`datetime` — Basic date and time types

Source code: Lib/datetime.py

The `datetime` module supplies classes for manipulating dates and times.

While date and time arithmetic is supported, the focus of the implementation is on efficient attribute extraction for output formatting and manipulation.

Tip: Skip to the format codes.

See also:

Module `calendar`
: General calendar related functions.

Module `time`
: Time access and conversions.

Module `zoneinfo`
: Concrete time zones representing the IANA time zone database.

Package dateutil
: Third-party library with expanded time zone and parsing support.

Package DateType
: Third-party library that introduces distinct static types to e.g. allow static type checkers to differentiate between naive and aware datetimes.

Aware and Naive Objects

Date and time objects may be categorized as "aware" or "naive" depending on whether or not they include timezone information.

With sufficient knowledge of applicable algorithmic and political time adjustments, such as time zone and daylight saving time information, an **aware** object can locate itself relative to other aware objects. An aware object represents a specific moment in time that is not open to interpretation. [1]

A **naive** object does not contain enough information to unambiguously locate itself relative to other date/time objects. Whether a naive object represents Coordinated Universal Time (UTC), local time, or time in some other timezone is purely up to the program, just like it is up to the program whether a particular number represents metres, miles, or mass. Naive objects are easy to understand and to work with, at the cost of ignoring some aspects of reality.

For applications requiring aware objects, `datetime` and `time` objects have an optional time zone information attribute, `tzinfo`, that can be set to 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.

EDT timezones. Supporting timezones at deeper levels of detail is up to the application. The rules for time adjustment across the world are more political than rational, change frequently, and there is no standard suitable for every application aside from UTC.

Constants

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`.

`datetime.`**`UTC`**

Alias for the UTC timezone singleton `datetime.timezone.utc`.

New in version 3.11.

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).

class `datetime.`**`timezone`**

A class that implements the `tzinfo` abstract base class as a fixed offset from the UTC.

New in version 3.2.

Objects of these types are immutable.

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

Common Properties

The `date`, `datetime`, `time`, and `timezone` types share these common features:

- Objects of these types are immutable.
- Objects of these types are hashable, meaning that they can be used as dictionary keys.
- Objects of these types support efficient pickling via the `pickle` module.

Determining if an Object is Aware or Naive

Objects of the `date` type are always naive.

An object of type `time` or `datetime` may be aware or naive.

A `datetime` object *d* is aware if both of the following hold:

1. `d.tzinfo` is not `None`
2. `d.tzinfo.utcoffset(d)` does not return `None`

Otherwise, *d* is naive.

A `time` object *t* is aware if both of the following hold:

1. `t.tzinfo` is not `None`
2. `t.tzinfo.utcoffset(None)` does not return `None`.

Otherwise, *t* is naive.

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

`timedelta` Objects

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

class datetime.**timedelta**(*days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0*)

All arguments are optional and default to `0`. Arguments may be integers 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.

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`

The following example illustrates how any arguments besides *days*, *seconds* and *microseconds* are “merged” and normalized into those three resulting attributes:

```
>>> from datetime import timedelta
>>> delta = timedelta(
...     days=50,
...     seconds=27,
...     microseconds=10,
...     milliseconds=29000,
...     minutes=5,
...     hours=8,
...     weeks=2
... )
>>> # Only days, seconds, and microseconds remain
>>> delta
datetime.timedelta(days=64, seconds=29156, microseconds=10)
```

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 using round-half-to-even tiebreaker. 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:

`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.

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)(6)
`t1 = t2 * i or t1 = i * t2`	Delta multiplied by an integer. Afterwards *t1* // i == *t2* is true, provided `i != 0`.
	In general, *t1* * i == *t1* * (i-1) + *t1* is true. (1)
`t1 = t2 * f or t1 = f * t2`	Delta multiplied by a float. The result is rounded to the nearest multiple of timedelta.resolution using round-half-to-even.
`f = t2 / t3`	Division (3) of overall duration *t2* by interval unit *t3*. Returns a `float` object.
`t1 = t2 / f or t1 = t2 / i`	Delta divided by a float or an int. The result is rounded to the nearest multiple of timedelta.resolution using round-half-to-even.
`t1 = t2 // i` or `t1 = t2 // t3`	The floor is computed and the remainder (if any) is thrown away. In the second case, an integer is returned. (3)
`t1 = t2 % t3`	The remainder is computed as a `timedelta` object. (3)
`q, r = divmod(t1, t2)`	Computes the quotient and the remainder: `q = t1 // t2` (3) and `r = t1 % t2`. q is an integer and r is a `timedelta` object.
`+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)
`str(t)`	Returns a string in the form `[D day[s], ][H]H:MM:SS[.UUUUUU]`, where D is negative for negative `t`. (5)
`repr(t)`	Returns a string representation of the `timedelta` object as a constructor call with canonical attribute values.

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.

5. String representations of `timedelta` objects are normalized similarly to their internal representation. This leads to somewhat unusual results for negative timedeltas. For example:

```
>>> timedelta(hours=-5)
datetime.timedelta(days=-1, seconds=68400)
>>> print(_)
-1 day, 19:00:00
```

6. The expression `t2 - t3` will always be equal to the expression `t2 + (-t3)` except when t3 is equal to `timedelta.max`; in that case the former will produce a result while the latter will overflow.

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

Changed in version 3.2: Floor division and true division of a `timedelta` object by another `timedelta` object are now supported, as are remainder operations and the `divmod()` function. True division and multiplication of a `timedelta` object by a `float` object are now supported.

Comparisons of `timedelta` objects are supported, with some caveats.

The comparisons == or `!=` *always* return a `bool`, no matter the type of the compared object:

```
>>> from datetime import timedelta
>>> delta1 = timedelta(seconds=57)
>>> delta2 = timedelta(hours=25, seconds=2)
>>> delta2 != delta1
True
>>> delta2 == 5
False
```

For all other comparisons (such as < and >), when a `timedelta` object is compared to an object of a different type, `TypeError` is raised:

```
>>> delta2 > delta1
True
>>> delta2 > 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances of 'datetime.timedelta' and 'int'
```

In Boolean contexts, a `timedelta` object is considered to be true if and only if it isn't equal to `timedelta(0)`.

