



# datetime — Basic date and time types

Código fuente: [Lib/datetime.py](#)

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

Si bien la implementación permite operaciones aritméticas con fechas y horas, su principal objetivo es poder extraer campos de forma eficiente para su posterior manipulación o formateo.

**Truco:** Skip to [the format codes](#).

## Ver también:

### Módulo [calendar](#)

Funciones generales relacionadas a *calendar*.

### Módulo [time](#)

Acceso a tiempo y conversiones.

### Módulo [zoneinfo](#)

Zonas horarias concretas que representan la base de datos de zonas horarias de la IANA.

### Paquete [dateutil](#)

Biblioteca de terceros con zona horaria ampliada y soporte de análisis.

### Package [DateType](#)

Third-party library that introduces distinct static types to e.g. allow [static type checkers](#) to differentiate between naive and aware datetimes.

## Objetos conscientes (*aware*) y naífs (*naive*)

Los objetos de fecha y hora pueden clasificarse como conscientes (*aware*) o naífs (*naive*) dependiendo de si incluyen o no información de zona horaria.

Con suficiente conocimiento de los ajustes de tiempo políticos y algorítmicos aplicables, como la zona horaria y la información del horario de verano, un objeto **consciente** puede ubicarse en relación con otros objetos conscientes. Un objeto consciente representa un momento específico en el tiempo que no está abierto a interpretación. [\[1\]](#)

Un objeto **ingenuo** no contiene suficiente información para ubicarse sin ambigüedades en relación con otros objetos de fecha/hora. Si un objeto ingenuo representa la hora universal coordinada (UTC), la hora local o la hora en alguna otra zona horaria depende exclusivamente del programa, al igual que depende del programa si un número en particular representa metros, millas o masa. Los objetos ingenuos son fáciles de entender y trabajar con ellos, a costa de ignorar algunos aspectos de la realidad.



abstracta [tzinfo](#). Estos objetos [tzinfo](#) capturan información sobre el desplazamiento de la hora UTC, el nombre de la zona horaria y si el horario de verano está en vigor.

Only one concrete [tzinfo](#) class, the [timezone](#) class, is supplied by the [datetime](#) module. The [timezone](#) class can represent simple timezones with fixed offsets from UTC, such as UTC itself or North American EST and 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.

## Constantes

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 para el singleton de zona horaria UTC [datetime.timezone.utc](#).

*Added in version 3.11.*

## Tipos disponibles

### `class datetime.date`

Una fecha naíf (*naive*) idealizada, suponiendo que el calendario gregoriano actual siempre estuvo, y siempre estará, vigente. Atributos: [year](#), [month](#), y [day](#).

### `class datetime.time`

Un tiempo idealizado, independiente de cualquier día en particular, suponiendo que cada día tenga exactamente  $24^* 60^* 60$  segundos. (Aquí no hay noción de «segundos intercalares».) Atributos: [hour](#), [minute](#), [second](#), [microsecond](#), y [tzinfo](#).

### `class datetime.datetime`

Una combinación de una fecha y una hora. Atributos: [year](#), [month](#), [day](#), [hour](#), [minute](#), [second](#), [microsecond](#), y [tzinfo](#).

### `class datetime.timedelta`

A duration expressing the difference between two [datetime](#) or [date](#) instances to microsecond resolution.

### `class datetime.tzinfo`

Una clase base abstracta para objetos de información de zona horaria. Estos son utilizados por las clases [datetime](#) y [time](#) para proporcionar una noción personalizable de ajuste de hora (por ejemplo, para tener en



Q

## class datetime.timezone

Una clase que implementa la clase de base abstracta [tzinfo](#) como un desplazamiento fijo desde el UTC.

*Added in version 3.2.*

Los objetos de este tipo son inmutables.

Relaciones de subclase:

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

## Propiedades comunes

Las clases [date](#), [datetime](#), [time](#), y [timezone](#) comparten estas características comunes:

- Los objetos de este tipo son inmutables.
- Objects of these types are [hashable](#), meaning that they can be used as dictionary keys.
- Los objetos de este tipo admiten el *pickling* eficiente a través del módulo [pickle](#).

Determinando si un objeto es Consciente (*Aware*) o Naíf (*Naive*)

Los objetos del tipo [date](#) son siempre naíf (*naive*).

Un objeto de tipo [time](#) o [datetime](#) puede ser consciente (*aware*) o naíf (*naive*).

Un objeto [datetime](#) *d* es consciente si se cumplen los dos siguientes:

1. *d.tzinfo* no es `None`
2. *d.tzinfo.utcoffset(d)* no retorna `None`

De lo contrario, *d* es naíf (*naive*).

A [time](#) object *t* es consciente si ambos de los siguientes se mantienen:

1. *t.tzinfo* no es `None`
2. *t.tzinfo.utcoffset(None)* no retorna `None`.

De lo contrario, *t* es naíf (*naive*).

La distinción entre los objetos consciente (*aware*) y naíf (*naive*) no se aplica a [timedelta](#).

## Objetos [timedelta](#)



```
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.

Solo *days*, *seconds* y *microseconds* se almacenan internamente. Los argumentos se convierten a esas unidades:

- Un milisegundo se convierte a 1000 microsegundos.
- Un minuto se convierte a 60 segundos.
- Una hora se convierte a 3600 segundos.
- Una semana se convierte a 7 días.

y los días, segundos y microsegundos se normalizan para que la representación sea única, con

- $0 \leq \text{microsegundos} < 1000000$
- $0 \leq \text{segundos} < 3600*24$  (el número de segundos en un día)
- $-999999999 \leq \text{days} \leq 999999999$

El siguiente ejemplo ilustra cómo cualquier argumento además de *days*, *seconds* y *microseconds* se «fusionan» y normalizan en esos tres atributos resultantes:

```
>>> 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)
```

Si algún argumento es flotante y hay microsegundos fraccionarios, los microsegundos fraccionarios que quedan de todos los argumentos se combinan y su suma se redondea al microsegundo más cercano utilizando el desempate de medio redondeo a par. Si ningún argumento es flotante, los procesos de conversión y normalización son exactos (no se pierde información).

Si el valor normalizado de los días se encuentra fuera del rango indicado, se lanza [OverflowError](#).

Tenga en cuenta que la normalización de los valores negativos puede ser sorprendente al principio. Por ejemplo:

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



Q

### `timedelta.min`

El objeto más negativo en [timedelta](#), `timedelta(-999999999)`.

### `timedelta.max`

El objeto más positivo de la [timedelta](#), `timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)`.

### `timedelta.resolution`

La diferencia más pequeña posible entre los objetos no iguales [timedelta](#) `timedelta(microseconds=1)`.

Note that, because of normalization, `timedelta.max` is greater than `-timedelta.min`. `-timedelta.max` is not representable as a [timedelta](#) object.

Atributos de instancia (solo lectura):

Atributo	Valor
<code>days</code>	Entre -999999999 y 999999999 inclusive
<code>seconds</code>	Entre 0 y 86399 inclusive
<code>microseconds</code>	Entre 0 y 999999 inclusive

Operaciones soportadas:

Operación	Resultado
<code>t1 = t2 + t3</code>	Sum of <code>t2</code> and <code>t3</code> . Afterwards <code>t1 - t2 == t3</code> and <code>t1 - t3 == t2</code> are true. (1)
<code>t1 = t2 - t3</code>	Difference of <code>t2</code> and <code>t3</code> . Afterwards <code>t1 == t2 - t3</code> and <code>t2 == t1 + t3</code> are true. (1) (6)
<code>t1 = t2 * i</code> o <code>t1 = i * t2</code>	Delta multiplied by an integer. Afterwards <code>t1 // i == t2</code> is true, provided <code>i != 0</code> . In general, <code>t1 * i == t1 * (i-1) + t1</code> is true. (1)
<code>t1 = t2 * f</code> o <code>t1 = f * t2</code>	Delta multiplicado por un número decimal. El resultado se redondea al múltiplo mas cercano de <code>timedelta.resolution</code> usando redondeo de medio a par.
<code>f = t2 / t3</code>	Division (3) of overall duration <code>t2</code> by interval unit <code>t3</code> . Returns a <a href="#">float</a> object.
<code>t1 = t2 / f</code> o <code>t1 = t2 / i</code>	Delta dividido por un número decimal o un entero. El resultado se redondea al múltiplo más cercano de <code>timedelta.resolution</code> usando redondeo de medio a par.
<code>t1 = t2 // i</code> o <code>t1 = t2 // t3</code>	El piso ( <i>floor</i> ) se calcula y el resto (si lo hay) se descarta. En el segundo caso, se retorna un entero. (3)
<code>t1 = t2 % t3</code>	El resto se calcula como un objeto <a href="#">timedelta</a> . (3)



Q

<code>q, r = divmod(t1, t2)</code>	Calcula el cociente y el resto: <code>q = t1 // t2</code> (3) y <code>r = t1% t2</code> . <code>q</code> es un entero y <code>r</code> es un objeto <a href="#">timedelta</a> .
<code>+t1</code>	Retorna un objeto <a href="#">timedelta</a> con el mismo valor. (2)
<code>-t1</code>	Equivalent to <code>timedelta(-t1.days, -t1.seconds*, -t1.microseconds)</code> , and to <code>t1 * -1</code> . (1)(4)
<code>abs(t)</code>	Equivalent to <code>+t</code> when <code>t.days &gt;= 0</code> , and to <code>-t</code> when <code>t.days &lt; 0</code> . (2)
<code>str(t)</code>	Retorna una cadena de caracteres en la forma <code>[D day[s], ][H]H:MM:SS[.UUUUUU]</code> , donde <code>D</code> es negativo para negativo <code>t</code> . (5)
<code>repr(t)</code>	Retorna una representación de cadena del objeto <a href="#">timedelta</a> como una llamada de constructor con valores de atributos canónicos.

Notas:

1. Esto es exacto pero puede desbordarse.
2. Esto es exacto pero no puede desbordarse.
3. Division by zero raises [ZeroDivisionError](#).
4. `-timedelta.max` is not representable as a [timedelta](#) object.
5. Las representaciones de cadena de caracteres de los objetos [timedelta](#) se normalizan de manera similar a su representación interna. Esto conduce a resultados algo inusuales para *timedeltas* negativos. Por ejemplo:

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

&gt;&gt;&gt;

6. La expresión `t2 - t3` siempre será igual a la expresión `t2 + (-t3)` excepto cuando `t3` es igual a `timedelta.max`; en ese caso, el primero producirá un resultado mientras que el segundo se desbordará.

Además de las operaciones enumeradas anteriormente, los objetos [timedelta](#) admiten ciertas sumas y restas con objetos [date](#) y [datetime](#) (ver más abajo).

*Distinto en la versión 3.2:* La división entera a la baja y la división verdadera de un objeto [timedelta](#) entre otro [timedelta](#) ahora son compatibles, al igual que las operaciones de resto y la función [divmod\(\)](#). La división verdadera y multiplicación de un objeto [timedelta](#) por un objeto [float](#) ahora son compatibles.

[timedelta](#) objects support equality and order comparisons.

En contextos booleanos, un objeto [timedelta](#) se considera verdadero si y solo si no es igual a `timedelta(0)`.

Métodos de instancia:



Q

`timedelta(segundos=1)`. Para unidades de intervalo que no sean segundos, use la forma de división directamente (por ejemplo, `td / timedelta(microseconds=1)`).

Tenga en cuenta que para intervalos de tiempo muy largos (más de 270 años en la mayoría de las plataformas) este método perderá precisión de microsegundos.

*Added in version 3.2.*

## Ejemplos de uso: [timedelta](#)

Ejemplos adicionales de normalización:

```
>>> # 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
```

## Ejemplos de [timedelta](#) aritmética:

```
>>> 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)
```

## Objeto [date](#)

El objeto [date](#) representa una fecha (año, mes y día) en un calendario idealizado, el calendario gregoriano actual se extiende indefinidamente en ambas direcciones.

El 1 de enero del año 1 se llama día número 1, el 2 de enero del año 1 se llama día número 2, y así sucesivamente.

[2]

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

Todos los argumentos son obligatorios. Los argumentos deben ser enteros, en los siguientes rangos:

- `MINYEAR <= year <= MAXYEAR`



Si se proporciona un argumento fuera de esos rangos, [ValueError](#) se genera.

Otros constructores, todos los métodos de clase:

*classmethod* `date.today()`

Retorna la fecha local actual.

Esto es equivalente a `date.fromtimestamp(time.time())`.

*classmethod* `date.fromtimestamp(timestamp)`

Retorna la fecha local correspondiente a la marca de tiempo POSIX, tal como la retorna [time.time\(\)](#).

Esto puede generar [OverflowError](#), si la marca de tiempo está fuera del rango de valores admitidos por la plataforma C `localtime()`, y [OSError](#) en `localtime()` falla. Es común que esto se restrinja a años desde 1970 hasta 2038. Tenga en cuenta que en los sistemas que no son POSIX que incluyen segundos bisiestos en su noción de marca de tiempo, los segundos bisiestos son ignorados por [fromtimestamp\(\)](#).

*Distinto en la versión 3.3:* Se genera [OverflowError](#) en lugar de [ValueError](#) si la marca de tiempo está fuera del rango de valores admitidos por la plataforma C `localtime()`. Se genera [OSError](#) en lugar de [ValueError](#) cuando `localtime()`, falla.

*classmethod* `date.fromordinal(ordinal)`

Retorna la fecha correspondiente al ordinal gregoriano proleptico, donde el 1 de enero del año 1 tiene el ordinal 1.

[ValueError](#) se genera a menos que  $1 \leq \text{ordinal} \leq \text{date.max.toordinal()}$ . Para cualquier fecha  $d$ , `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, with the following exceptions:

1. Reduced precision dates are not currently supported (YYYY-MM, YYYY).
2. Extended date representations are not currently supported ( $\pm$ YYYYYY-MM-DD).
3. Ordinal dates are not currently supported (YYYY-000).

Ejemplos:

```
>>> 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)
```

*Added in version 3.7.*



### `classmethod date.fromisocalendar(year, week, day)`

Retorna [date](#) correspondiente a la fecha del calendario ISO especificada por año, semana y día. Esta es la inversa de la función [date.isocalendar\(\)](#).

*Added in version 3.8.*

Atributos de clase:

#### `date.min`

La fecha representable más antigua, `date(MINYEAR, 1, 1)`.

#### `date.max`

La última fecha representable, `date(MAXYEAR, 12, 31)`.

#### `date.resolution`

La menor diferencia entre objetos de fecha no iguales, `timedelta(days=1)`.

Atributos de instancia (solo lectura):

#### `date.year`

Entre [MINYEAR](#) y [MAXYEAR](#) inclusive.

#### `date.month`

Entre 1 y 12 inclusive.

#### `date.day`

Entre 1 y el número de días en el mes dado del año dado.

Operaciones soportadas:

Operación	Resultado
<code>date2 = date1 + timedelta</code>	<code>date2</code> will be <code>timedelta.days</code> days after <code>date1</code> . (1)
<code>date2 = date1 - timedelta</code>	Computes <code>date2</code> such that <code>date2 + timedelta == date1</code> . (2)
<code>timedelta = date1 - date2</code>	(3)
<code>date1 == date2</code> <code>date1 != date2</code>	Equality comparison. (4)
<code>date1 &lt; date2</code> <code>date1 &gt; date2</code> <code>date1 &lt;= date2</code> <code>date1 &gt;= date2</code>	Order comparison. (5)

Notas:



Q

- ran. [OverflowError](#) se lanza si `date2.year` sería menor que [MINYEAR](#) o mayor que [MAXYEAR](#).
2. `timedelta.seconds` y `timedelta.microseconds` son ignorados.
  3. This is exact, and cannot overflow. `timedelta.seconds` and `timedelta.microseconds` are 0, and `date2 + timedelta == date1` after.
  4. [date](#) objects are equal if they represent the same date.
  5. `date1` is considered less than `date2` when `date1` precedes `date2` in time. In other words, `date1 < date2` if and only if `date1.toordinal() < date2.toordinal()`.

En contextos booleanos, todos los objetos [date](#) se consideran verdaderos.

Métodos de instancia:

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

Retorna una fecha con el mismo valor, a excepción de aquellos parámetros dados nuevos valores por cualquier argumento de palabra clave especificado.

Ejemplo:

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

`date.timetuple()`

Retorna una [time.struct\\_time](#) como la que retorna [time.localtime\(\)](#).

Las horas, minutos y segundos son 0, y el indicador DST es -1.

`d.timetuple()` es equivalente a:

```
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()`

Retorna el ordinal gregoriano proléptico de la fecha, donde el 1 de enero del año 1 tiene el ordinal 1. Para cualquiera [date](#) object `d`, `date.fromordinal(d.toordinal()) == d`.

`date.weekday()`

Retorna el día de la semana como un número entero, donde el lunes es 0 y el domingo es 6. Por ejemplo, `date(2002, 12, 4).weekday() == 2`, un miércoles. Ver también [isoweekday\(\)](#).

`date.isoweekday()`

Retorna el día de la semana como un número entero, donde el lunes es 1 y el domingo es 7. Por ejemplo, `date(2002, 12, 4).isoweekday() == 3`, un miércoles. Ver también [weekday\(\)](#), [isocalendar\(\)](#).



Q

El calendario ISO es una variante amplia utilizada del calendario gregoriano. [\[3\]](#)

El año ISO consta de 52 o 53 semanas completas, y donde una semana comienza un lunes y termina un domingo. La primera semana de un año ISO es la primera semana calendario (gregoriana) de un año que contiene un jueves. Esto se llama semana número 1, y el año ISO de ese jueves es el mismo que el año gregoriano.

Por ejemplo, 2004 comienza en jueves, por lo que la primera semana del año ISO 2004 comienza el lunes 29 de diciembre de 2003 y termina el domingo 4 de enero de 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)
```

*Distinto en la versión 3.9:* El resultado cambió de una tupla a un [named tuple](#).

### date.isoformat()

Retorna una cadena de caracteres que representa la fecha en formato ISO 8601, AAAA-MM-DD:

```
>>> from datetime import date
>>> date(2002, 12, 4).isoformat()
'2002-12-04'
```

### date.\_\_str\_\_()

Para una fecha *d*, str(*d*) es equivalente a *d.isoformat()*.

### date.ctime()

Retorna una cadena de caracteres que representa la fecha:

```
>>> from datetime import date
>>> date(2002, 12, 4).ctime()
'Wed Dec 4 00:00:00 2002'
```

*d.ctime()* es equivalente a:

```
time.ctime(time.mktime(d.timetuple()))
```

en plataformas donde la función nativa C ctime() (donde [time.ctime\(\)](#) llama, pero que [date.ctime\(\)](#) no se llama) se ajusta al estándar C.

### 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)

[date.isoformat\(\)](#).Ejemplos de uso: [date](#)

Ejemplo de contar días para un evento:

```
>>> 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
```

Más ejemplos de trabajo con [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'
>>> dctime()
'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 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
```



... **print()**

```

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)

```

## Objetos [datetime](#)

El objeto [datetime](#) es un único objeto que contiene toda la información de un objeto [date](#) y un objeto [time](#).

Como un objeto [date](#), [datetime](#) asume el calendario gregoriano actual extendido en ambas direcciones; como un objeto [time](#), [datetime](#) supone que hay exactamente 3600\*24 segundos en cada día.

Constructor:

```
class datetime.datetime(year, month, day, hour=0, minute=0, second=0, microsecond=0,
tzinfo=None, *, fold=0)
```

Se requieren los argumentos `year`, `month` y `day`. `tzinfo` puede ser `None`, o una instancia de una subclase [tzinfo](#). Los argumentos restantes deben ser enteros en los siguientes rangos:

- `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]`.

Si se proporciona un argumento fuera de esos rangos, [ValueError](#) se genera.

 *Distinto en la versión 3.6:* Added the `fold` parameter.

Otros constructores, todos los métodos de clase:

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

Retorna la fecha y hora local actual, con [tzinfo](#) `None`.

Equivalente a:

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

Ver también [now\(\)](#), [fromtimestamp\(\)](#).

Este método es funciona como [now\(\)](#), pero sin un parámetro `tz`.



Q

Si el argumento opcional `tz` es `None` o no se especifica, es como [today\(\)](#), pero, si es posible, proporciona más precisión de la que se puede obtener al pasar por [time.time\(\)](#) marca de tiempo (por ejemplo, esto puede ser posible en plataformas que suministran la función C `gettimeofday()`).

Si `tz` no es `None`, debe ser una instancia de una subclase [tzinfo](#), y la fecha y hora actuales se convierten en la zona horaria de `tz`.

Esta función es preferible a [today\(\)](#) y [utcnow\(\)](#).

### Classmethod datetime.utcnow()

Retorna la fecha y hora UTC actual, con [tzinfo](#) `None`.

Esto es como [now\(\)](#), pero retorna la fecha y hora UTC actual, como un objeto naíf (*naive*): [datetime](#). Se puede obtener una fecha y hora UTC actual consciente (*aware*) llamando a `datetime.now(timezone.utc)`. Ver también [now\(\)](#).

**Advertencia:** Debido a que los objetos naífs (*naive*) de `datetime` son tratados por muchos métodos de `datetime` como horas locales, se prefiere usar fechas y horas conscientes(*aware*) para representar las horas en UTC. Como tal, la forma recomendada de crear un objeto que represente la hora actual en UTC es llamando a `datetime.now(timezone.utc)`.

*Obsoleto desde la versión 3.12:* Use [datetime.now\(\)](#) with [UTC](#) instead.

### Classmethod datetime.fromtimestamp(timestamp, tz=None)

Retorna la fecha y hora local correspondiente a la marca de tiempo POSIX, tal como la retorna [time.time\(\)](#).

Si el argumento opcional `tz` es `None` o no se especifica, la marca de tiempo se convierte a la fecha y hora local de la plataforma, y el objeto returnedo [datetime](#) es naíf (*naive*).

Si `tz` no es `None`, debe ser una instancia de una subclase [tzinfo](#), y la fecha y hora actuales se convierten en la zona horaria de `tz`.

[fromtimestamp\(\)](#) puede aumentar [OverflowError](#), si la marca de tiempo está fuera del rango de valores admitidos por la plataforma C `localtime()` o `gmtime()`, y [OSError](#) en `localtime()` o `gmtime()` falla. Es común que esto se restrinja a los años 1970 a 2038. Tenga en cuenta que en los sistemas que no son POSIX que incluyen segundos bisiestos en su noción de marca de tiempo, los segundos bisiestos son ignorados por [fromtimestamp\(\)](#), y luego es posible tener dos marcas de tiempo que difieren en un segundo que producen objetos idénticos [datetime](#). Se prefiere este método sobre [utcfromtimestamp\(\)](#).

*Distinto en la versión 3.3:* Se genera [OverflowError](#) en lugar de [ValueError](#) si la marca de tiempo está fuera del rango de valores admitidos por la plataforma C `localtime()` o `gmtime()`. genera [OSError](#) en lugar de la función [ValueError](#) en `localtime()` o error `gmtime()`.

*Distinto en la versión 3.6:* [fromtimestamp\(\)](#) puede retornar instancias con [fold](#) establecido en 1.



tante es naïf (*naive*)).

