



datetime — Tipos básicos de fecha y hora

Código fuente: [Lib/datetime.py](https://lib/datetime.py)

El módulo `datetime` proporciona clases para manipular fechas y horas.

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.

Ver también:

Módulo `calendar`

Funciones generales relacionadas a *calendar*.

Módulo `time`

Acceso a tiempo y conversiones.

Module `zoneinfo`

Concrete time zones representing the IANA time zone database.

Paquete `dateutil`

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

Objetos conscientes (*aware*) y naïfs (*naive*)

Los objetos de fecha y hora pueden clasificarse como conscientes (*aware*) o naïfs (*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.

Para las aplicaciones que requieren objetos conscientes, los objetos `datetime` y `time` tienen un atributo de información de zona horaria opcional, `tzinfo`, que se puede establecer en una instancia de una subclase de la clase 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.

El módulo `datetime` solo proporciona una clase concreta `tzinfo`, la clase `timezone`. La clase `timezone` puede representar zonas horarias simples con desplazamientos fijos desde UTC, como UTC o las zonas horarias EST y EDT de América del Norte. La compatibilidad de zonas horarias con niveles de detalle más profundos depende de la aplicación. Las reglas para el ajuste del tiempo en todo el mundo son mas políticas que racionales, cambian con frecuencia y no existe un estándar adecuado para cada aplicación, aparte de UTC.

Constantes



3.10.7



Ir

`datetime.MINYEAR`

El número de año más pequeño permitido en un objeto `date` o `datetime`. `MINYEAR` es `1`.

`datetime.MAXYEAR`

El número de año más grande permitido en un objeto `date` o en `datetime`. `MAXYEAR` es `9999`.

Tipos disponibles

`class datetime.date`

Una fecha naïf (*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`

Una duración que expresa la diferencia entre dos instancias `date`, `time` o `datetime` a una resolución de microsegundos.

`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 cuenta la zona horaria y / o el horario de verano).

`class datetime.timezone`

Una clase que implementa la clase de base abstracta `tzinfo` como un desplazamiento fijo desde el UTC.

Nuevo en la versión 3.2.

Los objetos de este tipo son inmutables.

Relaciones de subclase:

```
graph TD
    object --> timedelta
    object --> tzinfo
    tzinfo --> timezone
    object --> time
    object --> date
    object --> datetime
```

Propiedades comunes

Las clases `date`, `datetime`, `time`, y `timezone` comparten estas características comunes:

- Los objetos de este tipo son inmutables.



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

Los objetos del tipo `date` son siempre naïf (*naive*).

Un objeto de tipo `time` o `datetime` puede ser consciente (*aware*) o naïf (*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 naïf (*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 naïf (*naive*).

La distinción entre los objetos consciente (*aware*) y naïf (*naive*) no se aplica a `timedelta`.

Objetos `timedelta`

El objeto `timedelta` representa una duración, la diferencia entre dos fechas u horas.

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

Todos los argumentos son opcionales y predeterminados a 0. Los argumentos pueden ser enteros o flotantes, y pueden ser positivos o negativos.

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

- $0 \leq \text{microsegundos} < 1000000$
- $0 \leq \text{segundos} < 3600 \cdot 24$ (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,
```

```
>>>
```

 3.10.7

Ir

```
... ,
>>> # 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)
```

Atributos de clase:

`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)`.

Tenga en cuenta que, debido a la normalización, `timedelta.max > -timedelta.min`. `-timedelta.max` no es representable como un objeto `timedelta`.

Atributos de instancia (solo lectura):

Atributo	Valor
days	Entre -999999999 y 999999999 inclusive
seconds	Entre 0 y 86399 inclusive
microseconds	Entre 0 y 999999 inclusive

Operaciones soportadas:

Operación	Resultado
<code>t1 = t2 + t3</code>	Suma de <code>t2</code> y <code>t3</code> . Después <code>t1-t2 == t3</code> y <code>t1-t3 == t2</code> son verdaderos. (1)



3.10.7



lr

<code>t1 = t2 - t3</code>	La suma de <code>t2</code> y <code>t3</code> . Después <code>t1 == t2 - t3</code> y <code>t2 == t1 + t3</code> son verdaderos. (1)(6)
<code>t1 = t2 * i</code> o <code>t1 = i * t2</code>	Delta multiplicado por un entero. Después <code>t1 // i == t2</code> es verdadero, siempre que <code>i != 0</code> .
	En general, <code>t1 * i == t1 * (i-1) + t1</code> es verdadero. (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>	División (3) de la duración total <code>t2</code> por unidad de intervalo <code>t3</code> . Retorna un objeto <code>float</code> .
<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 <code>timedelta</code> . (3)
<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 <code>timedelta</code> .
<code>+t1</code>	Retorna un objeto <code>timedelta</code> con el mismo valor. (2)
<code>-t1</code>	equivalente a <code>timedelta(-t1.days, -t1.seconds, -t1.microseconds)</code> , y a <code>t1*-1</code> . (1)(4)
<code>abs(t)</code>	equivalente a <code>+t</code> cuando <code>t.days >= 0</code> , y a <code>-t</code> cuando <code>t.days < 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 <code>timedelta</code> 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. División por 0 genera `ZeroDivisionError`.
4. `-timedelta.max` no es representable como un objeto `timedelta`.
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)
```

>>>



3.10.7



Ir

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 de piso y la división verdadera de un objeto `timedelta` por otro `timedelta` ahora son compatibles, al igual que las operaciones restantes y la función `divmod()`. La división verdadera y multiplicación de un objeto `timedelta` por un objeto flotante ahora son compatibles.

