: HH:MM:SS.mmmmmm+HH:MM or, if self.microsecond is 0, HH:MM:SS+HH:MM

`time.__str__`()¶

For a time *t*, `str(t)` is equivalent to `t.isoformat()`.

`time.strftime`(*format*)¶

Return a string representing the time, controlled by an explicit format string. See section *strftime() Behavior*.

`time.utcoffset`()¶

If tzinfo is `None`, returns `None`, else returns `self.tzinfo.utcoffset(None)`, and raises an exception if the latter doesn't return `None` or a timedelta object representing a whole number of minutes with magnitude less than one day.

`time.dst`()¶

If tzinfo is `None`, returns `None`, else returns `self.tzinfo.dst(None)`, and raises an exception if the latter doesn't return `None`, or a timedelta object representing a whole number of minutes with magnitude less than one day.

`time.tzname`()¶

If tzinfo is `None`, returns `None`, else returns `self.tzinfo.tzname(None)`, or raises an exception if the latter doesn't return `None` or a string object.

Example:

```
>>> from datetime import time, tzinfo
>>> class GMT1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1)
...     def dst(self, dt):
...         return timedelta(0)
...     def tzname(self,dt):
...         return "Europe/Prague"
...
>>> t = time(12, 10, 30, tzinfo=GMT1())
>>> t                                # doctest: +ELLIPSIS
datetime.time(12, 10, 30, tzinfo=<GMT1 object at 0x...>)
>>> gmt = GMT1()
>>> t.isoformat()
'12:10:30+01:00'
>>> t.dst()
datetime.timedelta(0)
>>> t.tzname()
'Europe/Prague'
>>> t.strftime("%H:%M:%S %Z")
'12:10:30 Europe/Prague'
```

9.1.6. `tzinfo` Objects¶

`tzinfo` is an abstract base class, meaning that this class should not be instantiated directly. You need to derive a concrete subclass, and (at least) supply implementations of the standard `tzinfo` methods needed by the `datetime` methods you use. The `datetime` module does not supply any concrete subclasses of `tzinfo`.

An instance of (a concrete subclass of) `tzinfo` can be passed to the constructors for `datetime` and `time` objects. The latter objects view their members as being in local time, and the `tzinfo` object supports methods revealing offset of local time from UTC, the name of the time zone, and DST offset, all relative to a date or time object passed to them.

Special requirement for pickling: A `tzinfo` subclass must have an `__init__()` method that can be called with no arguments, else it can be pickled but possibly not unpickled again. This is a technical requirement that may be relaxed in the future.

A concrete subclass of `tzinfo` may need to implement the following methods. Exactly which methods are needed depends on the uses made of aware `datetime` objects. If in doubt, simply implement all of them.

`tzinfo.utcoffset`(*self*, *dt*)¶

Return offset of local time from UTC, in minutes east of UTC. If local time is west of UTC, this should be negative. Note that this is intended to be the total offset from UTC; for example, if a `tzinfo` object represents both time zone and DST adjustments, `utcoffset()` should return their sum. If the UTC offset isn't known, return `None`. Else the value returned must be a `timedelta` object specifying a whole number of minutes in the range -1439 to 1439 inclusive (1440 = 24*60; the magnitude of the offset must be less than one day). Most implementations of `utcoffset()` will probably look like one of these two:

```
return CONSTANT                 # fixed-offset class
return CONSTANT + self.dst(dt)  # daylight-aware class
```

If `utcoffset()` does not return `None`, `dst()` should not return `None` either.

The default implementation of `utcoffset()` raises `NotImplementedError`.