Esto puede generar [OverflowError](#), si la marca de tiempo está fuera del rango de valores admitidos por la plataforma C `gmtime()`, y error en [OSError](#) en `gmtime()`. Es común que esto se restrinja a los años entre 1970 a 2038.

Para conocer un objeto [datetime](#), llama a [fromtimestamp\(\)](#):

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

En las plataformas compatibles con POSIX, es equivalente a la siguiente expresión:

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

excepto que la última fórmula siempre admite el rango de años completo: entre [MINYEAR](#) y [MAXYEAR](#) inclusive.

**Advertencia:** Debido a que los objetos naïf (*naive*) de `datetime` son tratados por muchos métodos de `datetime` como horas locales, se prefiere usar fechas y horas conscientes para representar las horas en UTC. Como tal, la forma recomendada de crear un objeto que represente una marca de tiempo específica en UTC es llamando a `datetime.fromtimestamp(timestamp, tz=timezone.utc)`.

*Distinto en la versión 3.3:* Se genera [OverflowError](#) en lugar de [ValueError](#) si la marca de tiempo está fuera del rango de valores admitidos por la plataforma C `gmtime()`. genera [OSError](#) en lugar de [ValueError](#) en el error de `gmtime()`.

*Obsoleto desde la versión 3.12:* Use [datetime.fromtimestamp\(\)](#) with [UTC](#) instead.

### `classmethod datetime.fromordinal(ordinal)`

Se genera [datetime](#) correspondiente al ordinal del proleptico gregoriano, donde el 1 de enero del año 1 tiene ordinal 1. [ValueError](#) se genera a menos que  $1 \leq \text{ordinal} \leq \text{datetime.max.toordinal()}$ . La hora, minuto, segundo y microsegundo del resultado son todos 0, y [tzinfo](#) es 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  $d$ ,  $d == \text{datetime.combine}(d.\text{date}(), d.\text{time}(), d.\text{tzinfo})$ .

*Distinto en la versión 3.6:* Se agregó el argumento [tzinfo](#).

### `classmethod datetime.fromisoformat(date_string)`

Devuelve un [datetime](#) correspondiente a un [date\\_string](#) en cualquier formato ISO 8601 válido, con las siguientes excepciones:



Q

3. No se admiten fracciones de horas y minutos.
4. Reduced precision dates are not currently supported (YYYY-MM, YYYY).
5. Extended date representations are not currently supported ( $\pm$ YYYYYY-MM-DD).
6. Ordinal dates are not currently supported (YYYY-000).

Ejemplos:

```
>>> 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)))
```

*Added in version 3.7.*

*Distinto en la versión 3.11:* Anteriormente, este método solo admitía formatos que podían ser emitidos por [date.isoformat\(\)](#) o [datetime.isoformat\(\)](#).

*classmethod* `datetime.fromisocalendar(year, week, day)`

Retorna [datetime](#) correspondiente a la fecha del calendario ISO especificada por año, semana y día. Los componentes que no son de fecha de fecha y hora se rellenan con sus valores predeterminados normales. Esta es la inversa de la función [datetime.isocalendar\(\)](#).

*Added in version 3.8.*

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

Retorna [datetime](#) correspondiente a `date_string`, analizado según `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\(\)](#).



## datetime.min

La primera fecha representable [datetime](#), `datetime(MINYEAR, 1, 1, tzinfo=None)`.

## datetime.max

La última fecha representable [datetime](#), `datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None)`.

## datetime.resolution

La diferencia más pequeña posible entre objetos no iguales [datetime](#), `timedelta(microseconds=1)`.

Atributos de instancia (solo lectura):

### datetime.year

Entre [MINYEAR](#) y [MAXYEAR](#) inclusive.

### datetime.month

Entre 1 y 12 inclusive.

### datetime.day

Entre 1 y el número de días en el mes dado del año dado.

### datetime.hour

En `range(24)`.

### datetime.minute

En `range(60)`.

### datetime.second

En `range(60)`.

### datetime.microsecond

En `range(1000000)`.

### datetime.tzinfo

El objeto pasó como argumento `tzinfo` al constructor [datetime](#), o `None` si no se pasó ninguno.

### 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 values 0 and 1 represent, respectively, the earlier and later of the two moments with the same wall time representation.

*Added in version 3.6.*

Operaciones soportadas:




<code>datetime2 = datetime1 + timedelta</code>	(1)
<code>datetime2 = datetime1 - timedelta</code>	(2)
<code>timedelta = datetime1 - datetime2</code>	(3)
<code>datetime1 == datetime2</code> <code>datetime1 != datetime2</code>	Equality comparison. (4)
<code>datetime1 &lt; datetime2</code> <code>datetime1 &gt; datetime2</code> <code>datetime1 &lt;= datetime2</code> <code>datetime1 &gt;= datetime2</code>	Order comparison. (5)

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. La resta de [datetime](#) de la [datetime](#) se define solo si ambos operandos son naíf(*naive*), o si ambos son conscientes(*aware*). Si uno es consciente y el otro es naíf, [TypeError](#) aparece.  
Si ambos son naíf (*naive*), o ambos son conscientes (*aware*) y tienen el mismo atributo [tzinfo](#), los atributos [tzinfo](#) se ignoran y el resultado es un objeto de `timedelta` *t*` tal que ``datetime2 + t == datetime1``. No se realizan ajustes de zona horaria en este caso.  
If both are aware and have different tzinfo attributes, a-b acts as if a and b were first converted to naive UTC datetimes. The result is (a.replace(tzinfo=None) - a.utcoffset()) - (b.replace(tzinfo=None) - b.utcoffset()) except that the implementation never overflows.`
4. [datetime](#) objects are equal if they represent the same date and time, taking into account the time zone.

Naive and aware `datetime` objects are never equal. `datetime` objects are never equal to [date](#) objects that are not also `datetime` instances, even if they represent the same date.

If both comparands are aware, and have the same `tzinfo` attribute, the `tzinfo` and [fold](#) attributes are ignored and the base datetimes are compared. If both comparands are aware and have different [tzinfo](#) attributes, the comparison acts as comparands were first converted to UTC datetimes except that the implementation never overflows. `datetime` instances in a repeated interval are never equal to `datetime` instances in other time zone.



Order comparison between naive and aware [datetime](#) objects, as well as a [datetime](#) object and a [date](#) object that is not also a [datetime](#) instance, raises [TypeError](#).

If both comparands are aware, and have the same `tzinfo` attribute, the `tzinfo` and `fold` attributes are ignored and the base datetimes are compared. If both comparands are aware and have different `tzinfo` attributes, the comparison acts as comparands were first converted to UTC datetimes except that the implementation never overflows.

*Distinto en la versión 3.3:* Las comparaciones de igualdad entre las instancias conscientes (*aware*) y naïf (*naive*) [datetime](#) no generan [TypeError](#).

Métodos de instancia:

### `datetime.date()`

Retorna el objeto [date](#) con el mismo año, mes y día.

### `datetime.time()`

Retorna el objeto [time](#) con la misma hora, minuto, segundo, microsegundo y doblado(*fold*). [tzinfo](#) es None.

Ver también método [timetz\(\)](#).

*Distinto en la versión 3.6:* El valor de plegado (*fold value*) se copia en el objeto [time](#) returned.

### `datetime.timetz()`

Retorna el objeto [time](#) con los mismos atributos de hora, minuto, segundo, microsegundo, pliegue y `tzinfo`.

Ver también método [time\(\)](#).

*Distinto en la versión 3.6:* El valor de plegado (*fold value*) se copia en el objeto [time](#) returned.

### `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)`

Retorna una fecha y hora con los mismos atributos, a excepción de aquellos atributos a los que se les asignan nuevos valores según los argumentos de palabras clave especificados. Tenga en cuenta que `tzinfo = None` se puede especificar para crear una fecha y hora naïf (*naive*) a partir de una fecha y hora consciente(*aware*) sin conversión de datos de fecha y hora.