Comparaciones de los objetos `timedelta` son compatibles, con algunas limitaciones.

Las comparaciones `==` o `!=` *Siempre* retornan `bool`, sin importar el tipo de objeto comparado:

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

>>>

Para todas las demás comparaciones (como `<` y `>`), cuando un objeto `timedelta` se compara con un objeto de un tipo diferente, se genera `TypeError`:

```
>>> 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'
```

>>>

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

Métodos de instancia:

`timedelta.total_seconds()`

Retorna el número total de segundos contenidos en la duración. Equivalente a `td / 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.

Nuevo en la versión 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)
```

>>>



3.10.7



Ir

```

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

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.



3.10.7



Ir

ordinal 1.

`ValueError` se genera a menos que `1 <= ordinal <= date.max.toordinal().()`. Para cualquier fecha `d`, `date.fromordinal(d.toordinal()) == d`.

classmethod `date.fromisoformat(date_string)`

Retorna `date` correspondiente a una `date_string` dada en el formato YYYY-MM-DD:

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

>>>

Este es el inverso de `date.isoformat()`. Solo admite el formato AAAA-MM-DD.

Nuevo en la versión 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()`.

Nuevo en la versión 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>	Calcula <code>date2</code> tal que <code>date2 + timedelta == date1</code> . (2)
<code>timedelta = date1 - date2</code>	(3)



3.10.7



Ir

`date1 < date2`*date1* se considera menor que *date2* cuando *date1* precede a *date2* en el tiempo. (4)**Notas:**

1. *date2* se mueve hacia adelante en el tiempo si `timedelta.days > 0`, o hacia atrás si `timedelta.days < 0`. Después `date2 - date1 == timedelta.days.timedelta.seconds` y `timedelta.microseconds` se ignoran. `OverflowError` se lanza si `date2.year` sería menor que `MINYEAR` o mayor que `MAXYEAR`.
2. `timedelta.seconds` y `timedelta.microseconds` son ignorados.
3. Esto es exacto y no puede desbordarse. `timedelta.seconds` y `timedelta.microseconds` son 0, y `date2 + timedelta == date1`.
4. En otras palabras, `date1 < date2` si y solo si `date1.toordinal()` < `date2.toordinal()`. La comparación de fechas plantea `TypeError` si el otro elemento comparado no es también un objeto `date`. Sin embargo, se retorna `NotImplemented` si el otro elemento comparado tiene un atributo `timetuple()`. Este enlace ofrece a otros tipos de objetos de fecha la posibilidad de implementar una comparación de tipos mixtos. Si no, cuando un objeto `date` se compara con un objeto de un tipo diferente, `TypeError` se genera a menos que la comparación sea `==` or `!=`. Los últimos casos retorna `False` o `True`, respectivamente.

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))
```

donde `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` es el número de día dentro del año actual que comienza con 1 para el 1 de enero.

`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()`



3.10.7



Ir

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\(\)](#).

date.isocalendar()

Retorna un objeto [named tuple](#) con tres componentes: year, week y weekday.

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'
```

>>>

Este es el inverso de [date.fromisoformat\(\)](#).

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)



3.10.7



Ir

una lista completa de las directivas de formato, consulte [Comportamiento strftime\(\) y strptime\(\)](#).

`date.__format__(format)`

Igual que `date.strftime()`. Esto hace posible especificar una cadena de formato para un objeto `date` en [literales de cadena con formato](#) y cuando se usa `str.format()`. Para obtener una lista completa de las directivas de formato, consulte [Comportamiento strftime\(\) y strptime\(\)](#).

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'
>>> 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
```



3.10.7



Ir

```
... 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.

Nuevo en la versión 3.6: Se agregó el argumento `fold`.

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 funciona como `now()`, pero sin un parámetro `tz`.

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

Retorna la fecha y hora local actual.

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()`).



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 naïf (*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 naïfs (*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)`.