`tzinfo.dst`(*self*, *dt*)¶

Return the daylight saving time (DST) adjustment, in minutes east of UTC, or `None` if DST information isn't known. Return `timedelta(0)` if DST is not in effect. If DST is in effect, return the offset as a `timedelta` object (see `utcoffset()` for details). Note that DST offset, if applicable, has already been added to the UTC offset returned by `utcoffset()`, so there's no need to consult `dst()` unless you're interested in obtaining DST info separately. For example, `datetime.timetuple()` calls its `tzinfo` member's `dst()` method to determine how the `tm_isdst` flag should be set, and `tzinfo.fromutc()` calls `dst()` to account for DST changes when crossing time zones.

An instance *tz* of a `tzinfo` subclass that models both standard and daylight times must be consistent in this sense:

```
tz.utcoffset(dt) - tz.dst(dt)
```

must return the same result for every `datetime` *dt* with `dt.tzinfo == tz` For sane `tzinfo` subclasses, this expression yields the time zone's "standard offset", which should not depend on the date or the time, but only on geographic location. The implementation of `datetime.astimezone()` relies on this, but cannot detect violations; it's the programmer's responsibility to ensure it. If a `tzinfo` subclass cannot guarantee this, it may be able to override the default implementation of `tzinfo.fromutc()` to work correctly with `astimezone()` regardless.

Most implementations of `dst()` will probably look like one of these two:

```
def dst(self):
   # a fixed-offset class:  doesn't account for DST
   return timedelta(0)
```

or

```
def dst(self):
   # Code to set dston and dstoff to the time zone's DST
   # transition times based on the input dt.year, and expressed
   # in standard local time.  Then

   if dston <= dt.replace(tzinfo=None) < dstoff:
       return timedelta(hours=1)
   else:
       return timedelta(0)
```

The default implementation of `dst()` raises `NotImplementedError`.

`tzinfo.tzname`(*self*, *dt*)¶

Return the time zone name corresponding to the `datetime` object *dt*, as a string. Nothing about string names is defined by the `datetime` module, and there's no requirement that it mean anything in particular. For example, "GMT", "UTC", "-500", "-5:00", "EDT", "US/Eastern", "America/New York" are all valid replies. Return `None` if a string name isn't known. Note that this is a method rather than a fixed string primarily because some `tzinfo` subclasses will wish to return different names depending on the specific value of *dt* passed, especially if the `tzinfo` class is accounting for daylight time.

The default implementation of `tzname()` raises `NotImplementedError`.

These methods are called by a `datetime` or `time` object, in response to their methods of the same names. A `datetime` object passes itself as the argument, and a `time` object passes `None` as the argument. A `tzinfo` subclass's methods should therefore be prepared to accept a *dt* argument of `None`, or of class `datetime`.

When `None` is passed, it's up to the class designer to decide the best response. For example, returning `None` is appropriate if the class wishes to say that time objects don't participate in the `tzinfo` protocols. It may be more useful for `utcoffset(None)` to return the standard UTC offset, as there is no other convention for discovering the standard offset.

When a `datetime` object is passed in response to a `datetime` method, `dt.tzinfo` is the same object as *self*. `tzinfo` methods can rely on this, unless user code calls `tzinfo` methods directly. The intent is that the `tzinfo` methods interpret *dt* as being in local time, and not need worry about objects in other timezones.

There is one more `tzinfo` method that a subclass may wish to override:

`tzinfo.fromutc`(*self*, *dt*)¶

This is called from the default `datetime.astimezone()` implementation. When called from that, `dt.tzinfo` is *self*, and *dt*'s date and time members are to be viewed as expressing a UTC time. The purpose of `fromutc()` is to adjust the date and time members, returning an equivalent datetime in *self*'s local time.

Most `tzinfo` subclasses should be able to inherit the default `fromutc()` implementation without problems. It's strong enough to handle fixed-offset time zones, and time zones accounting for both standard and daylight time, and the latter even if the DST transition times differ in different years. An example of a time zone the default `fromutc()` implementation may not handle correctly in all cases is one where the standard offset (from UTC) depends on the specific date and time passed, which can happen for political reasons. The default implementations of `astimezone()` and `fromutc()` may not produce the result you want if the result is one of the hours straddling the moment the standard offset changes.