*Distinto en la versión 3.6:* Added the *fold* parameter.

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

Retorna un objeto [datetime](#) con el atributo nuevo `tzinfo` `tz`, ajustando los datos de fecha y hora para que el resultado sea la misma hora UTC que `self`, pero en hora local `tz`.

Si se proporciona, `tz` debe ser una instancia de una subclase [tzinfo](#), y sus métodos [utcoffset\(\)](#) y [dst\(\)](#) no deben retornar `None`. Si `self` es naïf, se supone que representa la hora en la zona horaria del sistema.



Q

[timezone](#) con el nombre de zona y el desplazamiento obtenido del sistema operativo.

Si `self.tzinfo` es `tz`, `self.astimezone(tz)` es igual a `self`. no se realiza ningún ajuste de datos de fecha u hora. De lo contrario, el resultado es la hora local en la zona horaria `tz`, que representa la misma hora UTC que `self`. después de `astz = dt.astimezone(tz)`, `astz - astz.utcoffset()` tendrá los mismos datos de fecha y hora que `dt - dt.utcoffset()`.

Si simplemente desea adjuntar un objeto de zona horaria `tz` a una fecha y hora `dt` sin ajustar los datos de fecha y hora, use `dt.replace(tzinfo=tz)`. Si simplemente desea eliminar el objeto de zona horaria de una fecha y hora `dt` sin conversión de datos de fecha y hora, use `dt.replace(tzinfo=None)`.

Tenga en cuenta que el método predeterminado [`tzinfo.fromutc\(\)`](#) se puede reemplazar en una subclase [`tzinfo`](#) para afectar el resultado returned por [`astimezone\(\)`](#). [`astimezone\(\)`](#) ignora los casos de error, actúa como:

```
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)
```

*Distinto en la versión 3.3:* `tz` ahora puede ser omitido.

*Distinto en la versión 3.6:* El método [`astimezone\(\)`](#) ahora se puede invocar en instancias naïf (*naive*) que se supone representan la hora local del sistema.

### `datetime.utcnow()`

Si [`tzinfo`](#) es `None`, retorna `None`, de lo contrario retorna `self.tzinfo.utcoffset(self)`, y genera una excepción si este último no retorna `None` o un objeto [`timedelta`](#) con magnitud inferior a un día.

*Distinto en la versión 3.7:* El desfase UTC no está restringido a un número entero de minutos.

### `datetime.dst()`

Si [`tzinfo`](#) es `None`, retorna `None`, de lo contrario retorna `self.tzinfo.utcoffset(self)`, y genera una excepción si este último no retorna `None` o un objeto [`timedelta`](#) con magnitud inferior a un día.

*Distinto en la versión 3.7:* El desfase DST no está restringido a un número entero de minutos.

### `datetime.tzname()`

Si [`tzinfo`](#) es `None`, retorna `None`, de lo contrario retorna `self.tzinfo.tzname(self)`, genera una excepción si este último no retorna `None` o un objeto de cadena de caracteres,

### `datetime.timetuple()`

Retorna una [`time.struct\_time`](#) como la que retorna [`time.localtime\(\)`](#).



Q

```
time.struct_time((d.year, d.month, d.day,
                  d.hour, d.minute, d.second,
                  d.weekday(), yday, dst))
```

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.

**Advertencia:** 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()`

Retorna el ordinal gregoriano proleptico de la fecha. Lo mismo que `self.date().toordinal()`.

### `datetime.timestamp()`

Retorna la marca de tiempo (`timestamp`) POSIX correspondiente a la instancia `datetime`. El valor de retorno es `float` similar al retornado por `time.time()`.

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

Para las instancias de `datetime`, el valor de retorno se calcula como:

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

*Added in version 3.3.*

*Distinto en la versión 3.6:* El método `timestamp()` utiliza el atributo `fold` para desambiguar los tiempos durante un intervalo repetido.



Instancia para [datetime](#) que representa la hora UTC. Si su aplicación utiliza esta convención y la zona horaria de su sistema no está configurada en UTC, puede obtener la marca de tiempo POSIX al proporcionar `tzinfo = timezone.utc`:

```
timestamp = dt.replace(tzinfo=timezone.utc).timestamp()
```

o calculando la marca de tiempo (`timestamp`) directamente:

```
timestamp = (dt - datetime(1970, 1, 1)) / timedelta(seconds=1)
```

## `datetime.weekday()`

Retorna el día de la semana como un entero, donde el lunes es 0 y el domingo es 6. Lo mismo que `self.date().weekday()`. Ver también [isoweekday\(\)](#).

## `datetime.isoweekday()`

Retorna el día de la semana como un número entero, donde el lunes es 1 y el domingo es 7. Lo mismo que `self.date().isoweekday()`. Ver también [weekday\(\)](#), [isocalendar\(\)](#).

## `datetime.isocalendar()`

Retorna una [named tuple](#) con tres componentes: `year`, `week`, y `weekday`. Lo mismo que `self.date().isocalendar()`.

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

Retorna una cadena de caracteres representando la fecha y la hora en formato ISO 8601:

- YYYY-MM-DDTHH:MM:SS.fffffff, si [microsecond](#) no es 0
- YYYY-MM-DDTHH:MM:SS, si [microsecond](#) es 0

Si [utcoffset\(\)](#) no retorna None, se agrega una cadena de caracteres dando el desplazamiento UTC:

- YYYY-MM-DDTHH:MM:SS.fffffff+HH:MM[:SS[.fffffff]], si [microsecond](#) no es 0
- YYYY-MM-DDTHH:MM:SS+HH:MM[:SS[.fffffff]], si [microsecond](#) es 0

Ejemplos:

```
>>> 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'
```

&gt;&gt;&gt;

El argumento opcional `sep` (default '`T`') es un separador de un carácter, ubicado entre las porciones de fecha y hora del resultado. Por ejemplo:

```
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     """A time zone with an arbitrary, constant -06:39 offset."""
...     def utcoffset(self, dt):
```

&gt;&gt;&gt;



Q

```
>>> 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'
```

El argumento opcional *timespec* especifica el número de componentes adicionales del tiempo a incluir (el valor predeterminado es ‘auto’). Puede ser uno de los siguientes:

- ‘auto’: Igual que ‘seconds’ si microsecond es 0, igual que ‘microseconds’ de lo contrario.
- ‘hours’: incluye el hour en el formato de dos dígitos HH.
- ‘minutes’: Incluye hour y minute en formato HH:MM.
- ‘seconds’: Incluye hour, minute, y second en formato HH:MM:SS.
- ‘milliseconds’: Incluye tiempo completo, pero trunca la segunda parte fraccionaria a milisegundos. Formato HH:MM:SS.sss.
- ‘microseconds’: Incluye tiempo completo en formato HH:MM:SS.fffffff.

**Nota:** Los componentes de tiempo excluidos están truncados, no redondeados.

ValueError lanzará un argumento inválido *timespec*:

```
>>> 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'
```

*Distinto en la versión 3.6:* Added the *timespec* parameter.

### datetime.\_\_str\_\_()

Para una instancia de la datetime *d*, *str(d)* es equivalente a *d.isoformat(' ')*.

### datetime.ctime()

Retorna una cadena de caracteres que representa la fecha y la hora:

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

La cadena de salida *no* incluirá información de zona horaria, independientemente de si la entrada es consciente (*aware*) o naíf (*naive*).

*d.ctime()* es equivalente a:

```
time.ctime(time.mktime(d.timetuple()))
```

en plataformas donde la función nativa C *ctime()* (que time.ctime() invoca, pero que datetime.ctime() no invoca) se ajusta al estándar C.



[strptime\(\) Behavior](#) and [datetime.isoformat\(\)](#).

### datetime.\_\_format\_\_(format)

Same as [datetime.strptime\(\)](#). 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\(\)](#).