Instance methods:

For interval units other than seconds, use the division form directly (e.g. `td / timedelta(microseconds=1)`).

Note that for very large time intervals (greater than 270 years on most platforms) this method will lose microsecond accuracy.

New in version 3.2.

Examples of usage: `timedelta`

An additional example of normalization:

```
>>> # Components of another_year add up to exactly 365 days
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> another_year = timedelta(weeks=40, days=84, hours=23,
...                          minutes=50, seconds=600)
>>> year == another_year
True
>>> year.total_seconds()
31536000.0
```

Examples of `timedelta` arithmetic:

```
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> ten_years = 10 * year
>>> ten_years
datetime.timedelta(days=3650)
>>> ten_years.days // 365
10
>>> nine_years = ten_years - year
>>> nine_years
datetime.timedelta(days=3285)
>>> three_years = nine_years // 3
>>> three_years, three_years.days // 365
(datetime.timedelta(days=1095), 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. [2]

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

All arguments are required. Arguments must be integers, in the following ranges:

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

Other constructors, all class methods:

classmethod date.**today**()

Return the current local date.

This is equivalent to `date.fromtimestamp(time.time())`.

classmethod date.**fromtimestamp**(*timestamp*)

Return the local date corresponding to the POSIX timestamp, such as is returned by `time.time()`.

This may raise `OverflowError`, if the timestamp is out of the range of values supported by the platform C `localtime()` function, and `OSError` on `localtime()` failure. 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()`.

Changed in version 3.3: Raise `OverflowError` instead of `ValueError` if the timestamp is out of the range of values supported by the platform C `localtime()` function. Raise `OSError` instead of `ValueError` on `localtime()` failure.

classmethod 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`.

classmethod date.**fromisoformat**(*date_string*)

Return a `date` corresponding to a *date_string* given in any valid ISO 8601 format, except ordinal dates (e.g. `YYYY-DDD`):

```
>>> from datetime import date
>>> date.fromisoformat('2019-12-04')
datetime.date(2019, 12, 4)
>>> date.fromisoformat('20191204')
datetime.date(2019, 12, 4)
>>> date.fromisoformat('2021-W01-1')
datetime.date(2021, 1, 4)
```

New in version 3.7.

Changed in version 3.11: Previously, this method only supported the format `YYYY-MM-DD`.

classmethod date.**fromisocalendar**(*year*, *week*, *day*)

Return a `date` corresponding to the ISO calendar date specified by year, week and day. This is the inverse of the function `date.isocalendar()`.

New in version 3.8.

Class attributes:

date.**min**

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

`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* will be `timedelta.days` days after *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. `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()`. Date comparison 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.

In Boolean contexts, all `date` objects are considered to be true.

Instance methods:

`date.`**`replace`**`(`*`year=self.year, month=self.month, day=self.day`*`)`

Example:

```
>>> from datetime import date
>>> d = date(2002, 12, 31)
>>> d.replace(day=26)
datetime.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(), yday, -1))
```

where `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` is the day number within the current year starting with `1` for January 1st.

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 named tuple object with three components: `year`, `week` and `weekday`.

The ISO calendar is a widely used variant of the Gregorian calendar. [3]

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:

```
>>> from datetime import date
>>> date(2003, 12, 29).isocalendar()
datetime.IsoCalendarDate(year=2004, week=1, weekday=1)
>>> date(2004, 1, 4).isocalendar()
datetime.IsoCalendarDate(year=2004, week=1, weekday=7)
```

date.**isoformat**()

Return a string representing the date in ISO 8601 format, `YYYY-MM-DD`:

```
>>> from datetime import date
>>> 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:

```
>>> from datetime import date
>>> 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 also strftime() and strptime() Behavior and `date.isoformat()`.

date.**__format__**(*format*)

Same as `date.strftime()`. This makes it possible to specify a format string for a `date` object in formatted string literals and when using `str.format()`. See also strftime() and strptime() Behavior and `date.isoformat()`.

Examples of Usage: `date`

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)
```

More examples of working with `date`:

```
>>> from datetime import date
>>> d = date.fromordinal(730920) # 730920th day after 1. 1. 0001
>>> d
datetime.date(2002, 3, 11)

>>> # Methods related to formatting string output
>>> d.isoformat()
'2002-03-11'
>>> d.strftime("%d/%m/%y")
'11/03/02'
>>> d.strftime("%A %d. %B %Y")
'Monday 11. March 2002'
>>> d.ctime()
'Mon Mar 11 00:00:00 2002'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}.'.format(d, "day", "month")
'The day is 11, the month is March.'

>>> # Methods for to extracting 'components' under different calendars
>>> 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 )

>>> # A date object is immutable; all operations produce a new object
>>> d.replace(year=2005)
datetime.date(2005, 3, 11)
```

`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=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, ***, *fold=0*)

- `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`,
- `fold in [0, 1]`.

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

New in version 3.6: Added the `fold` argument.

Other constructors, all class methods:

classmethod `datetime.`**`today`**`()`

Return the current local datetime, with `tzinfo` `None`.

Equivalent to:

```
datetime.fromtimestamp(time.time())
```

See also `now()`, `fromtimestamp()`.

This method is functionally equivalent to `now()`, but without a `tz` parameter.

classmethod `datetime.`**`now`**`(`*`tz=None`*`)`

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).

If *tz* is not `None`, it must be an instance of a `tzinfo` subclass, and the current date and time are converted to *tz*'s time zone.

This function is preferred over `today()` and `utcnow()`.

classmethod `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. An aware current UTC datetime can be obtained by calling `datetime.now(timezone.utc)`. See also `now()`.

> **Warning:** Because naive `datetime` objects are treated by many `datetime` methods as local times, it is preferred to use aware datetimes to represent times in UTC. As such, the recommended way to create an object representing the current time in UTC is by calling `datetime.now(timezone.utc)`.

classmethod datetime.**fromtimestamp**(*timestamp*, *tz=None*)

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.

If *tz* is not `None`, it must be an instance of a `tzinfo` subclass, and the timestamp is converted to *tz*'s time zone.

`fromtimestamp()` may raise `OverflowError`, if the timestamp is out of the range of values supported by the platform C `localtime()` or `gmtime()` functions, and `OSError` on `localtime()` or `gmtime()` failure. It's common for this to be restricted to years 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. This method is preferred over `utcfromtimestamp()`.