Skipping code for error cases, the default `fromutc()` implementation acts like:

```
def fromutc(self, dt):
   # raise ValueError error if dt.tzinfo is not self
   dtoff = dt.utcoffset()
   dtdst = dt.dst()
   # raise ValueError if dtoff is None or dtdst is None
   delta = dtoff - dtdst  # this is self's standard offset
   if delta:
       dt += delta   # convert to standard local time
       dtdst = dt.dst()
       # raise ValueError if dtdst is None
   if dtdst:
       return dt + dtdst
   else:
       return dt
```

Example `tzinfo` classes:

```
from datetime import tzinfo, timedelta, datetime

ZERO = timedelta(0)
HOUR = timedelta(hours=1)

# A UTC class.
```

```
class UTC(tzinfo):
   """UTC"""

   def utcoffset(self, dt):
       return ZERO

   def tzname(self, dt):
       return "UTC"

   def dst(self, dt):
       return ZERO

utc = UTC()

# A class building tzinfo objects for fixed-offset time zones.
# Note that FixedOffset(0, "UTC") is a different way to build a
# UTC tzinfo object.

class FixedOffset(tzinfo):
   """Fixed offset in minutes east from UTC."""

   def __init__(self, offset, name):
       self.__offset = timedelta(minutes = offset)
       self.__name = name

   def utcoffset(self, dt):
       return self.__offset

   def tzname(self, dt):
       return self.__name

   def dst(self, dt):
       return ZERO

# A class capturing the platform's idea of local time.

import time as _time

STDOFFSET = timedelta(seconds = -_time.timezone)
if _time.daylight:
   DSTOFFSET = timedelta(seconds = -_time.altzone)
else:
   DSTOFFSET = STDOFFSET

DSTDIFF = DSTOFFSET - STDOFFSET

class LocalTimezone(tzinfo):

   def utcoffset(self, dt):
       if self._isdst(dt):
           return DSTOFFSET
       else:
           return STDOFFSET

   def dst(self, dt):
       if self._isdst(dt):
           return DSTDIFF
       else:
           return ZERO

   def tzname(self, dt):
       return _time.tzname[self._isdst(dt)]

   def _isdst(self, dt):
       tt = (dt.year, dt.month, dt.day,
             dt.hour, dt.minute, dt.second,
             dt.weekday(), 0, -1)
       stamp = _time.mktime(tt)
```

```
        tt = _time.localtime(stamp)
        return tt.tm_isdst > 0

Local = LocalTimezone()


# A complete implementation of current DST rules for major US time zones.

def first_sunday_on_or_after(dt):
    days_to_go = 6 - dt.weekday()
    if days_to_go:
        dt += timedelta(days_to_go)
    return dt


# US DST Rules
#
# This is a simplified (i.e., wrong for a few cases) set of rules for US
# DST start and end times. For a complete and up-to-date set of DST rules
# and timezone definitions, visit the Olson Database (or try pytz):
# http://www.twinsun.com/tz/tz-link.htm
# http://sourceforge.net/projects/pytz/ (might not be up-to-date)
#
# In the US, since 2007, DST starts at 2am (standard time) on the second
# Sunday in March, which is the first Sunday on or after Mar 8.
DSTSTART_2007 = datetime(1, 3, 8, 2)