Ejemplos de uso: [datetime](#)

Examples of working with [datetime](#) objects:

```
>>> from datetime import datetime, date, time, timezone
>>> # 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
```



TUESDAY, 21. NOVEMBER 2000 04:30PM  
 >>> 'The {1} is {0:%d}, the {2} is {0:%B}, the {3} is {0:%I:%M%p}'.format(dt, "day", "month",  
 'The day is 21, the month is November, the time is 04:30PM.'

El siguiente ejemplo define una subclase `tzinfo` que captura la información de zona horaria de *Kabul*, Afganistán, que utilizó +4 UTC hasta 1945 y +4:30 UTC a partir de entonces:

```
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)

    def utcoffset(self, dt):
        if dt.year < 1945:
            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"
```

Uso de `KabulTz` desde arriba

```
>>> tz1 = KabulTz()
>>> # Datetime before the change
```

>>>



Q

```
>>> # 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
```

## Objetos [time](#)

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)`

Todos los argumentos son opcionales. `tzinfo` puede ser `None`, o una instancia de una subclase [tzinfo](#). Los argumentos restantes deben ser enteros en los siguientes rangos:

- `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`.

Atributos de clase:

**time.min**

El primero representable [time](#), ``time (0, 0, 0, 0)``.

**time.max**

El último representable [time](#), `time(23, 59, 59, 999999)`.

**time.resolution**

La diferencia más pequeña posible entre los objetos no iguales [time](#), `timedelta(microseconds=1)`, aunque tenga en cuenta que la aritmética en objetos [time](#) no es compatible.

Atributos de instancia (solo lectura):

**time.hour**

En `range(24)`.



Q

### time.second

En range(60).

### time.microsecond

En range(1000000).

### time.tzinfo

El objeto pasado como argumento `tzinfo` al constructor de la clase [time](#), o `None` si no se pasó ninguno.

### time.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 values 0 and 1 represent, respectively, the earlier and later of the two moments with the same wall time representation.

*Added in version 3.6.*

[time](#) objects support equality and order comparisons, where *a* is considered less than *b* when *a* precedes *b* in time.

Naive and aware `time` objects are never equal. Order comparison between naive and aware `time` objects raises [TypeError](#).

If both comparands are aware, and have the same `tzinfo` attribute, the `tzinfo` and `fold` attributes are 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()`).

*Distinto en la versión 3.3:* Equality comparisons between aware and naive `time` instances don't raise [TypeError](#).

En contextos booleanos, un objeto `time` siempre se considera verdadero.

*Distinto en la versión 3.5:* Antes de Python 3.5, un objeto `time` se consideraba falso si representaba la medianoche en UTC. Este comportamiento se consideró oscuro y propenso a errores y se ha eliminado en Python 3.5. Ver [bpo-13936](#) para más detalles.

Otro constructor:

### Classmethod `time.fromisoformat(time_string)`

Devuelve un `time` correspondiente a un `time_string` en cualquier formato ISO 8601 válido, con las siguientes excepciones:

1. Las compensaciones de zona horaria pueden tener fracciones de segundo.
2. No se requiere el T inicial, que normalmente se requiere en los casos en que puede haber ambigüedad entre una fecha y una hora.



Q

4. No se admiten fracciones de horas y minutos.

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('04:23:01,000384')
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)
>>> time.fromisoformat('04:23:01+00:00')
datetime.time(4, 23, 1, tzinfo=datetime.timezone.utc)
```

*Added in version 3.7.*

*Distinto en la versión 3.11:* Anteriormente, este método solo admitía formatos que podía emitir [time.isoformat\(\)](#).

Métodos de instancia:

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

Retorna una [time](#) con el mismo valor, excepto para aquellos atributos a los que se les otorgan nuevos valores según los argumentos de palabras clave especificados. Tenga en cuenta que `tzinfo = None` se puede especificar para crear una [time](#) naíf (*naive*) desde un consciente (*aware*) [time](#), sin conversión de los datos de tiempo.

*Distinto en la versión 3.6:* Added the `fold` parameter.

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

Retorna una cadena que representa la hora en formato ISO 8601, una de:

- HH:MM:SS.fffffff, si [microsecond](#) no es 0
- HH:MM:SS, si [microsecond](#) es 0
- HH:MM:SS.fffffff+HH:MM[:SS[.fffffff]], si [utcoffset\(\)](#) no retorna None
- HH:MM:SS+HH:MM[:SS[.fffffff]], si [microsecond](#) es 0 y [utcoffset\(\)](#) no retorna None

El argumento opcional `timespec` especifica el número de componentes adicionales del tiempo a incluir (el valor predeterminado es ‘auto’). Puede ser uno de los siguientes:

- ‘auto’: Igual que ‘seconds’ si [microsecond](#) es 0, igual que ‘microseconds’ de lo contrario.



Q

- 'seconds': Incluye `hour`, `minute`, y `second` en formato HH:MM:SS.
- 'milliseconds': Incluye tiempo completo, pero trunca la segunda parte fraccionaria a milisegundos. Formato HH:MM:SS.sss.
- 'microseconds': Incluye tiempo completo en formato HH:MM:SS.fffffff.

**Nota:** Los componentes de tiempo excluidos están truncados, no redondeados.

`ValueError` lanzará un argumento inválido `timespec`.

Ejemplo:

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

*Distinto en la versión 3.6:* Added the `timespec` parameter.

### time.\_\_str\_\_()

Durante un tiempo `t`, `str(t)` es equivalente a `t.isoformat()`.

### time.strftime(format)

Return a string representing the time, controlled by an explicit format string. See also [strftime\(\) and strptime\(\) Behavior](#) and [time.isoformat\(\)](#).

### 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.utcnowoffset()

Si `tzinfo` es `None`, retorna `None`, sino retorna `self.tzinfo.utcoffset(None)`, y genera una excepción si este último no retorna `None` o un objeto de [timedelta](#) con magnitud inferior a un día.

*Distinto en la versión 3.7:* El desfase UTC no está restringido a un número entero de minutos.

### time.dst()

Si `tzinfo` es `None`, retorna `None`, sino retorna `self.tzinfo.utcoffset(None)`, y genera una excepción si este último no retorna `None`, o un objeto de [timedelta](#) con magnitud inferior a un día.

*Distinto en la versión 3.7:* El desfase DST no está restringido a un número entero de minutos.

### time.tzname()



Q

## Ejemplos de uso: [time](#)

Ejemplos de trabajo con el objeto [time](#):

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

## Objetos [tzinfo](#)

### *class* `datetime.tzinfo`

Esta es una clase base abstracta, lo que significa que esta clase no debe ser instanciada directamente. Defina una subclase de [tzinfo](#) para capturar información sobre una zona horaria particular.

Una instancia (de una subclase concreta) [tzinfo](#) se puede pasar a los constructores para objetos de [datetime](#) y [time](#). Los últimos objetos ven sus atributos como en la hora local, y el objeto [tzinfo](#) admite métodos que revelan el desplazamiento de la hora local desde UTC, el nombre de la zona horaria y el desplazamiento DST, todo en relación con un objeto de fecha u hora pasado a ellos.

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.



## `tzinfo.utcoffset(dt)`

Retorna el desplazamiento de la hora local desde UTC, como un objeto [timedelta](#) que es positivo al este de UTC. Si la hora local es al oeste de UTC, esto debería ser negativo.

Esto representa el desplazamiento *total* de UTC; por ejemplo, si un objeto [tzinfo](#) representa ajustes de zona horaria y DST, [utcoffset\(\)](#) debería retornar su suma. Si no se conoce el desplazamiento UTC, retorna `None`. De lo contrario, el valor devuelto debe ser un objeto de [timedelta](#) estrictamente entre `-timedelta(hours = 24)` y `timedelta(hours = 24)` (la magnitud del desplazamiento debe ser inferior a un día). La mayoría de las implementaciones de [utcoffset\(\)](#) probablemente se parecerán a una de estas dos:

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

Si [utcoffset\(\)](#) no retorna `None`, [dst\(\)](#) no debería retornar `None` tampoco.

La implementación por defecto de [utcoffset\(\)](#) genera [NotImplementedError](#).

**Distinto en la versión 3.7:** El desfase UTC no está restringido a un número entero de minutos.

## `tzinfo.dst(dt)`

Retorna el ajuste del horario de verano (DST), como un objeto de [timedelta](#) o `None` si no se conoce la información de DST.

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.

Una instancia `tz` de una subclase [tzinfo](#) que modela los horarios estándar y diurnos debe ser coherente en este sentido:

```
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.

