

**Fastbook 08**

**Programación en Python**

**Series temporales**



Edix Educación

## 08. Series temporales

Una de las aplicaciones más clásicas de la analítica de datos es la predicción del comportamiento futuro de datos a lo largo de un eje temporal: por ejemplo, cuántas unidades se venderán de un producto el siguiente mes, o cuál será el precio de la gasolina mañana. Para llegar a realizar estas predicciones, deberemos aprender antes a manejar datos temporales.

En este fastbook aprenderemos a tratar con fechas en Python, introduciremos las series temporales y realizaremos diversas operaciones con ellas en Pandas.

*Autor: Jesús Aguirre Pemán*

≡ Fechas en Python

≡ Introducción a las series temporales

≡ Operaciones con series temporales

≡ Conclusiones

Lesson 1 of 4

# Fechas en Python

X Edix Educación

Para trabajar con fechas en Python, deberemos importar el **módulo datetime**.

```
In [1]: 1 from datetime import date, time, datetime
```

Esta librería cuenta con **tres tipos** importantes para representar objetos temporales.

Tipo	Definición	Atributos
<i>date</i>	Representa una fecha en el calendario	Year, month, day
<i>time</i>	Representa una hora en el día. También incluye información de la zona horaria (tzinfo)	Hour, minute, second, microsecond, tzinfo
<i>datetime</i>	Combina los tipos <i>date</i> y <i>time</i> , representando una fecha con hora	Year, month, day, hour, minute, second, microsecond, tzinfo

## Tipo date

Comencemos definiendo un objeto de tipo *date*.

```
In [2]: 1 navidad = date(2020, 12, 25)
         2 navidad
Out[2]: datetime.date(2020, 12, 25)
```

Como veíamos en la tabla, podemos acceder a sus atributos de año, mes y día.

```
In [3]: 1 navidad.year
Out[3]: 2020
```

## Tipo time

Sigamos con un objeto de tipo *time*, es decir, una hora de un día.

```
In [4]: 1 almuerzo = time(10, 35)
         2 almuerzo
Out[4]: datetime.time(10, 35)
```

Nuestro objeto *almuerzo* marca las 10:35. Observad que hemos especificado solo hora y minutos, con lo que el resto de los atributos toman por defecto el valor 0. ¡Comprobémoslo!

```
In [5]: 1 almuerzo.second
Out[5]: 0
```

## Tipo datetime

Veamos ahora un objeto de tipo *datetime*.

Como veis, por defecto se inicializa la hora a las 00:00.

```
In [6]: 1 datetime(2020, 1, 1)
Out[6]: datetime.datetime(2020, 1, 1, 0, 0)
```

## Operaciones con fechas

Es muy habitual querer realizar operaciones con fechas: por ejemplo, sumar o restar días a una fecha, o sumar o restar minutos a una hora. Para ello, los **operadores de suma y resta** también están disponibles para los objetos *date*, *time* y *datetime*.

Si restamos dos de ellos, obtendremos un **objeto de tipo *timedelta***: una duración, o diferencia entre dos instantes expresada en microsegundos.

En el ejemplo, hemos definido los días de Navidad y Reyes como dos objetos de tipo *date*, y al restarlos hemos obtenido una diferencia de 12 días.

```
In [7]: ► 1 navidad = date(2020, 12, 25)
          2 reyes = date(2021, 1, 6)
          3 dias_pasados = reyes - navidad
          4 dias_pasados
```

Out[7]: datetime.timedelta(days=12)

Para convertir este objeto en un número, podemos acceder a su atributo *days*, con lo que obtendremos el valor 12, o llamar a su método *total\_seconds*, que nos devolverá el tiempo transcurrido en segundos.

```
In [8]: ► 1 dias_pasados.total_seconds()/(24*3600)
```

Out[8]: 12.0

También podemos usar estos objetos *timedelta* para calcular nuevas fechas a partir de otras. Por ejemplo, sumemos un año a la fecha de Reyes de 2021, obteniendo la de 2022. Como veis, se trata de otro objeto *date*, igual que lo era la variable *reyes*.