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 retornado `datetime` es naïf (*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.

classmethod `datetime.utcfromtimestamp(timestamp)`

Retorna el UTC `datetime` correspondiente a la marca de tiempo POSIX, con `tzinfo` `None`. (El objeto resultante 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)
```



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()`.

classmethod `datetime.fromordinal(ordinal)`

Se genera `datetime` correspondiente al ordinal del proléptico 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=self.tzinfo)`

Se genera un nuevo objeto `datetime` cuyos componentes de fecha son iguales a los dados en el objeto `date`, y cuyos componentes de tiempo son iguales a los dados en el objeto `time`. Si se proporciona el argumento `tzinfo`, su valor se usa para establecer el atributo `tzinfo` del resultado; de lo contrario, se usa el atributo `tzinfo` del argumento `time`.

Para cualquier objeto de `datetime` `d`, `d == datetime.combine(d.date(), d.time(), d.tzinfo)`. Si la fecha es un objeto de `datetime`, se ignoran sus componentes de tiempo y `tzinfo`.

Distinto en la versión 3.6: Se agregó el argumento `tzinfo`.

classmethod `datetime.fromisoformat(date_string)`

Retorna `datetime` correspondiente a `date_string` en uno de los formatos emitidos por `date.isoformat()` y `datetime.isoformat()`.

Específicamente, esta función admite cadenas de caracteres en el formato:

```
YYYY-MM-DD[*HH[:MM[:SS[.fff[fff]]]]][+HH:MM[:SS[.ffffff]]]
```

donde `*` puede coincidir con cualquier carácter individual.

Prudencia: Esto *no* admite el *parsing* de cadenas de caracteres arbitrarias ISO 8601; solo está pensado cómo la operación inversa de `datetime.isoformat()`. Un *parseador* ISO 8601 mas completo, `dateutil.parser.isoparse` está disponible en el paquete de terceros `dateutil`.

Ejemplos:

```
>>> from datetime import datetime
>>> datetime.fromisoformat('2011-11-04')
datetime.datetime(2011, 11, 4, 0, 0)
>>> datetime.fromisoformat('2011-11-04T00:05:23')
datetime.datetime(2011, 11, 4, 0, 5, 23)
>>> datetime.fromisoformat('2011-11-04 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')
```

>>>



3.10.7



Ir

Nuevo en la versión 3.7.

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()`.

Nuevo en la versión 3.8.

classmethod `datetime.strptime(date_string, format)`

Retorna `datetime` correspondiente a `date_string`, analizado según `format`.

Esto es equivalente a:

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

Se genera `ValueError` se genera si `date_string` y el formato no pueden ser analizados por `time.strptime()` o si retorna un valor que no es una tupla de tiempo. Para obtener una lista completa de las directivas de formato, consulte [Comportamiento strptime\(\) y strptime\(\)](#).

Atributos de clase:

`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`



3.10.7



lr

datetime.tzinfo

El objeto pasó como argumento *tzinfo* al constructor `datetime`, o `None` si no se pasó ninguno.

datetime.fold

En `[0, 1]`. Se usa para desambiguar los tiempos de pared durante un intervalo repetido. (Se produce un intervalo repetido cuando los relojes se retrotraen al final del horario de verano o cuando el desplazamiento UTC para la zona actual se reduce por razones políticas). El valor 0 (1) representa el anterior (posterior) de los dos momentos con la misma representación de tiempo real transcurrido (*wall time*).

Nuevo en la versión 3.6.

Operaciones soportadas:

Operación	Resultado
<code>datetime2 = datetime1 + timedelta</code>	(1)
<code>datetime2 = datetime1 - timedelta</code>	(2)
<code>timedelta = datetime1 - datetime2</code>	(3)
<code>datetime1 < datetime2</code>	Compara <code>datetime</code> to <code>datetime</code> . (4)

1. *datetime2* es una duración de *timedelta* eliminada de *datetime1*, avanzando en el tiempo si `timedelta.days > 0`, o hacia atrás si `timedelta.days < 0`. El resultado tiene el mismo *tzinfo* como el atributo *datetime* y `datetime2 - datetime1 == timedelta`. `OverflowError` se genera si *datetime2.year* sería menor que `MINYEAR` o mayor que `MAXYEAR`. Tenga en cuenta que no se realizan ajustes de zona horaria, incluso si la entrada es un objeto consciente.
2. Calcula el *datetime* tal que `datetime2 + timedelta == datetime1`. En cuanto a la adición, el resultado tiene el mismo atributo *tzinfo* que la fecha y hora de entrada, y no se realizan ajustes de zona horaria, incluso si la entrada es consciente.
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.`