Changed in version 3.3: Raise `OverflowError` instead of `ValueError` if the timestamp is out of the range of values supported by the platform C `localtime()` or `gmtime()` functions. Raise `OSError` instead of `ValueError` on `localtime()` or `gmtime()` failure.

Changed in version 3.6: `fromtimestamp()` may return instances with `fold` set to 1.

classmethod datetime.**utcfromtimestamp**(*timestamp*)

Return the UTC `datetime` corresponding to the POSIX timestamp, with `tzinfo` `None`. (The resulting object is naive.)

This may raise `OverflowError`, if the timestamp is out of the range of values supported by the platform C `gmtime()` function, and `OSError` on `gmtime()` failure. It's common for this to be restricted to years in 1970 through 2038.

To get an aware `datetime` object, call `fromtimestamp()`:

```
datetime.fromtimestamp(timestamp, timezone.utc)
```

On the POSIX compliant platforms, it is equivalent to the following expression:

```
datetime(1970, 1, 1, tzinfo=timezone.utc) + timedelta(seconds=timestamp)
```

except the latter formula always supports the full years range: between `MINYEAR` and `MAXYEAR` inclusive.

Warning: Because naive `datetime` objects are treated by many `datetime` methods as local times, it is preferred to use aware datetimes to represent times in UTC. As such, the recommended way to create an object representing a specific timestamp in UTC is by calling `datetime.fromtimestamp(timestamp, tz=timezone.utc)`.

Changed in version 3.3: Raise `OverflowError` instead of `ValueError` if the timestamp is out of the range of values supported by the platform C `gmtime()` function. Raise `OSError` instead of `ValueError` on `gmtime()` failure.

classmethod 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`.

classmethod datetime.**combine**(*date, time, tzinfo=time.tzinfo*)

Return a new `datetime` object whose date components are equal to the given `date` object's, and whose time components are equal to the given `time` object's. If the *tzinfo* argument is provided, its value is used to set the `tzinfo` attribute of the result, otherwise the `tzinfo` attribute of the *time* argument is used. If the *date* argument is a `datetime` object, its time components and `tzinfo` attributes are ignored.

For any `datetime` object *d*, `d == datetime.combine(d.date(), d.time(), d.tzinfo)`.

Changed in version 3.6: Added the *tzinfo* argument.

classmethod datetime.**fromisoformat**(*date_string*)

Return a `datetime` corresponding to a *date_string* in any valid ISO 8601 format, with the following exceptions:

1. Time zone offsets may have fractional seconds.
2. The `T` separator may be replaced by any single unicode character.
3. Ordinal dates are not currently supported.
4. Fractional hours and minutes are not supported.

Examples:

```
>>> from datetime import datetime
>>> datetime.fromisoformat('2011-11-04')
datetime.datetime(2011, 11, 4, 0, 0)
>>> datetime.fromisoformat('20111104')
datetime.datetime(2011, 11, 4, 0, 0)
>>> datetime.fromisoformat('2011-11-04T00:05:23')
datetime.datetime(2011, 11, 4, 0, 5, 23)
>>> datetime.fromisoformat('2011-11-04T00:05:23Z')
datetime.datetime(2011, 11, 4, 0, 5, 23, tzinfo=datetime.timezone.utc)
>>> datetime.fromisoformat('20111104T000523')
datetime.datetime(2011, 11, 4, 0, 5, 23)
>>> datetime.fromisoformat('2011-W01-2T00:05:23.283')
datetime.datetime(2011, 1, 4, 0, 5, 23, 283000)
>>> datetime.fromisoformat('2011-11-04 00:05:23.283')
datetime.datetime(2011, 11, 4, 0, 5, 23, 283000)
>>> datetime.fromisoformat('2011-11-04 00:05:23.283+00:00')
datetime.datetime(2011, 11, 4, 0, 5, 23, 283000, tzinfo=datetime.timezone.utc)
>>> datetime.fromisoformat('2011-11-04T00:05:23+04:00')
datetime.datetime(2011, 11, 4, 0, 5, 23,
    tzinfo=datetime.timezone(datetime.timedelta(seconds=14400)))
```

New in version 3.7.

Changed in version 3.11: Previously, this method only supported formats that could be emitted by `date.isoformat()` or `datetime.isoformat()`.

components of the datetime are populated with their normal default values. This is the inverse of the function `datetime.isocalendar()`.

New in version 3.8.

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

Return a `datetime` corresponding to *date_string*, parsed according to *format*.

If *format* does not contain microseconds or timezone information, 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. See also strftime() and strptime() Behavior and `datetime.fromisoformat()`.

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`**

datetime.**tzinfo**

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

datetime.**fold**

In `[0, 1]`. Used to disambiguate wall times during a repeated interval. (A repeated interval occurs when clocks are rolled back at the end of daylight saving time or when the UTC offset for the current zone is decreased for political reasons.) The value 0 (1) represents the earlier (later) of the two moments with the same wall time representation.

New in version 3.6.

Supported operations:

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

1. 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` attribute 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.

2. Computes the datetime2 such that datetime2 + timedelta == datetime1. As for addition, the result has the same `tzinfo` attribute as the input datetime, and no time zone adjustments are done even if the input is aware.

3. 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` attribute, the `tzinfo` attributes 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` attributes, `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.

4. *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 an order comparison is attempted. For equality comparisons, naive instances are never equal to aware instances.

attributes, the comparands are first adjusted by subtracting their UTC offsets (obtained from `self.utcoffset()`).

Changed in version 3.3: Equality comparisons between aware and naive `datetime` instances don't raise `TypeError`.

> **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.

Instance methods:

`datetime.`**`date`**`()`

Return `date` object with same year, month and day.

`datetime.`**`time`**`()`

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

Changed in version 3.6: The fold value is copied to the returned `time` object.

`datetime.`**`timetz`**`()`

Return `time` object with same hour, minute, second, microsecond, fold, and tzinfo attributes. See also method `time()`.

Changed in version 3.6: The fold value is copied to the returned `time` object.

`datetime.`**`replace`**`(year=self.year, month=self.month, day=self.day, hour=self.hour, minute=self.minute, second=self.second, microsecond=self.microsecond, tzinfo=self.tzinfo, *, fold=0)`

Return a datetime with the same attributes, except for those attributes 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 data.