# and ends at 2am (DST time; 1am standard time) on the first Sunday of Nov.
DSTEND_2007 = datetime(1, 11, 1, 1)
# From 1987 to 2006, DST used to start at 2am (standard time) on the first
# Sunday in April and to end at 2am (DST time; 1am standard time) on the last
# Sunday of October, which is the first Sunday on or after Oct 25.
DSTSTART_1987_2006 = datetime(1, 4, 1, 2)
DSTEND_1987_2006 = datetime(1, 10, 25, 1)
# From 1967 to 1986, DST used to start at 2am (standard time) on the last
# Sunday in April (the one on or after April 24) and to end at 2am (DST time;
# 1am standard time) on the last Sunday of October, which is the first Sunday
# on or after Oct 25.
DSTSTART_1967_1986 = datetime(1, 4, 24, 2)
DSTEND_1967_1986 = DSTEND_1987_2006

class USTimeZone(tzinfo):

    def __init__(self, hours, reprname, stdname, dstname):
        self.stdoffset = timedelta(hours=hours)
        self.reprname = reprname
        self.stdname = stdname
        self.dstname = dstname

    def __repr__(self):
        return self.reprname

    def tzname(self, dt):
        if self.dst(dt):
            return self.dstname
        else:
            return self.stdname

    def utcoffset(self, dt):
        return self.stdoffset + self.dst(dt)

    def dst(self, dt):
        if dt is None or dt.tzinfo is None:
            # An exception may be sensible here, in one or both cases.
            # It depends on how you want to treat them.  The default
            # fromutc() implementation (called by the default astimezone()
            # implementation) passes a datetime with dt.tzinfo is self.
            return ZERO
        assert dt.tzinfo is self
        # Find start and end times for US DST. For years before 1967, return
        # ZERO for no DST.
        if 2006 < dt.year:
            dststart, dstend = DSTSTART_2007, DSTEND_2007
        elif 1986 < dt.year < 2007:
            dststart, dstend = DSTSTART_1987_2006, DSTEND_1987_2006
        elif 1966 < dt.year < 1987:
            dststart, dstend = DSTSTART_1967_1986, DSTEND_1967_1986
        else:
            return ZERO

        start = first_sunday_on_or_after(dststart.replace(year=dt.year))
        end = first_sunday_on_or_after(dstend.replace(year=dt.year))

        # Can't compare naive to aware objects, so strip the timezone from
        # dt first.
        if start <= dt.replace(tzinfo=None) < end:
            return HOUR
        else:
            return ZERO

Eastern  = USTimeZone(-5, "Eastern",  "EST", "EDT")
Central  = USTimeZone(-6, "Central",  "CST", "CDT")
Mountain = USTimeZone(-7, "Mountain", "MST", "MDT")
Pacific  = USTimeZone(-8, "Pacific",  "PST", "PDT")
```