Si ambos son conscientes (*aware*) y tienen atributos diferentes *tzinfo*, `a-b` actúa como si primero a y b se convirtieran primero en fechas naíf (naive) UTC. El resultado es (a.replace(tzinfo = None) - a.utcoffset()) - (b.replace (tzinfo = None) - b.utcoffset()) excepto que la implementación nunca se desborda.`

4. *datetime1* se considera menor que *datetime2* cuando *datetime1* precede *datetime2* en el tiempo.

Si un de los elementos comparados es naíf (*naive*) y el otro lo sabe, se genera un `TypeError` si se intenta una comparación de órdenes. Para las comparaciones de igualdad, las instancias naíf nunca son iguales a las instancias conscientes (*aware*).

Si ambos comparados son conscientes (*aware*) y tienen el mismo atributo *tzinfo*, el atributo común *tzinfo* se ignora y se comparan las fechas base. Si ambos elementos comparados son conscientes y



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

Nota: Para evitar que la comparación vuelva al esquema predeterminado de comparación de direcciones de objetos, la comparación de fecha y hora normalmente genera `TypeError` si el otro elemento comparado no es también un objeto `datetime`. Sin embargo, `NotImplemented` se retorna si el otro elemento comparado tiene un atributo `timetuple()`. Este enlace ofrece a otros tipos de objetos de fecha la posibilidad de implementar una comparación de tipos mixtos. Si no, cuándo un objeto `datetime` se compara con un objeto de un tipo diferente, se genera `TypeError` a menos que la comparación sea `==` o `!=`. Los últimos casos retornan `False` o `True`, respectivamente.

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` retornado.

`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` retornado.

`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.

Nuevo en la versión 3.6: Se agregó el argumento `fold`.

`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.

Si se llama sin argumentos (o con `tz=None`), se asume la zona horaria local del sistema para la zona horaria objetivo. El atributo `.tzinfo` de la instancia de fecha y hora convertida se establecerá en una instancia de `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



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 sub-clase `tzinfo` para afectar el resultado retornado 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.utcoffset()`

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()`.

`d.timetuple()` es equivalente a:

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

donde `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` es el número de día dentro del año actual que comienza con 1 para el 1 de enero. El indicador `tm_isdst` del resultado se establece de acuerdo con el método `dst()`: `tzinfo` es `None` o `dst()` retorna `None`, `tm_isdst` se establece en `-1`; si no `dst()` retorna un valor distinto de cero, `tm_isdst` se establece en `1`; de lo contrario `tm_isdst` se establece en `0`.



3.10.7



lr

fuerza a 0 independientemente de lo que retorna `d.dst()`. El horario de verano nunca está en vigencia durante un horario UTC.

Si *d* es consciente (*aware*), *d* se normaliza a la hora UTC, restando `d.utcoffset()`, y se retorna `time.struct_time` para la hora normalizada. `tm_isdst` se fuerza a 0. Tenga en cuenta que un `OverflowError` puede aparecer si `d*.year` fue `MINYEAR` o `MAXYEAR` y el ajuste UTC se derrama durante el límite de un año.

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 (*aware*) para representar las horas en UTC; como resultado, el uso de `utcfromtimetuple` puede dar resultados engañosos. Si tiene una `datetime` naïf que representa UTC, use `datetime.replace(tzinfo=timezone.utc)` para que sea consciente, en cuyo punto se puede usar `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()`.

Se supone que las instancias Naïf `datetime` representan la hora local y este método se basa en la función de plataforma C `mktime()` para realizar la conversión. Dado que `datetime` admite un rango de valores más amplio que `mktime()` en muchas plataformas, este método puede lanzar `OverflowError` para tiempos lejanos en el pasado o en el futuro.

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

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

Nuevo en la versión 3.3.

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

Nota: No hay ningún método para obtener la marca de tiempo (*timestamp*) POSIX directamente de una instancia naïf `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()`



3.10.7



lr

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

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):
...         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'
```