New in version 3.6: Added the `fold` argument.

`datetime.`**`astimezone`**`(tz=None)`

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

If provided, *tz* must be an instance of a `tzinfo` subclass, and its `utcoffset()` and `dst()` methods must not return `None`. If *self* is naive, it is presumed to represent time in the system timezone.

zone name and offset obtained from the OS.

If `self.tzinfo` is *tz*, `self.astimezone(tz)` is equal to *self*: no adjustment of date or time data is performed. Else the result is local time in the timezone *tz*, representing the same UTC time as *self*: after `astz = dt.astimezone(tz)`, `astz - astz.utcoffset()` will have the same date and time data as `dt - dt.utcoffset()`.

If you merely want to attach a time zone object *tz* to a datetime *dt* without adjustment of date and time data, 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 data, 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)
```

Changed in version 3.3: *tz* now can be omitted.

Changed in version 3.6: The `astimezone()` method can now be called on naive instances that are presumed to represent system local time.

`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 with magnitude less than one day.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

`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 with magnitude less than one day.

Changed in version 3.7: The DST offset is not restricted to a whole number of minutes.

`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,
```

where `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` is the day number within the current year starting with `1` for January 1st. 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 an `OverflowError` may be raised if *d*.year was `MINYEAR` or `MAXYEAR` and UTC adjustment spills over a year boundary.

> **Warning:** Because naive `datetime` objects are treated by many `datetime` methods as local times, it is preferred to use aware datetimes to represent times in UTC; as a result, using `datetime.utctimetuple()` may give misleading results. If you have a naive `datetime` representing UTC, use `datetime.replace(tzinfo=timezone.utc)` to make it aware, at which point you can use `datetime.timetuple()`.

`datetime.`**`toordinal()`**

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

`datetime.`**`timestamp()`**

Return POSIX timestamp corresponding to the `datetime` instance. The return value is a `float` similar to that returned by `time.time()`.

Naive `datetime` instances are assumed to represent local time and this method relies on the platform C `mktime()` function to perform the conversion. Since `datetime` supports wider range of values than `mktime()` on many platforms, this method may raise `OverflowError` or `OSError` for times far in the past or far in the future.

For aware `datetime` instances, the return value is computed as:

```
(dt - datetime(1970, 1, 1, tzinfo=timezone.utc)).total_seconds()
```

New in version 3.3.

Changed in version 3.6: The `timestamp()` method uses the `fold` attribute to disambiguate the times during a repeated interval.

> **Note:** There is no method to obtain the POSIX timestamp directly from a naive `datetime` instance representing UTC time. If your application uses this convention and your system timezone is not set to UTC, you can obtain the POSIX timestamp by supplying `tzinfo=timezone.utc`:
>
> ```
> timestamp = dt.replace(tzinfo=timezone.utc).timestamp()
> ```
>
> or by calculating the timestamp directly:

`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 named tuple with three components: `year`, `week` and `weekday`. The same as `self.date().isocalendar()`.

`datetime.`**`isoformat`**`(`*`sep='T', timespec='auto'`*`)`

Return a string representing the date and time in ISO 8601 format:

- `YYYY-MM-DDTHH:MM:SS.ffffff`, if `microsecond` is not 0
- `YYYY-MM-DDTHH:MM:SS`, if `microsecond` is 0

If `utcoffset()` does not return `None`, a string is appended, giving the UTC offset:

- `YYYY-MM-DDTHH:MM:SS.ffffff+HH:MM[:SS[.ffffff]]`, if `microsecond` is not 0
- `YYYY-MM-DDTHH:MM:SS+HH:MM[:SS[.ffffff]]`, if `microsecond` is 0

Examples:

```
>>> from datetime import datetime, timezone
>>> datetime(2019, 5, 18, 15, 17, 8, 132263).isoformat()
'2019-05-18T15:17:08.132263'
>>> datetime(2019, 5, 18, 15, 17, tzinfo=timezone.utc).isoformat()
'2019-05-18T15:17:00+00:00'
```

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):
...     """A time zone with an arbitrary, constant -06:39 offset."""
...     def utcoffset(self, dt):
...         return timedelta(hours=-6, minutes=-39)
...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
>>> datetime(2009, 11, 27, microsecond=100, tzinfo=TZ()).isoformat()
'2009-11-27T00:00:00.000100-06:39'
```

The optional argument *timespec* specifies the number of additional components of the time to include (the default is `'auto'`). It can be one of the following:

- `'auto'`: Same as `'seconds'` if `microsecond` is 0, same as `'microseconds'` otherwise.
- `'hours'`: Include the `hour` in the two-digit `HH` format.

- `'milliseconds'`: Include full time, but truncate fractional second part to milliseconds. `HH:MM:SS.sss` format.
- `'microseconds'`: Include full time in `HH:MM:SS.ffffff` format.

Note: Excluded time components are truncated, not rounded.

`ValueError` will be raised on an invalid *timespec* argument:

```
>>> from datetime import datetime
>>> datetime.now().isoformat(timespec='minutes')
'2002-12-25T00:00'
>>> dt = datetime(2015, 1, 1, 12, 30, 59, 0)
>>> dt.isoformat(timespec='microseconds')
'2015-01-01T12:30:59.000000'
```

New in version 3.6: Added the *timespec* argument.

`datetime.__str__()`

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

`datetime.ctime()`

Return a string representing the date and time:

```
>>> from datetime import datetime
>>> datetime(2002, 12, 4, 20, 30, 40).ctime()
'Wed Dec  4 20:30:40 2002'
```

The output string will *not* include time zone information, regardless of whether the input is aware or naive.

`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 also strftime() and strptime() Behavior and `datetime.isoformat()`.

`datetime.__format__(format)`

Same as `datetime.strftime()`. This makes it possible to specify a format string for a `datetime` object in formatted string literals and when using `str.format()`. See also strftime() and strptime() Behavior and `datetime.isoformat()`.

Examples of Usage: `datetime`

Examples of working with `datetime` objects:

```
>>> # 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()
>>> datetime.now()
datetime.datetime(2007, 12, 6, 16, 29, 43, 79043)   # GMT +1
>>> datetime.now(timezone.utc)
datetime.datetime(2007, 12, 6, 15, 29, 43, 79060, tzinfo=datetime.timezone.utc)

>>> # 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 a datetime
>>> dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}, the {3} is {0:%I:%M%p}.'.format(dt, "day", '
'The day is 21, the month is November, the time is 04:30PM.'
```

The example below defines a `tzinfo` subclass capturing time zone information for Kabul, Afghanistan, which used +4 UTC until 1945 and then +4:30 UTC thereafter:

```
from datetime import timedelta, datetime, tzinfo, timezone

class KabulTz(tzinfo):
    # Kabul used +4 until 1945, when they moved to +4:30
    UTC_MOVE_DATE = datetime(1944, 12, 31, 20, tzinfo=timezone.utc)
```

```
            return timedelta(hours=4)
        elif (1945, 1, 1, 0, 0) <= dt.timetuple()[:5] < (1945, 1, 1, 0, 30):
            # An ambiguous ("imaginary") half-hour range representing
            # a 'fold' in time due to the shift from +4 to +4:30.
            # If dt falls in the imaginary range, use fold to decide how
            # to resolve. See PEP495.
            return timedelta(hours=4, minutes=(30 if dt.fold else 0))
        else:
            return timedelta(hours=4, minutes=30)

    def fromutc(self, dt):
        # Follow same validations as in datetime.tzinfo
        if not isinstance(dt, datetime):
            raise TypeError("fromutc() requires a datetime argument")
        if dt.tzinfo is not self:
            raise ValueError("dt.tzinfo is not self")

        # A custom implementation is required for fromutc as
        # the input to this function is a datetime with utc values
        # but with a tzinfo set to self.
        # See datetime.astimezone or fromtimestamp.
        if dt.replace(tzinfo=timezone.utc) >= self.UTC_MOVE_DATE:
            return dt + timedelta(hours=4, minutes=30)
        else:
            return dt + timedelta(hours=4)

    def dst(self, dt):
        # Kabul does not observe daylight saving time.
        return timedelta(0)

    def tzname(self, dt):
        if dt >= self.UTC_MOVE_DATE:
            return "+04:30"
        return "+04"
```

Usage of `KabulTz` from above:

```
>>> tz1 = KabulTz()

>>> # Datetime before the change
>>> dt1 = datetime(1900, 11, 21, 16, 30, tzinfo=tz1)
>>> print(dt1.utcoffset())
4:00:00

>>> # Datetime after the change
>>> dt2 = datetime(2006, 6, 14, 13, 0, tzinfo=tz1)
>>> print(dt2.utcoffset())
4:30:00

>>> # Convert datetime to another time zone
>>> dt3 = dt2.astimezone(timezone.utc)
>>> dt3
datetime.datetime(2006, 6, 14, 8, 30, tzinfo=datetime.timezone.utc)
>>> dt2
datetime.datetime(2006, 6, 14, 13, 0, tzinfo=KabulTz())
>>> dt2 == dt3
True
```

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=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, ***, *fold=0*)

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

- `0 <= hour < 24`,
- `0 <= minute < 60`,
- `0 <= second < 60`,
- `0 <= microsecond < 1000000`,
- `fold in [0, 1]`.

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.

`time.`**`fold`**

decreased for political reasons.) The value 0 (1) represents the earlier (later) of the two moments with the same wall time representation.

New in version 3.6.

`time` objects support 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 an order comparison is attempted. For equality comparisons, naive instances are never equal to aware instances.

If both comparands are aware, and have the same `tzinfo` attribute, the common `tzinfo` attribute is ignored and the base times are compared. If both comparands are aware and have different `tzinfo` attributes, 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.

Changed in version 3.3: Equality comparisons between aware and naive `time` instances don't raise `TypeError`.

In Boolean contexts, a `time` object is always considered to be true.

Changed in version 3.5: Before Python 3.5, a `time` object was considered to be false if it represented midnight in UTC. This behavior was considered obscure and error-prone and has been removed in Python 3.5. See bpo-13936 for full details.

Other constructor:

classmethod `time.`**`fromisoformat`**`(`*`time_string`*`)`

Return a `time` corresponding to a *time_string* in any valid ISO 8601 format, with the following exceptions:

1. Time zone offsets may have fractional seconds.
2. The leading `T`, normally required in cases where there may be ambiguity between a date and a time, is not required.
3. Fractional seconds may have any number of digits (anything beyond 6 will be truncated).
4. Fractional hours and minutes are not supported.

Examples:

```
>>> from datetime import time
>>> time.fromisoformat('04:23:01')
datetime.time(4, 23, 1)
>>> time.fromisoformat('T04:23:01')
datetime.time(4, 23, 1)
>>> time.fromisoformat('T042301')
datetime.time(4, 23, 1)
>>> time.fromisoformat('04:23:01.000384')
datetime.time(4, 23, 1, 384)
>>> time.fromisoformat('04:23:01,000')
datetime.time(4, 23, 1, 384)
>>> time.fromisoformat('04:23:01+04:00')
datetime.time(4, 23, 1, tzinfo=datetime.timezone(datetime.timedelta(seconds=14400))
>>> time.fromisoformat('04:23:01Z')
datetime.time(4, 23, 1, tzinfo=datetime.timezone.utc)
```

New in version 3.7.

Changed in version 3.11: Previously, this method only supported formats that could be emitted by `time.isoformat()`.

Instance methods:

`time.`**`replace`**`(`*`hour=self.hour, minute=self.minute, second=self.second, microsecond=self.microsecond, tzinfo=self.tzinfo, *, fold=0`*`)`

Return a `time` with the same value, except for those attributes 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 data.

`time.`**`isoformat`**`(`*`timespec='auto'`*`)`

Return a string representing the time in ISO 8601 format, one of:

- `HH:MM:SS.ffffff`, if `microsecond` is not 0
- `HH:MM:SS`, if `microsecond` is 0
- `HH:MM:SS.ffffff+HH:MM[:SS[.ffffff]]`, if `utcoffset()` does not return `None`
- `HH:MM:SS+HH:MM[:SS[.ffffff]]`, if `microsecond` is 0 and `utcoffset()` does not return `None`

The optional argument *timespec* specifies the number of additional components of the time to include (the default is `'auto'`). It can be one of the following:

- `'auto'`: Same as `'seconds'` if `microsecond` is 0, same as `'microseconds'` otherwise.
- `'hours'`: Include the `hour` in the two-digit `HH` format.
- `'minutes'`: Include `hour` and `minute` in `HH:MM` format.
- `'seconds'`: Include `hour`, `minute`, and `second` in `HH:MM:SS` format.
- `'milliseconds'`: Include full time, but truncate fractional second part to milliseconds. `HH:MM:SS.sss` format.
- `'microseconds'`: Include full time in `HH:MM:SS.ffffff` format.

Note: Excluded time components are truncated, not rounded.

`ValueError` will be raised on an invalid *timespec* argument.

Example:

```
>>> from datetime import time
>>> time(hour=12, minute=34, second=56, microsecond=123456).isoformat(timespec='min
'12:34'
>>> dt = time(hour=12, minute=34, second=56, microsecond=0)
>>> dt.isoformat(timespec='microseconds')
'12:34:56.000000'
```

New in version 3.6: Added the *timespec* argument.

`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 also strftime() and strptime()

`time.__format__(format)`

Same as `time.strftime()`. This makes it possible to specify a format string for a `time` object in formatted string literals and when using `str.format()`. See also strftime() and strptime() Behavior and `time.isoformat()`.

`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 with magnitude less than one day.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

`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 with magnitude less than one day.

Changed in version 3.7: The DST offset is not restricted to a whole number of minutes.

`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.

Examples of Usage: `time`

Examples of working with a `time` object:

```
>>> from datetime import time, tzinfo, timedelta
>>> class TZ1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1)
...     def dst(self, dt):
...         return timedelta(0)
...     def tzname(self,dt):
...         return "+01:00"
...     def  __repr__(self):
...         return f"{self.__class__.__name__}()"
...
>>> t = time(12, 10, 30, tzinfo=TZ1())
>>> t
datetime.time(12, 10, 30, tzinfo=TZ1())
>>> t.isoformat()
```

```
datetime.timedelta(0)
>>> t.tzname()
'+01:00'
>>> t.strftime("%H:%M:%S %Z")
'12:10:30 +01:00'
>>> 'The {} is {:%H:%M}.'.format("time", t)
'The time is 12:10.'
```

`tzinfo` Objects

class `datetime.`**`tzinfo`**

This is an abstract base class, meaning that this class should not be instantiated directly. Define a subclass of `tzinfo` to capture information about a particular time zone.

An instance of (a concrete subclass of) `tzinfo` can be passed to the constructors for `datetime` and `time` objects. The latter objects view their attributes 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.

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 provides `timezone`, a simple concrete subclass of `tzinfo` which can represent timezones with fixed offset from UTC such as UTC itself or North American EST and EDT.

Special requirement for pickling: A `tzinfo` subclass must have an `__init__()` method that can be called with no arguments, otherwise 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`**`(`*`dt`*`)`

Return offset of local time from UTC, as a `timedelta` object that is positive east of UTC. If local time is west of UTC, this should be negative.

This represents 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 strictly between `-timedelta(hours=24)` and `timedelta(hours=24)` (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`.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

`tzinfo.`**`dst`**`(`*`dt`*`)`

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` attribute'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, dt):
    # a fixed-offset class:  doesn't account for DST
    return timedelta(0)
```

or:

```
def dst(self, dt):
    # 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.

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

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

Changed in version 3.7: The DST offset is not restricted to a whole number of minutes.

`tzinfo.`**`tzname`**`(`*`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.

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`**`(`*`dt`*`)`

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

In the following `tzinfo_examples.py` file there are some examples of `tzinfo` classes:

```
from datetime import tzinfo, timedelta, datetime

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

```
# A class capturing the platform's idea of local time.
# (May result in wrong values on historical times in
#  timezones where UTC offset and/or the DST rules had
#  changed in the past.)
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 fromutc(self, dt):
        assert dt.tzinfo is self
        stamp = (dt - datetime(1970, 1, 1, tzinfo=self)) // SECOND
        args = _time.localtime(stamp)[:6]
        dst_diff = DSTDIFF // SECOND
        # Detect fold
        fold = (args == _time.localtime(stamp - dst_diff))
        return datetime(*args, microsecond=dt.microsecond,
                        tzinfo=self, fold=fold)

    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, 0)
        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)
```

```
# 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
# https://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) on the first Sunday of Nov.
DSTEND_2007 = datetime(1, 11, 1, 2)
# 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) 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, 2)
# 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)
# 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

def us_dst_range(year):
    # Find start and end times for US DST. For years before 1967, return
    # start = end for no DST.
    if 2006 < year:
        dststart, dstend = DSTSTART_2007, DSTEND_2007
    elif 1986 < year < 2007:
        dststart, dstend = DSTSTART_1987_2006, DSTEND_1987_2006
    elif 1966 < year < 1987:
        dststart, dstend = DSTSTART_1967_1986, DSTEND_1967_1986
    else:
        return (datetime(year, 1, 1), ) * 2

    start = first_sunday_on_or_after(dststart.replace(year=year))
    end = first_sunday_on_or_after(dstend.replace(year=year))
    return start, end


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:
```

```
    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
        start, end = us_dst_range(dt.year)
        # Can't compare naive to aware objects, so strip the timezone from
        # dt first.
        dt = dt.replace(tzinfo=None)
        if start + HOUR <= dt < end - HOUR:
            # DST is in effect.
            return HOUR
        if end - HOUR <= dt < end:
            # Fold (an ambiguous hour): use dt.fold to disambiguate.
            return ZERO if dt.fold else HOUR
        if start <= dt < start + HOUR:
            # Gap (a non-existent hour): reverse the fold rule.
            return HOUR if dt.fold else ZERO
        # DST is off.
        return ZERO

    def fromutc(self, dt):
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        start = start.replace(tzinfo=self)
        end = end.replace(tzinfo=self)
        std_time = dt + self.stdoffset
        dst_time = std_time + HOUR
        if end <= dst_time < end + HOUR:
            # Repeated hour
            return std_time.replace(fold=1)
        if std_time < start or dst_time >= end:
            # Standard time
            return std_time
        if start <= std_time < end - HOUR:
            # Daylight saving time
            return dst_time


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 second Sunday in March, and ends the minute after 1:59 (EDT) on the first Sunday in November:

```
  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
```

```
  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. For example, at the Spring forward transition of 2016, we get:

```
>>> from datetime import datetime, timezone
>>> from tzinfo_examples import HOUR, Eastern
>>> u0 = datetime(2016, 3, 13, 5, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname())
...
05:00:00 UTC = 00:00:00 EST
06:00:00 UTC = 01:00:00 EST
07:00:00 UTC = 03:00:00 EDT
08:00:00 UTC = 04:00:00 EDT
```

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, but earlier times have the `fold` attribute set to 0 and the later times have it set to 1. For example, at the Fall back transition of 2016, we get:

```
>>> u0 = datetime(2016, 11, 6, 4, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname(), t.fold)
...
04:00:00 UTC = 00:00:00 EDT 0
05:00:00 UTC = 01:00:00 EDT 0
06:00:00 UTC = 01:00:00 EST 1
07:00:00 UTC = 02:00:00 EST 0
```

Note that the `datetime` instances that differ only by the value of the `fold` attribute are considered equal in comparisons.

Applications that can’t bear wall-time ambiguities should explicitly check the value of the `fold` attribute or avoid using hybrid `tzinfo` subclasses; there are no ambiguities when using `timezone`, 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)).

See also:

`zoneinfo`

zoneinfo brings the *IANA timezone database* (also known as the Olson database) to Python, and its usage is recommended.

IANA timezone database
The Time Zone Database (often called tz, tzdata or zoneinfo) contains code and data that represent the history of local time for many representative locations around the globe. It is updated periodically to reflect changes made by political bodies to time zone boundaries, UTC offsets, and daylight-saving rules.

`timezone` Objects

The `timezone` class is a subclass of `tzinfo`, each instance of which represents a timezone defined by a fixed offset from UTC.

Objects of this class cannot be used to represent timezone information in the locations where different offsets are used in different days of the year or where historical changes have been made to civil time.

class datetime.**timezone**(*offset*, *name=None*)

The *offset* argument must be specified as a `timedelta` object representing the difference between the local time and UTC. It must be strictly between `-timedelta(hours=24)` and `timedelta(hours=24)`, otherwise `ValueError` is raised.

The *name* argument is optional. If specified it must be a string that will be used as the value returned by the `datetime.tzname()` method.

New in version 3.2.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

timezone.**utcoffset**(*dt*)

Return the fixed value specified when the `timezone` instance is constructed.

The *dt* argument is ignored. The return value is a `timedelta` instance equal to the difference between the local time and UTC.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

timezone.**tzname**(*dt*)

Return the fixed value specified when the `timezone` instance is constructed.

If *name* is not provided in the constructor, the name returned by `tzname(dt)` is generated from the value of the `offset` as follows. If *offset* is `timedelta(0)`, the name is “UTC”, otherwise it is a string in the format `UTC±HH:MM`, where ± is the sign of `offset`, HH and MM are two digits of `offset.hours` and `offset.minutes` respectively.

Changed in version 3.6: Name generated from `offset=timedelta(0)` is now plain `'UTC'`, not `'UTC+00:00'`.

timezone.**dst**(*dt*)

Always returns `None`.

Class attributes:

timezone.**utc**

The UTC timezone, `timezone(timedelta(0))`.

`strftime()` and `strptime()` 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.

Conversely, the `datetime.strptime()` class method creates a `datetime` object from a string representing a date and time and a corresponding format string.

The table below provides a high-level comparison of `strftime()` versus `strptime()`:

	strftime	**strptime**
Usage	Convert object to a string according to a given format	Parse a string into a `datetime` object given a corresponding format
Type of method	Instance method	Class method
Method of	`date`; `datetime`; `time`	`datetime`
Signature	`strftime(format)`	`strptime(date_string, format)`

`strftime()` and `strptime()` Format Codes

These methods accept format codes that can be used to parse and format dates:

```
>>> datetime.strptime('31/01/22 23:59:59.999999',
...                   '%d/%m/%y %H:%M:%S.%f')
datetime.datetime(2022, 1, 31, 23, 59, 59, 999999)
>>> _.strftime('%a %d %b %Y, %I:%M%p')
'Mon 31 Jan 2022, 11:59PM'
```

The following is a list of all the format codes that the 1989 C standard requires, and these work on all platforms with a standard C implementation.

Directive	**Meaning**	**Example**	**Notes**
%a	Weekday as locale’s abbreviated name.	Sun, Mon, …, Sat (en_US); So, Mo, …, Sa (de_DE)	(1)
%A	Weekday as locale’s full name.	Sunday, Monday, …, Saturday (en_US); Sonntag, Montag, …, Samstag (de_DE)	(1)

`%w`	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.	0, 1, …, 6	
`%d`	Day of the month as a zero-padded decimal number.	01, 02, …, 31	(9)
`%b`	Month as locale’s abbreviated name.	Jan, Feb, …, Dec (en_US); Jan, Feb, …, Dez (de_DE)	(1)
`%B`	Month as locale’s full name.	January, February, …, December (en_US); Januar, Februar, …, Dezember (de_DE)	(1)
`%m`	Month as a zero-padded decimal number.	01, 02, …, 12	(9)
`%y`	Year without century as a zero-padded decimal number.	00, 01, …, 99	(9)
`%Y`	Year with century as a decimal number.	0001, 0002, …, 2013, 2014, …, 9998, 9999	(2)
`%H`	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, …, 23	(9)
`%I`	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, …, 12	(9)
`%p`	Locale’s equivalent of either AM or PM.	AM, PM (en_US); am, pm (de_DE)	(1), (3)
`%M`	Minute as a zero-padded decimal number.	00, 01, …, 59	(9)
`%S`	Second as a zero-padded decimal number.	00, 01, …, 59	(4), (9)
`%f`	Microsecond as a decimal number, zero-padded to 6 digits.	000000, 000001, …, 999999	(5)
`%z`	UTC offset in the form `±HHMM[SS[.ffffff]]` (empty string if the object is naive).	(empty), +0000, -0400, +1030, +063415, -030712.345216	(6)
`%Z`	Time zone name (empty string if the object is naive).	(empty), UTC, GMT	(6)
`%j`	Day of the year as a zero-padded decimal number.	001, 002, …, 366	(9)
`%U`	Week number of the year (Sunday as the first day of the week) as a zero-padded decimal number. All days in a new year preceding the first Sunday are considered to be in week 0.	00, 01, …, 53	(7), (9)
`%W`	Week number of the year (Monday as the first day of the week) as a zero-padded decimal number. All days in a new year preceding the first Monday are considered to be in week 0.	00, 01, …, 53	(7), (9)