In [9]: ►

```
1 from datetime import timedelta
2 reyes + timedelta(days = 365)
```

Out[9]: `datetime.date(2022, 1, 6)`

Aunque podemos especificar varios **argumentos temporales** para crear un objeto *timedelta*, solo se almacenarán días, segundos y microsegundos. A continuación, podéis ver la tabla con los argumentos posibles, y su conversión en los atributos que se almacenarán en el objeto.

Atributo	Definición	Conversión	Almacenado
<i>weeks</i>	Semanas	7 days	No
<i>days</i>	Días	-	Sí
<i>hours</i>	Horas	3600 seconds	No
<i>minutes</i>	Minutos	60 seconds	No
<i>seconds</i>	Segundos	-	Sí
<i>milliseconds</i>	Milisegundos	1000 microseconds	No
<i>microseconds</i>	Microsegundos	-	Sí

Vemos cómo en el ejemplo hemos especificado semanas, horas y minutos, y se han convertido automáticamente en días y segundos en el objeto almacenado.

```
In [10]: 1 incremento = timedelta(
2         weeks = 4, hours = 48, minutes = 25*60)
3 incremento
```

Out[10]: `datetime.timedelta(days=31, seconds=3600)`

También es posible crear u obtener un *timedelta* negativo. En ese caso, los días se almacenarán como un *int* negativo, mientras que los segundos y microsegundos permanecerán positivos.

```
In [11]: 1 incremento_negativo = timedelta(
2         weeks = -4, hours = -48, minutes = -25*60)
3 incremento_negativo
```

Out[11]: `datetime.timedelta(days=-32, seconds=82800)`

Al ser esencialmente un número, los objetos *timedelta* soportan la mayoría sus **operaciones**. En la siguiente tabla podéis ver las más habituales.

Operación	Definición
$t_1 = t_2 + t_3$	Suma de dos duraciones
$t_1 = t_2 - t_3$	Resta de dos duraciones
$t_1 = t_2 * i$	Multiplica la duración $t_2$ por un entero $i$
$t_1 = t_2 * f$	Multiplica la duración $t_2$ por un <i>float</i> $f$ , redondeando a microsegundos
$f = t_2 / t_3$	Divide la duración $t_2$ por una duración $t_3$ . Devuelve un <i>float</i>
$abs(t)$	Valor absoluto de la duración (si es negativa, se transformará a positiva)

Por ejemplo, si dividimos el anterior objeto *timedelta* entre 4, vemos cómo se dividen tanto los días como los segundos.

```
In [12]: 1 incremento_negativo/4  
Out[12]: datetime.timedelta(days=-8, seconds=20700)
```

También podemos utilizar la función de valor absoluto para transformar el incremental de tiempo en positivo.

```
In [13]: 1 abs(incremento_negativo)  
Out[13]: datetime.timedelta(days=31, seconds=3600)
```

Por último, al existir las diferencias entre fechas, también podemos utilizar los operadores de comparación entre dos fechas. Una fecha será menor que otra si la precede en el calendario.

```
In [14]: 1 navidad <= reyes  
Out[14]: True
```

## Fechas y strings

Habrá muchas veces que queramos convertir una fecha a formato string para visualizarla correctamente. O realizar el proceso inverso, cuando leamos una fecha de un fichero y no aparezca en el formato correcto.

---

Para ello, tenemos los **métodos strftime** (*string format time*, formatea a string) y **strptime** (*string parse time*, parsea desde string).

Antes de ver ejemplos de estas funciones, debemos introducir el **estándar de notación para fechas que utiliza Python** (1989 C Standard). Esto es simplemente una tabla de equivalencias entre abreviaturas y significados que nos ayudará a la hora de convertir fechas a cadenas de texto y viceversa. En la siguiente tabla podéis ver algunas de las más utilizadas.