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.ffffff``.

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')
```



3.10.7



Ir

```
... d.isoformat(timespec='microseconds')
'2015-01-01T12:30:59.000000'
```

Nuevo en la versión 3.6: Se agregó el argumento *timespec*.

`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 naïf (*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.

`datetime.strftime(format)`

Retorna una cadena de caracteres que representa la fecha y la hora, controlada por una cadena de formato explícito. Para obtener una lista completa de las directivas de formato, consulte [Comportamiento strftime\(\) y strptime\(\)](#).

`datetime.__format__(format)`

Igual que `date.strftime()`. Esto hace posible especificar una cadena de formato para un objeto `date` en [literales de cadena con formato](#) y cuando se usa `str.format()`. Para obtener una lista completa de las directivas de formato, consulte [Comportamiento strftime\(\) y strptime\(\)](#).

Ejemplos de uso: `datetime`

Ejemplos de trabajo con objetos `datetime`:

```
>>> 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
```



3.10.7



lr

```

>>> # Using datetime.datetime() 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", "month", "time")
'The day is 21, the month is November, the time is 04:30PM.'

```

El siguiente ejemplo define una subclase `KabulTz` 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

```



3.10.7



Ir

```

    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
>>> 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

```

Objetos `time`

Un objeto `time` representa a una hora del día (local), independiente de cualquier día en particular, y está sujeto a ajustes a través de un objeto `tzinfo`.

```
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]`.

Si se proporciona un argumento fuera de esos rangos, se genera `ValueError`. Todo predeterminado a `0` excepto `tzinfo`, que por defecto es `None`.

Atributos de clase:



`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)`.

`time.minute`

En `range(60)`.

`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`

En `[0, 1]`. Se usa para desambiguar los tiempos de pared durante un intervalo repetido. (Se produce un intervalo repetido cuando los relojes se retrotraen al final del horario de verano o cuando el desplazamiento UTC para la zona actual se reduce por razones políticas). El valor 0 (1) representa el anterior (posterior) de los dos momentos con la misma representación de tiempo real transcurrido (*wall time*).

Nuevo en la versión 3.6.

Los objetos `time` admiten la comparación de `time` con `time`, donde *a* se considera menor que *b* cuando *a* precede a *b* en el tiempo. Si un elemento comparado es naíf (*naive*) y el otro lo sabe se genera `TypeError` si se intenta una comparación de orden. Para las comparaciones de igualdad, las instancias naíf nunca son iguales a las instancias conscientes (*aware*).

Si ambos elementos comparados son conscientes (*aware*) y tienen el mismo atributo `tzinfo`, el atributo común `tzinfo` se ignora y se comparan los tiempos base. Si ambos elementos comparados son conscientes y tienen atributos diferentes `tzinfo`, los elementos comparados se ajustan primero restando sus compensaciones UTC (obtenidas de `self.utcoffset()`). Para evitar que las comparaciones de tipos mixtos vuelvan a la comparación predeterminada por dirección de objeto, cuando un objeto `time` se compara con un objeto de un tipo diferente, se genera `TypeError` a menos que la comparación es `==` o `!=`. Los últimos casos retornan `False` o `True`, respectivamente.

Distinto en la versión 3.3: Las comparaciones de igualdad entre las instancias conscientes (*aware*) y naífs (*naive*) `time` no generan `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



3.10.7



Ir

Otro constructor:

classmethod `time.fromisoformat(time_string)`

Retorna una `time` correspondiente a `time_string` en uno de los formatos emitidos por `time.isoformat()`. Específicamente, esta función admite cadenas de caracteres en el formato:

```
HH[:MM[:SS[.fff[fff]]]][+HH:MM[:SS[.ffffff]]]
```

Prudencia: Esto *no* admite el *parsing* de cadenas arbitrarias ISO 8601. Solo pretende ser la operación inversa de `time.isoformat()`.

Ejemplos:

```
>>> from datetime import time
>>> time.fromisoformat('04:23:01')
datetime.time(4, 23, 1)
>>> 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)))
```

Nuevo en la versión 3.7.

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` naïf (*naive*) desde un consciente (*aware*) `time`, sin conversión de los datos de tiempo.

Nuevo en la versión 3.6: Se agregó el argumento `fold`.

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

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

- HH:MM:SS.ffffff, si `microsecond` no es 0
- HH:MM:SS, si `microsecond` es 0
- HH:MM:SS.ffffff+HH:MM[:SS[.ffffff]], si `utcoffset()` no retorna None
- HH:MM:SS+HH:MM[:SS[.ffffff]], 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.
- '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.ffffff``.



3.10.7



Ir

`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'
```

Nuevo en la versión 3.6: Se agregó el argumento *timespec*.

`time.__str__()`

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

`time.strftime(format)`

Retorna una cadena que representa la hora, controlada por una cadena de formato explícito. Para obtener una lista completa de las directivas de formato, consulte [Comportamiento strftime\(\) y strptime\(\)](#).

`time.__format__(format)`

Igual que `time.strftime()`. Esto permite especificar una cadena de formato para un objeto `time` en [cadenas de caracteres literales con formato](#) y cuando se usa `str.format()`. Para obtener una lista completa de las directivas de formato, consulte [Comportamiento strftime\(\) y strptime\(\)](#).

`time.utcoffset()`

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()`

Si `tzinfo` es `None`, retorna `None`, sino retorna `self.tzinfo.tzname(None)`, o genera una excepción si este último no retorna `None` o un objeto de cadena.

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)
```



3.10.7



lr

```

...     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 {:%H:%M}.'.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. Define 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 pasó a ellos.

Debe derivar una subclase concreta y (al menos) proporcionar implementaciones de los métodos estándar `tzinfo` que necesitan los métodos `datetime` que utiliza. El módulo `datetime` proporciona `timezone`, una subclase concreta simple de `tzinfo` que puede representar zonas horarias con desplazamiento fijo desde UTC como UTC o Norte América EST y EDT.

Requisito especial para el *pickling*: La subclase `tzinfo` debe tener un método `__init__()` que se pueda invocar sin argumentos; de lo contrario, se puede hacer *pickling* pero posiblemente no se vuelva a despegar. Este es un requisito técnico que puede ser relajado en el futuro.

Una subclase concreta de `tzinfo` puede necesitar implementar los siguientes métodos. Exactamente qué métodos son necesarios depende de los usos de los objetos conscientes(*aware*) `datetime`. En caso de duda, simplemente implemente todos ellos.