Note that there are unavoidable subtleties twice per year in a `tzinfo` subclass accounting for both standard and daylight time, at the DST transition points. For concreteness, consider US Eastern (UTC -0500), where EDT begins the minute after 1:59 (EST) on the first Sunday in April, and ends the minute after 1:59 (EDT) on the last Sunday in October:

```
  UTC   3:MM  4:MM  5:MM  6:MM  7:MM  8:MM
 EST  22:MM 23:MM  0:MM  1:MM  2:MM  3:MM
 EDT  23:MM  0:MM  1:MM  2:MM  3:MM  4:MM

start  22:MM 23:MM  0:MM  1:MM  3:MM  4:MM

 end  23:MM  0:MM  1:MM  1:MM  2:MM  3:MM
```

When DST starts (the “start” line), the local wall clock leaps from 1:59 to 3:00. A wall time of the form 2:MM doesn’t really make sense on that day, so `astimezone(Eastern)` won’t deliver a result with `hour == 2` on the day DST begins. In order for `astimezone()` to make this guarantee, the `rzinfo.dst()` method must consider times in the “missing hour” (2:MM for Eastern) to be in daylight time.

When DST ends (the “end” line), there’s a potentially worse problem: there’s an hour that can’t be spelled unambiguously in local wall time: the last hour of daylight time. In Eastern, that’s times of the form 5:MM UTC on the day daylight time ends. The local wall clock leaps from 1:59 (daylight time) back to 1:00 (standard time) again. Local times of the form 1:MM are ambiguous. `astimezone()` mimics the local clock’s behavior by mapping two adjacent UTC hours into the same local hour then. In the Eastern example, UTC times of the form 5:MM and 6:MM both map to 1:MM when converted to Eastern. In order for `astimezone()` to make this guarantee, the `tzinfo.dst()` method must consider times in the “repeated hour” to be in standard time. This is easily arranged, as in the example, by expressing DST switch times in the time zone’s standard local time.

Applications that can’t bear such ambiguities should avoid using hybrid `tzinfo` subclasses; there are no ambiguities when using UTC, or any other fixed-offset `tzinfo` subclass (such as a class representing only EST (fixed offset -5 hours), or only EDT (fixed offset -4 hours)).

9.1.7. `strftime()` Behavior¶

`date`, `datetime`, and `time` objects all support a `strftime(format)` method, to create a string representing the time under the control of an explicit format string. Broadly speaking, `d.strftime(fmt)` acts like the `time` module’s `time.strftime(fmt, d.timetuple())` although not all objects support a `timetuple()` method.

For `time` objects, the format codes for year, month, and day should not be used, as time objects have no such values. If they’re used anyway, `1900` is substituted for the year, and `0` for the month and day.

For `date` objects, the format codes for hours, minutes, seconds, and microseconds should not be used, as `date` objects have no such values. If they’re used anyway, `0` is substituted for them.

New in version 2.6: `time` and `datetime` objects support a `%f` format code which expands to the number of microseconds in the object, zero-padded on the left to six places.

For a naive object, the `%z` and `%Z` format codes are replaced by empty strings.

For an aware object:

`%z`

`utcoffset()` is transformed into a 5-character string of the form +HHMM or -HHMM, where HH is a 2-digit string giving the number of UTC offset hours, and MM is a 2-digit string giving the number of UTC offset minutes. For example, if `utcoffset()` returns `timedelta(hours=-3, minutes=-30)`, `%z` is replaced with the string `'-0330'`.

`%Z`

If `tzname()` returns `None`, `%Z` is replaced by an empty string. Otherwise `%Z` is replaced by the returned value, which must be a string.

The full set of format codes supported varies across platforms, because Python calls the platform C library's `strftime()` function, and platform variations are common.

The following is a list of all the format codes that the C standard (1989 version) requires, and these work on all platforms with a standard C implementation. Note that the 1999 version of the C standard added additional format codes.

The exact range of years for which `strftime()` works also varies across platforms. Regardless of platform, years before 1900 cannot be used.

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].	
`%f`	Microsecond as a decimal number [0,999999], zero-padded on the left	(1)
`%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.	(2)
`%S`	Second as a decimal number [00,61].	(3)
`%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.	(4)
`%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.	(4)
`%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`	UTC offset in the form +HHMM or -HHMM (empty string if the the object is naive).	(5)
`%Z`	Time zone name (empty string if the object is naive).	
`%%`	A literal `'%'` character.	

Notes:

1. When used with the `strptime()` function, the `%f` directive accepts from one to six digits and zero pads on the right. `%f` is an extension to the set of format characters in the C standard (but implemented separately in datetime objects, and therefore always available).
2. 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.
3. The range really is `0` to `61`; according to the Posix standard this accounts for leap seconds and the (very rare) double leap seconds. The `time` module may produce and does accept leap seconds since it is based on the Posix standard, but the `datetime` module does not accept leap seconds in `strptime()` input nor will it produce them in `strftime()` output.
4. 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.
5. For example, if `utcoffset()` returns `timedelta(hours=-3, minutes=-30)`, `%z` is replaced with the string `'-0330'`.

Table Of Contents

Previous topic

9. Data Types

Next topic

9.2. `calendar` — General calendar-related functions

This Page

- Show Source

Navigation

- index
- modules |
- next |
- previous |
- Python v2.6.4 documentation »
- The Python Standard Library »
- 9. Data Types »