Abreviatura	Significado	Ejemplo
%Y	Año completo	2021
%y	Año (dos últimas cifras)	21
%m	Número de mes (incluyendo el 0 a la izquierda en los números de una cifra)	01, ..., 12
%B	Mes (según el idioma local configurado)	January (en_US); Enero (es_ES)
%b	Mes abreviado	Jan (en_US); Ene (es_ES)

Abreviatura	Significado	Ejemplo
%d	Día del mes (incluyendo el 0 a la izquierda en los números de una cifra)	01, ..., 31
%H	Hora del día (formato 24 horas, incluyendo el 0 a la izquierda en los números de una cifra)	00, ..., 23
%M	Minuto (incluyendo el 0 a la izquierda en los números de una cifra)	00, ..., 59
%S	Segundo (incluyendo el 0 a la izquierda en los números de una cifra)	00, ..., 59
%W	Semana del año (comenzando en 0)	00, ..., 53
%w	Número correspondiente al día de la semana (0 para domingo y 6 para el sábado)	0, ..., 6
%A	Día de la semana (abreviado)	Sunday (en_US); Domingo (es_ES)
%a	Día de la semana	Sun (en_US); Dom (es_ES)



Podéis consultar la tabla completa en la [documentación de Python](#).

Ahora sí, veamos cómo **pasar de una fecha a una cadena de texto** aplicando la notación que hemos introducido.

```
In [15]: 1 reyes.strftime("%Y-%m-%d")  
Out[15]: '2021-01-06'
```

También podemos hacer la operación inversa, **pasar de un objeto string a una fecha**. En este caso, el objeto que obtendremos será de tipo *datetime*.

```
In [16]: 1 datetime.strptime("2020-03-05", "%Y-%m-%d")  
Out[16]: datetime.datetime(2020, 3, 5, 0, 0)
```

Si queremos operar con fechas en formato YYYY-MM-DD (el más habitual en analítica de datos), podemos utilizar las **funciones *isoformat* y *fromisoformat***, que utilizarán ese formato, ahorrándonos la necesidad de introducir parámetros adicionales.

```
In [17]: 1 reyes.isoformat()  
Out[17]: '2021-01-06'  
  
In [18]: 1 datetime.fromisoformat("2020-03-05")  
Out[18]: datetime.datetime(2020, 3, 5, 0, 0)  
  
In [19]: 1 date.fromisoformat("2020-03-05")  
Out[19]: datetime.date(2020, 3, 5)
```

## Métodos sobre fechas

En este apartado vamos a aprender una serie de métodos que tenemos disponibles para los objetos *date* o *datetime*.

1

*Today*

El método *today* nos permite crear un objeto del tipo *date* o *datetime* que contendrá la fecha actual (o fecha y hora si es el perteneciente a *datetime*).

```
In [20]: 1 datetime.today()  
Out[20]: datetime.datetime(2021, 6, 8, 17, 12, 31, 43979)
```

2

*Weekday*

El método *weekday* nos permite obtener el día de la semana a partir de un objeto *date* o *datetime*. Concretamente, obtendremos un número entre 0 y 6, donde el 0 representa el domingo, el 1 el lunes, y así hasta llegar al sábado (6). Observad que se comienza en domingo y no en lunes, como corresponde a los calendarios norteamericanos.

```
In [21]: 1 reyes.weekday()  
Out[21]: 2
```

3

*Replace*

La función *replace* permite cambiar alguno de los atributos del objeto, creando uno nuevo con ese cambio aplicado.

```
In [22]: 1 reyes.replace(year = 2022)
```

```
Out[22]: datetime.date(2022, 1, 6)
```

4

*Combine*

La función *combine* nos permite crear un objeto *datetime* a partir de una fecha (objeto *date*) y una hora (objeto *time*).

```
In [23]: 1 datetime.combine(date(2020, 12, 25), time(10, 35))
```

```
Out[23]: datetime.datetime(2020, 12, 25, 10, 35)
```

Lesson 2 of 4

# Introducción a las series temporales

 Edix Educación

---



Las series temporales son sucesiones de datos a lo largo de una escala temporal, que puede ser de intervalos fijos o desiguales.

En el mundo de Analytics tienen una gran importancia, con modelos específicos para este tipo de datos como pueden ser ARIMA o prophet.

---

**Durante este apartado y los siguientes, trabajaremos con series y dataframes de Pandas, que contendrán al menos una columna (o el índice) con un dato temporal.**

## Fechas en Pandas

Antes de comenzar con las series temporales, vamos a introducir los tipos de **datos de fechas disponibles en Pandas**, y cómo se relacionan con sus equivalentes en el módulo *datetime* de Python.

Pandas define los siguientes 4 tipos de datos, aunque nosotros utilizaremos principalmente los primeros dos.