`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
```



3.10.7



lr

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.

Retorna `timedelta(0)` si el horario de verano no está en vigor. Si DST está en vigor, retorna el desplazamiento como un objeto `timedelta` (consulte `utcoffset()` para más detalles). Tenga en cuenta que el desplazamiento DST, si corresponde, ya se ha agregado al desplazamiento UTC retornado por `utcoffset()`, por lo que no es necesario consultar `dst()` a menos que esté interesado en obtener información DST por separado. Por ejemplo, `datetime.timetuple()` llama a su `tzinfo` del atributo `dst()` para determinar cómo se debe establecer el indicador `tm_isdst`, y `tzinfo.fromutc()` llama `dst()` para tener en cuenta los cambios de horario de verano al cruzar zonas horarias.

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

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

debe retornar el mismo resultado para cada `datetime dt` con `dt.tzinfo == tz`. Para las subclases sanas `tzinfo`, esta expresión produce el «desplazamiento estándar» de la zona horaria, que no debe depender de la fecha o la hora, sino solo de la ubicación geográfica. La implementación de `datetime.astimezone()` se basa en esto, pero no puede detectar violaciones; es responsabilidad del programador asegurarlo. Si una subclase `tzinfo` no puede garantizar esto, puede anular la implementación predeterminada de `tzinfo.fromutc()` para que funcione correctamente con `astimezone()` independientemente.

La mayoría de las implementaciones de `dst()` probablemente se parecerán a una de estas dos:

```
def dst(self, dt):
    # a fixed-offset class: doesn't account for DST
    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)`

Retorna el nombre de zona horaria correspondiente al objeto `datetime dt`, como una cadena de caracteres. El módulo `datetime` no define nada sobre los nombres de cadena, y no hay ningún requisito de que signifique algo en particular. Por ejemplo, «GMT», «UTC», «-500», «-5:00», «EDT», «US/Eastern»,



3.10.7



Ir

nas subclases `tzinfo` desearán retornar diferentes nombres dependiendo del valor específico de `dt` pasado, especialmente si la clase `tzinfo` es contable para el horario de verano.

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)`

Esto se llama desde la implementación predeterminada `datetime.astimezone()`. Cuando se llama desde eso, `dt.tzinfo` es `self`, y los datos de fecha y hora de `dt` deben considerarse como una hora UTC. El propósito de `fromutc()` es ajustar los datos de fecha y hora, retornando una fecha y hora equivalente en la hora local de `self`.

La mayoría de las subclases `tzinfo` deberían poder heredar la implementación predeterminada `fromutc()` sin problemas. Es lo suficientemente fuerte como para manejar zonas horarias de desplazamiento fijo y zonas horarias que representan tanto el horario estándar como el horario de verano, y esto último incluso si los tiempos de transición DST difieren en años diferentes. Un ejemplo de una zona horaria predeterminada `fromutc()`, la implementación puede que no se maneje correctamente en todos los casos es aquella en la que el desplazamiento estándar (desde UTC) depende de la fecha y hora específicas que pasan, lo que puede suceder por razones políticas. Las implementaciones predeterminadas de `astimezone()` y `fromutc()` pueden no producir el resultado que desea si el resultado es una de las horas a horcajadas en el momento en que cambia el desplazamiento estándar.

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
```



3.10.7



lr

```

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):

    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()

```



3.10.7



lr

```

# US DST Rules
#
# This is a simplified (i.e., wrong for a few cases) set of rules for US
# DST start and end times. For a complete and up-to-date set of DST rules
# and timezone definitions, visit the Olson Database (or try pytz):
# http://www.twinsun.com/tz/tz-link.htm
# http://sourceforge.net/projects/pytz/ (might not be up-to-date)
#
# In the US, since 2007, DST starts at 2am (standard time) on the second
# Sunday in March, which is the first Sunday on or after Mar 8.
DSTSTART_2007 = datetime(1, 3, 8, 2)
# and ends at 2am (DST time) 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:
            return self.stdname

```




3.10.7



lr

```

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")

```

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
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



3.10.7



Ir

(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):
...     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
```

Cuando finaliza el horario de verano (la línea «final»), hay un problema potencialmente peor: hay una hora que no se puede deletrear sin ambigüedades en el tiempo de la pared local: la última hora del día. En el Este, esos son los tiempos de la forma 5:MM UTC en el día en que termina el horario de verano. El reloj de pared local salta de 1:59 (hora del día) a la 1:00 (hora estándar) nuevamente. Horas locales de la forma 1:MM son ambiguas. `astimezone()` imita el comportamiento del reloj local al mapear dos horas UTC adyacentes en la misma hora local. En el ejemplo oriental, los tiempos UTC de la forma 5: MM y 6: MM se correlacionan con 1:MM cuando se convierten en oriental, pero los tiempos anteriores tienen el atributo `fold` establecido en 0 y los tiempos posteriores configúrelo en 1. Por ejemplo, en la transición alternativa de 2016, obtenemos:

```
>>> 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
```

Tenga en cuenta que las instancias `datetime` que difieren solo por el valor del atributo `fold` se consideran iguales en las comparaciones.