%c	Locale's appropriate date and time representation.	Tue Aug 16 21:30:00 1988 (en_US); Di 16 Aug 21:30:00 1988 (de_DE)	(1)
%x	Locale's appropriate date representation.	08/16/88 (None); 08/16/1988 (en_US); 16.08.1988 (de_DE)	(1)
%X	Locale's appropriate time representation.	21:30:00 (en_US); 21:30:00 (de_DE)	(1)
%%	A literal `'%'` character.	%	

Several additional directives not required by the C89 standard are included for convenience. These parameters all correspond to ISO 8601 date values.

Directive	Meaning	Example	Notes
`%G`	ISO 8601 year with century representing the year that contains the greater part of the ISO week (`%V`).	0001, 0002, …, 2013, 2014, …, 9998, 9999	(8)
`%u`	ISO 8601 weekday as a decimal number where 1 is Monday.	1, 2, …, 7	
`%V`	ISO 8601 week as a decimal number with Monday as the first day of the week. Week 01 is the week containing Jan 4.	01, 02, …, 53	(8), (9)
`%:z`	UTC offset in the form `±HH:MM[:SS[.ffffff]]` (empty string if the object is naive).	(empty), +00:00, -04:00, +10:30, +06:34:15, -03:07:12.345216	(6)

These may not be available on all platforms when used with the `strftime()` method. The ISO 8601 year and ISO 8601 week directives are not interchangeable with the year and week number directives above. Calling `strptime()` with incomplete or ambiguous ISO 8601 directives will raise a `ValueError`.

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. To see the full set of format codes supported on your platform, consult the *strftime(3)* documentation. There are also differences between platforms in handling of unsupported format specifiers.

New in version 3.6: `%G`, `%u` and `%V` were added.

New in version 3.12: `%:z` was added.

Technical Detail

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 the `datetime.strptime()` class method, the default value is `1900-01-01T00:00:00.000`: any components not specified in the format string will be pulled from the default value. [4]

```
datetime(*(time.strptime(date_string, format)[0:6]))
```

except when the format includes sub-second components or timezone offset information, which are supported in `datetime.strptime` but are discarded by `time.strptime`.

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 `1` 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.

For the same reason, handling of format strings containing Unicode code points that can't be represented in the charset of the current locale is also platform-dependent. On some platforms such code points are preserved intact in the output, while on others `strftime` may raise `UnicodeError` or return an empty string instead.

Notes:

1. Because the format depends on the current locale, care should be taken when making assumptions about the output value. Field orderings will vary (for example, "month/day/year" versus "day/month/year"), and the output may contain non-ASCII characters.

2. The `strptime()` method can parse years in the full [1, 9999] range, but years < 1000 must be zero-filled to 4-digit width.

   Changed in version 3.2: In previous versions, `strftime()` method was restricted to years >= 1900.

   Changed in version 3.3: In version 3.2, `strftime()` method was restricted to years >= 1000.

3. When used with the `strptime()` method, the `%p` directive only affects the output hour field if the `%I` directive is used to parse the hour.

4. Unlike the `time` module, the `datetime` module does not support leap seconds.

5. When used with the `strptime()` method, 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).

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

   For an aware object:

   `%z`

   `utcoffset()` is transformed into a string of the form `±HHMM[SS[.ffffff]]`, where `HH` is a 2-digit string giving the number of UTC offset hours, `MM` is a 2-digit string giving the number of UTC offset minutes, `SS` is a 2-digit string giving the number of UTC offset seconds and `ffffff` is a 6-digit string giving the number of UTC offset microseconds. The `ffffff` part is omitted when the offset is a whole number of seconds and both the `ffffff` and the `SS` part is omitted when the offset is a whole number of minutes. For example, if `utcoffset()` returns `timedelta(hours=-3, minutes=-30)`, `%z` is replaced with the string `'-0330'`.

Changed in version 3.7: When the `%z` directive is provided to the `strptime()` method, the UTC offsets can have a colon as a separator between hours, minutes and seconds. For example, `'+01:00:00'` will be parsed as an offset of one hour. In addition, providing `'Z'` is identical to `'+00:00'`.

`%:z`
: Behaves exactly as `%z`, but has a colon separator added between hours, minutes and seconds.

`%Z`
: In `strftime()`, `%Z` is replaced by an empty string if `tzname()` returns `None`; otherwise `%Z` is replaced by the returned value, which must be a string.

  `strptime()` only accepts certain values for `%Z`:

  1. any value in `time.tzname` for your machine's locale
  2. the hard-coded values `UTC` and `GMT`

  So someone living in Japan may have `JST`, `UTC`, and `GMT` as valid values, but probably not `EST`. It will raise `ValueError` for invalid values.

Changed in version 3.2: When the `%z` directive is provided to the `strptime()` method, an aware `datetime` object will be produced. The `tzinfo` of the result will be set to a `timezone` instance.

7. When used with the `strptime()` method, `%U` and `%W` are only used in calculations when the day of the week and the calendar year (`%Y`) are specified.

8. Similar to `%U` and `%W`, `%V` is only used in calculations when the day of the week and the ISO year (`%G`) are specified in a `strptime()` format string. Also note that `%G` and `%Y` are not interchangeable.

9. When used with the `strptime()` method, the leading zero is optional for formats `%d`, `%m`, `%H`, `%I`, `%M`, `%S`, `%j`, `%U`, `%W`, and `%V`. Format `%y` does require a leading zero.

Footnotes

[1] If, that is, we ignore the effects of Relativity

[2] 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.

[3] See R. H. van Gent's guide to the mathematics of the ISO 8601 calendar for a good explanation.

[4] Passing `datetime.strptime('Feb 29', '%b %d')` will fail since `1900` is not a leap year.