La mayoría de las implementaciones de [dst\(\)](#) probablemente se parecerán a una de estas dos:



```
return timedelta(0)
```

O:

```
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)
```

La implementación predeterminada de [dst\(\)](#) genera [NotImplementedError](#).

*Distinto en la versión 3.7:* El desfase DST no está restringido a un número entero de minutos.

### `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.

La implementación predeterminada de [tzname\(\)](#) genera [NotImplementedError](#).

Estos métodos son llamados por un objeto [datetime](#) o [time](#), en respuesta a sus métodos con los mismos nombres. El objeto de [datetime](#) se pasa a sí mismo como argumento, y un objeto [time](#) pasa a None como argumento. Los métodos de la subclase [tzinfo](#) deben, por lo tanto, estar preparados para aceptar un argumento *dt* de None, o de clase [datetime](#).

Cuando se pasa None, corresponde al diseñador de la clase decidir la mejor respuesta. Por ejemplo, retornar None es apropiado si la clase desea decir que los objetos de tiempo no participan en los protocolos [tzinfo](#). Puede ser más útil que `utcoffset(None)` retorne el desplazamiento UTC estándar, ya que no existe otra convención para descubrir el desplazamiento estándar.

Cuando se pasa un objeto [datetime](#) en respuesta a un método [datetime](#), `dt.tzinfo` es el mismo objeto que `self.tzinfo`. Los métodos pueden confiar en esto, a menos que el código del usuario llame [tzinfo](#) métodos directamente. La intención es que los métodos [tzinfo](#) interpreten *dt* como si estuvieran en la hora local, y no necesiten preocuparse por los objetos en otras zonas horarias.

Hay un método más [tzinfo](#) que una subclase puede desear anular:

### `tzinfo.fromutc(dt)`




sito de [fromutc\(\)](#) es ajustar los datos de fecha y hora, retornando una fecha y hora equivalente en la hora local de *self*.

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 day-light 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.

Código de omisión para casos de error, el valor predeterminado [fromutc\(\)](#) la implementación actúa como

```
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
```

En el siguiente archivo [tzinfo\\_examples.py](#) hay algunos ejemplos de clases [tzinfo](#):

```
from datetime import tzinfo, timedelta, datetime

ZERO = timedelta(0)
HOUR = timedelta(hours=1)
SECOND = timedelta(seconds=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):
```




```

stamp = _ut - datetime(1970, 1, 1, tzinfo=SET) // 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)
    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
# 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

```

```
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:
            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
        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:
```




```

    # 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.dstoffset
    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")

```

Tenga en cuenta que hay sutilezas inevitables dos veces al año en una subclase [tzinfo](#) que representa tanto el horario estándar como el horario de verano, en los puntos de transición DST. Para mayor concreción, considere *US Eastern* (UTC -0500), donde EDT comienza el minuto después de 1:59 (EST) el segundo domingo de marzo y termina el minuto después de 1:59 (EDT) el primer domingo de noviembre

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
<b>start</b>	22:MM	23:MM	0:MM	1:MM	3:MM	4:MM
<b>end</b>	23:MM	0:MM	1:MM	1:MM	2:MM	3:MM

Cuando comienza el horario de verano (la línea de «inicio»), tiempo real transcurrido (*wall time*) salta de 1:59 a 3:00. Un tiempo de pared de la forma 2: MM realmente no tiene sentido ese día, por lo que `astimezone` (*Eastern*) no entregará un resultado con `hour == 2` el día en que comienza el horario de verano. Por ejemplo, en la transición de primavera de 2016, obtenemos

```

>>> 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):

```



Q

```
...
print(u.time(), t.tzinfo, 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, 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)).

## Ver también:

### [zoneinfo](#)

The [datetime](#) module has a basic [timezone](#) class (for handling arbitrary fixed offsets from UTC) and its [timezone.utc](#) attribute (a UTC timezone instance).

[zoneinfo](#) trae la *base de datos de zonas horarias de la IANA* (también conocida como la base de datos Olson) a Python y se recomienda su uso.

### [IANA timezone database](#)

La base de datos de zonas horarias (a menudo llamada *tz*, *tzdata* o *zoneinfo*) contiene código y datos que representan el historial de la hora local de muchos lugares representativos de todo el mundo. Se actualiza periódicamente para reflejar los cambios realizados por los cuerpos políticos en los límites de la zona horaria, las compensaciones UTC y las reglas de horario de verano.



Q

La clase [timezone](#) es una subclase de [tzinfo](#), cada una de las cuales representa una zona horaria definida por un desplazamiento fijo desde UTC.

Los objetos de esta clase no se pueden usar para representar la información de zona horaria en los lugares donde se usan diferentes desplazamientos en diferentes días del año o donde se han realizado cambios históricos en la hora civil.

### `class datetime.timezone(offset, name=None)`

El argumento `offset` debe especificarse como un objeto de [timedelta](#) que representa la diferencia entre la hora local y UTC. Debe estar estrictamente entre `-timedelta(hours = 24)` y `timedelta(hours = 24)`, de lo contrario [ValueError](#) se genera.

El argumento `name` es opcional. Si se especifica, debe ser una cadena de caracteres que se utilizará como el valor returnedo por el método [datetime.tzname\(\)](#).

*Added in version 3.2.*

*Distinto en la versión 3.7:* El desfase UTC no está restringido a un número entero de minutos.

#### `timezone.utcoffset(dt)`

Retorna el valor fijo especificado cuando se construye la instancia [timezone](#).

El argumento `dt` se ignora. El valor de retorno es una instancia de [timedelta](#) igual a la diferencia entre la hora local y UTC.

*Distinto en la versión 3.7:* El desfase UTC no está restringido a un número entero de minutos.

#### `timezone.tzname(dt)`

Retorna el valor fijo especificado cuando se construye la instancia [timezone](#).

Si no se proporciona `name` en el constructor, el nombre returnedo por `tzname(dt)` se genera a partir del valor del `offset` de la siguiente manera. Si `offset` es `timedelta(0)`, el nombre es «UTC», de lo contrario es una cadena en el formato `UTC±HH:MM`, donde `±` es el signo de `offset`, `HH` y `MM` son dos dígitos de `offset.hours` y `offset.minutes` respectivamente.

*Distinto en la versión 3.6:* El nombre generado a partir de `offset=timedelta(0)` ahora es simplemente 'UTC', no 'UTC+00:00'.

#### `timezone.dst(dt)`

Siempre retorna `None`.

#### `timezone.fromutc(dt)`

Retorna `dt + offset`. El argumento `dt` debe ser una instancia consciente (`aware`) [datetime](#), con `tzinfo` establecido en `self`.

Atributos de clase:



## strftime() and strptime() Behavior

[date](#), [datetime](#), y [time](#) los objetos admiten un método `strftime(format)`, para crear una cadena que represente el tiempo bajo el control de una cadena de caracteres de formato explícito.

Por el contrario, el método de clase [datetime.strptime\(\)](#) crea un objeto [datetime](#) a partir de una cadena que representa una fecha y hora y una cadena de formato correspondiente.

The table below provides a high-level comparison of [strftime\(\)](#) versus [strptime\(\)](#):

	<code>strftime</code>	<code>strptime</code>
Uso	Convierte objetos en una cadena de caracteres de acuerdo con un formato dado	<i>parsear</i> una cadena en un objeto <a href="#">datetime</a> con el formato correspondiente
Tipo de método	Método de instancia	Método de clase
Método de	<a href="#">date</a> ; <a href="#">datetime</a> ; <a href="#">time</a>	<a href="#">datetime</a>
Firma	<code>strftime(format)</code>	<code>strptime(date_string, format)</code>

## 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'
```

&gt;&gt;&gt;

La siguiente es una lista de todos los códigos de formato que requiere el estándar 1989 C, y estos funcionan en todas las plataformas con una implementación estándar C.