Las aplicaciones que no pueden soportar ambigüedades de tiempo real (*wall time*) deben verificar explícitamente el valor del atributo `fold` o evitar el uso de las subclases híbridas `tzinfo`; no existen ambigüedades cuando se utiliza `timezone`, o cualquier otra subclase de clase offset `tzinfo` (como una clase que representa solo *EST* (desplazamiento fijo -5 horas), o solo *EDT* (desplazamiento fijo -4 horas)).

Ver también:

`zoneinfo`

El módulo `datetime` tiene una clase básica `timezone` (para manejar compensaciones fijas arbitrarias desde UTC) y su atributo `timezone.utc` (una instancia de zona horaria UTC).

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

[IANA timezone database](#)



3.10.7



lr

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.

Objetos `timezone`

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(horas = 24)` y `timedelta(horas = 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 retornado por el método `datetime.tzname()`.

Nuevo en la versión 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 retornado 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 ±`, 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 simple ``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:

`timezone.utc`

Comportamiento `strftime()` y `strptime()`

`date`, `datetime`, y `time` los objetos admiten un método `strftime(format)`, para crear una cadena que presente 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.

La siguiente tabla proporciona una comparación de alto nivel de `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 <code>datetime</code> con el formato correspondiente
Tipo de método	Método de instancia	Método de clase
Método de	<code>date</code> ; <code>datetime</code> ; <code>time</code>	<code>datetime</code>
Firma	<code>strftime(format)</code>	<code>strptime(date_string, format)</code>

Códigos de formato `strftime()` y `strptime()`

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.

Directiva	Significado	Ejemplo	Notas
<code>%a</code>	Día de la semana como nombre abreviado según la configuración regional.	<i>Sun, Mon, ..., Sat (en_US); So, Mo, ..., Sa (de_DE)</i>	(1)
<code>%A</code>	Día de la semana como nombre completo de la localidad.	<i>Sunday, Monday, ..., Saturday (en_US); Sonntag, Montag, ..., Samstag (de_DE)</i>	(1)
<code>%w</code>	Día de la semana como un número decimal, donde 0 es domingo y 6 es sábado.	0, 1, ..., 6	
<code>%d</code>	Día del mes como un número decimal rellenado con ceros.	01, 02, ..., 31	(9)
<code>%b</code>	Mes como nombre abreviado según la configuración regional.	<i>Jan, Feb, ..., Dec (en_US); Jan, Feb, ..., Dez (de_DE)</i>	(1)
<code>%B</code>	Mes como nombre completo según la configuración regional.	<i>January, February, ..., December (en_US); Januar, Februar, ..., Dezember (de_DE)</i>	(1)
<code>%m</code>	Mes como un número decimal rellenado con ceros.	01, 02, ..., 12	(9)
<code>%y</code>	Año sin siglo como un número decimal rellenado con ceros.	00, 01, ..., 99	(9)




3.10.7



Ir

%Y	Año con siglo como número decimal.	0001, 0002, ..., 2013, 2014, ..., 9998, 9999	(2)
%H	Hora (reloj de 24 horas) como un número decimal rellenado con ceros.	00, 01, ..., 23	(9)
%I	Hora (reloj de 12 horas) como un número decimal rellenado con ceros.	01, 02, ..., 12	(9)
%p	El equivalente de la configuración regional de AM o PM.	AM, PM (en_US); am, pm (de_DE)	(1), (3)
%M	Minuto como un número decimal rellenado con ceros.	00, 01, ..., 59	(9)
%S	Segundo como un número decimal rellenado con ceros.	00, 01, ..., 59	(4), (9)
%f	Microsecond as a decimal number, zero-padded to 6 digits.	000000, 000001, ..., 999999	(5)
%z	Desplazamiento (<i>offset</i>) UTC en la forma ±HHMM[SS[.ffffff]] (cadena de caracteres vacía si el objeto es naif (<i>naive</i>)).	(vacío), +0000, -0400, +1030, +063415, -030712.345216	(6)
%Z	Nombre de zona horaria (cadena de caracteres vacía si el objeto es naif (<i>naive</i>)).	(vacío), UTC, GMT	(6)
%j	Día del año como un número decimal rellenado con ceros.	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	Representación apropiada de fecha y hora de la configuración regional.	<i>Tue Aug 16 21:30:00 1988</i> (en_US); <i>Di 16 Aug 21:30:00 1988</i> (de_DE)	(1)
%x	Representación de fecha apropiada de la configuración regional.	08/16/88 (<i>None</i>); 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.

	3.10.7	<input type="text" value="Q"/>	Ir
%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)

Es posible que no estén disponibles en todas las plataformas cuando se usan con el método `strftime()`. Las directivas ISO 8601 año e ISO 8601 semana no son intercambiables con las directivas de número de año y semana anteriores. Llamar a `strptime()` con directivas ISO 8601 incompletas o ambiguas lanzará un [ValueError](#).

El conjunto completo de códigos de formato admitidos varía según las plataformas, porque Python llama a la función `strftime()` de la biblioteca de la plataforma C, y las variaciones de la plataforma son comunes. Para ver el conjunto completo de códigos de formato admitidos en su plataforma, consulte la documentación [strftime\(3\)](#). También existen diferencias entre plataformas en el manejo de especificadores de formato no admitidos.

Nuevo en la versión 3.6: %G, %u y %V fueron añadidos.

Detalle técnico

En términos generales, `d.strftime (fmt)` actúa como el módulo `time` `time.strftime(fmt, d.timetuple())` aunque no todos los objetos admiten el método `timetuple()`.

Para el método de clase `datetime.strptime()`, el valor predeterminado es `1900-01-01T00:00:00.000`: cualquier componente no especificado en la cadena de formato se extraerá del valor predeterminado. [\[4\]](#)

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`.

Para objetos de `time`, los códigos de formato para año, mes y día no deben usarse, ya que los objetos de `time` no tienen tales valores. Si se usan de todos modos, 1900 se sustituye por el año y 1 por el mes y el día.

Para los objetos `date`, los códigos de formato para horas, minutos, segundos y microsegundos no deben usarse, ya que los objetos `date` no tienen tales valores. Si se usan de todos modos, 0 se sustituye por ellos.

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:



3.10.7



Ir

«mes/día/año» versus «día/mes/año»), y la salida puede contener caracteres Unicode codificados utilizando la codificación predeterminada de la configuración regional (por ejemplo, si la configuración regional actual es `ja_JP`, la codificación predeterminada podría ser cualquiera de `eucJP`, ```SJIS``` o `utf-8`; use `locale.getlocale()` para determinar la codificación de la configuración regional actual).

2. El método `strptime()` puede analizar años en el rango completo `[1, 9999]`, pero los años `< 1000` deben llenarse desde cero hasta un ancho de 4 dígitos.

Distinto en la versión 3.2: En versiones anteriores, el método `strptime()` estaba restringido a años `>= 1900`.

Distinto en la versión 3.3: En la versión 3.2, el método `strptime()` estaba restringido a años `>= 1000`.

3. Cuando se usa con el método `strptime()`, la directiva `%p` solo afecta el campo de hora de salida si se usa la directiva `%I` para analizar la hora.
4. A diferencia del módulo `time`, el módulo `datetime` no admite segundos intercalares.
5. Cuando se usa con el método `strptime()`, la `%f` directiva acepta de uno a seis dígitos y cero *pads* a la derecha. `%f` es una extensión del conjunto de caracteres de formato en el estándar C (pero implementado por separado en objetos de fecha y hora y, por lo tanto, siempre disponible).
6. Para un objeto naíf (*naive*), los códigos de formato `%z` y `%Z` se reemplazan por cadenas vacías.

Para un objeto consciente (*aware*)

`%z`

`utcoffset()` se transforma en una cadena de la forma `±HHMM[SS[.ffffff]]`, donde ```HH``` es una cadena de 2 dígitos que da el número de horas de desplazamiento UTC, ```MM``` es una cadena de 2 dígitos que da el número de minutos de desplazamiento UTC, ```SS``` es una cadena de 2 dígitos que da el número de segundos de desplazamiento UTC y `ffffff` es una cadena de 6 dígitos que da el número de microsegundos de desplazamiento UTC. La parte `ffffff` se omite cuando el desplazamiento es un número entero de segundos y tanto la parte `ffffff` como la parte `SS` se omiten cuando el desplazamiento es un número entero de minutos. Por ejemplo, si `utcoffset()` retorna `timedelta(hours=-3, minutes=-30)`, `%z` se reemplaza con la cadena `'-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: Cuando la directiva `%z` se proporciona al método `strptime()`, las compensaciones UTC pueden tener dos puntos como separador entre horas, minutos y segundos. Por ejemplo, `'+01:00:00'` se analizará como una compensación de una hora. Además, proporcionar `'Z'` es idéntico a `'+00:00'`.

`%Z`

En `strptime()`, `%Z` se reemplaza por una cadena de caracteres vacía si `tzname()` retorna `None`; de lo contrario, `%Z` se reemplaza por el valor retornado, que debe ser una cadena de caracteres.

`strptime()` solo acepta ciertos valores para `%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.



3.10.7



Ir

7. Cuando se usa con el método `strptime()`, `%U` y `%W` solo se usan en los cálculos cuando se especifican el día de la semana y el año calendario (`%Y`).
8. Similar a `%U` y `%W`, `%V` solo se usa en cálculos cuando el día de la semana y el año ISO (`%G`) se especifican en `strptime()` cadena de formato. También tenga en cuenta que `%G` y `%Y` no son intercambiables.
9. Cuando se usa con el método `strptime()`, el cero inicial es opcional para los formatos `%d`, `%m`, `%H`, `%I`, `%M`, `%S`, `%J`, `%U`, `%W` y `%V`. El formato `%y` requiere un cero a la izquierda.

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 Rein-gold 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] See R. H. van Gent's [guide to the mathematics of the ISO 8601 calendar](#) for a good explanation.
- [4] Si se pasa `datetime.strptime('29 de febrero', '%b %d')` fallará ya que 1900 no es un año bisiesto.