Di-rec-tiva	Significado	Ejemplo	No-tas
%a	Día de la semana como nombre abreviado según la configuración regional.	<i>Sun, Mon, ..., Sat (en_US); So, Mo, ..., Sa (de_DE)</i>	(1)
%A	Día de la semana como nombre completo de la localidad.	<i>Sunday, Monday, ..., Saturday (en_US);</i>	(1)

Formato	Significado	Ejemplo	tas
%w	Día de la semana como un número decimal, donde 0 es domingo y 6 es sábado.	Sonntag, Montag, ..., Samstag (de_DE)	
%d	Día del mes como un número decimal relleno con ceros.	0, 1, ..., 6	
%b	Mes como nombre abreviado según la configuración regional.	01, 02, ..., 31	(9)
%B	Mes como nombre completo según la configuración regional.	Jan, Feb, ..., Dec (en_US); Jan, Feb, ..., Dez (de_DE)	(1)
%m	Mes como un número decimal relleno con ceros.	January, February, ..., December (en_US); Januar, Februar, ..., Dezember (de_DE)	(1)
%y	Año sin siglo como un número decimal relleno con ceros.	01, 02, ..., 12	(9)
%Y	Año con siglo como número decimal.	00, 01, ..., 99	(9)
%H	Hora (reloj de 24 horas) como un número decimal relleno con ceros.	0001, 0002, ..., 2013, 2014, ..., 9998, 9999	(2)
%I	Hora (reloj de 12 horas) como un número decimal relleno con ceros.	00, 01, ..., 23	(9)
%p	El equivalente de la configuración regional de AM o PM.	01, 02, ..., 12	(9)
%M	Minuto como un número decimal relleno con ceros.	AM, PM (en_US); am, pm (de_DE)	(1), (3)
%S	Segundo como un número decimal relleno con ceros.	00, 01, ..., 59	(9)
%f	Microsegundo como número decimal, con ceros hasta 6 dígitos.	00, 01, ..., 59	(4), (9)
%z	Desplazamiento ( <i>offset</i> ) UTC en la forma <code>±HHMM[SS[.fffffff]]</code> (cadena de caracteres vacía si el objeto es naíf ( <i>naive</i> )).	000000, 000001, ..., 999999	(5)
%Z	Nombre de zona horaria (cadena de caracteres vacía si el objeto es naíf ( <i>naive</i> )).	(vacío), +0000, -0400, +1030, +063415, -030712.345216	(6)
%j	Día del año como un número decimal relleno con ceros.	(vacío), UTC, GMT	(6)
%U	Número de semana del año (domingo como primer día de la semana) como un número decimal con ceros. Todos los días de un	001, 002, ..., 366	(9)
		00, 01, ..., 53	(7), (9)



Q

Directiva	Significado	Ejemplo	Notas
	nuevo año que preceden al primer domingo se consideran en la semana 0.		
%W	Número de semana del año (lunes como primer día de la semana) como un número decimal con ceros. Todos los días de un nuevo año que preceden al primer lunes se consideran en la semana 0.	00, 01, ..., 53	(7), (9)
%c	Representación apropiada de fecha y hora de la configuración regional.	Tue Aug 16 21:30:00 1988 (en_US); Di 16 Aug 21:30:00 1988 (de_DE)	(1)
%x	Representación de fecha apropiada de la configuración regional.	08/16/88 (None); 08/16/1988 (en_US); 16.08.1988 (de_DE)	(1)
%X	Representación de la hora apropiada de la configuración regional.	21:30:00 (en_US); 21:30:00 (de_DE)	(1)
%%	Un carácter literal '%' .	%	

Se incluyen varias directivas adicionales no requeridas por el estándar C89 por conveniencia. Todos estos parámetros corresponden a valores de fecha ISO 8601.

Directiva	Significado	Ejemplo	Notas
%G	ISO 8601 año con siglo que representa el año que contiene la mayor parte de la semana ISO (%V).	0001, 0002, ..., 2013, 2014, ..., 9998, 9999	(8)
%u	ISO 8601 día de la semana como un número decimal donde 1 es lunes.	1, 2, ..., 7	
%V	ISO 8601 semana como un número decimal con lunes como primer día de la semana. La semana 01 es la semana que contiene el 4 de enero.	01, 02, ..., 53	(8), (9)
%:z	UTC offset in the form <code>±HH:MM[:SS[.fffffff]]</code> (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](#).



platform, consult the [strftime\(3\)](#) documentation. There are also differences between platforms in handling of unsupported format specifiers.

*Added in version 3.6:* %G, %u y %V fueron añadidos.

*Added in version 3.12:* %:z was added.

## Detalle técnico

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. [\[4\]](#)

Usar `datetime.strptime(date_string, format)` es equivalente a:

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

excepto cuando el formato incluye componentes de sub-segundos o información de compensación de zona horaria, que son compatibles con `datetime.strptime` pero son descartados por `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.

Por la misma razón, el manejo de cadenas de formato que contienen puntos de código Unicode que no se pueden representar en el conjunto de caracteres del entorno local actual también depende de la plataforma. En algunas plataformas, estos puntos de código se conservan intactos en la salida, mientras que en otros `strftime` puede generar [UnicodeError](#) o retornar una cadena vacía.

Notas:

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

*Distinto en la versión 3.2:* In previous versions, [strftime\(\)](#) method was restricted to years >= 1900.

*Distinto en la versión 3.3:* In version 3.2, [strftime\(\)](#) method was restricted to years >= 1000.

- When used with the [strftime\(\)](#) method, the %p directive only affects the output hour field if the %I directive is used to parse the hour.



Q

5. When used with the [strftime\(\)](#) 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.

Para un objeto consciente (*aware*)

%z

[utcoffset\(\)](#) is transformed into a string of the form `±HHMM[SS[.fffffff]]`, 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 fffffff is a 6-digit string giving the number of UTC offset microseconds. The fffffff part is omitted when the offset is a whole number of seconds and both the fffffff 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`'.

*Distinto en la versión 3.7:* El desfase UTC no está restringido a un número entero de minutos.

*Distinto en la versión 3.7:* When the %z directive is provided to the [strftime\(\)](#) 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.

[strftime\(\)](#) only accepts certain values for %Z:

1. cualquier valor en `time.tzname` para la configuración regional de su máquina
2. los valores codificados de forma rígida UTC y GMT

Entonces, alguien que viva en Japón puede tener JST, UTC y GMT como valores válidos, pero probablemente no EST. Lanzará `ValueError` para valores no válidos.

*Distinto en la versión 3.2:* When the %z directive is provided to the [strftime\(\)](#) 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 [strftime\(\)](#) 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 [strftime\(\)](#) format string. Also note that %G and %Y are not interchangeable.



## Pie de notas

- [1] Es decir, si ignoramos los efectos de la relatividad
- [2] Esto coincide con la definición del calendario «proléptico gregoriano» en el libro de *Dershowitz y Reingold Cálculos calendáricos*, donde es el calendario base para todos los cálculos. Consulte el libro sobre algoritmos para convertir entre ordinales gregorianos prolépticos y muchos otros sistemas de calendario.
- [3] Consulte [guide to the mathematics of the ISO 8601 calendar](#) de R. H. van Gent para obtener una buena explicación.
- [4] Passing `datetime.strptime('Feb 29', '%b %d')` will fail since 1900 is not a leap year.