Def.	Clase Escalar	Clase de Serie	Clase de Índice	Equivalente de datetime	Método de creación
Fecha con hora	Timestamp	datetime64[ns]	DatetimeIndex	datetime	to_datetime, date_range
Diferencia temporal	Timedelta	timedelta64[ns]	TimedeltaIndex	timedelta	to_timedelta, timedelta_range
Periodo	Period	period[freq]	PeriodIndex	-	Period, period_range
Duración relativa	DateOffset	-	-	relativedelta	Dateoffset

Veamos en profundidad estos conceptos.

## Fechas con hora

1

### Timestamps

Comencemos con los *Timestamps*. Como veíamos en la tabla, su equivalente en el módulo *datetime* serían los objetos *datetime*, ya que ambos incluyen información de fecha y hora.

Podemos crear un objeto *Timestamp* mediante la función de Pandas del mismo nombre. En su propia documentación nos indica que estos objetos son intercambiables con objetos *datetime* en la mayoría de los contextos.

```
In [25]: 1 pd.Timestamp("2021-01-01")
```

```
Out[25]: Timestamp('2021-01-01 00:00:00')
```

No obstante, lo más común es transformar un objeto compatible (*string*, *datetime*, lista o array unidimensional) a un *Timestamp* mediante la función de Pandas *to\_datetime*.

```
In [26]: 1 pd.to_datetime("2021-01-01")  
Out[26]: Timestamp('2021-01-01 00:00:00')
```

2

## Índices temporales

Sigamos profundizando en los objetos *datetime*.

Crearemos una serie cuyo índice sean objetos de este tipo.

```
In [27]: 1 indice_temporal = [datetime(2021,1,1), datetime(2021,1,2),  
2                               datetime(2021,1,3)]  
3 serie_temporal = pd.Series([200, 250, 180], index = indice_temporal)  
4 serie_temporal  
Out[27]: 2021-01-01    200  
2021-01-02    250  
2021-01-03    180  
dtype: int64
```

Como veis, hemos construido una serie con valores para los tres primeros días del año.

Veamos de qué tipo es el índice de la serie.

```
In [28]: 1 serie_temporal.index  
Out[28]: DatetimeIndex(['2021-01-01', '2021-01-02', '2021-01-03'], dtype='datetime64  
[ns]', freq=None)
```

Tal y como aparece en la tabla anterior, el índice es un objeto *DatetimeIndex*, y su tipo *datetime64[ns]*, indicando que se mide en nanosegundos.

Al igual que con los demás índices, podemos utilizarlos para seleccionar datos.

```
In [29]: 1 serie_temporal["2021-01-01"]

Out[29]: 200
```

Para crear una secuencia de frecuencia fija, utilizaremos la función `date_range` de Pandas. Esta función recibe los parámetros `start` y `end` para su comienzo y final, y `periods` y `freq`, para el número de periodos, y su frecuencia. Para definir la secuencia, necesitaremos 3 de ellos. Veamos un ejemplo donde creamos una secuencia con los primeros 7 días del año, definiendo `start`, `end` y `freq`.

```
In [30]: pd.date_range("2021-01-01", "2021-01-7", freq = "D")  
Out[30]: DatetimeIndex(['2021-01-01', '2021-01-02', '2021-01-03', '2021-01-04',  
                      '2021-01-05', '2021-01-06', '2021-01-07'],  
                     dtype='datetime64[ns]', freq='D')
```

El parámetro *freq* nos da mucha flexibilidad para crear la secuencia. Por ejemplo, podemos indicar que es semanal y que se sitúa en los lunes.

En la siguiente tabla tenéis algunas de las **frecuencias más utilizadas**.

Abreviatura	Significado
<b>A-JAN, A-FEB, ...</b>	Anual, en el último día del mes indicado (utilizando las abreviaturas inglesas)
<b>AS-JAN, AS-FEB, ...</b>	Anual, en el primer día del mes indicado (utilizando las abreviaturas inglesas)
<b>M</b>	Mensual, en el último día del mes
<b>MS</b>	Mensual, en el primer día del mes
<b>W</b>	Semanal, en domingo
<b>W-MON, W-TUE, ...</b>	Semanal, en el día especificado de la semana (utilizando las abreviaturas inglesas)
<b>D</b>	Diaria
<b>B</b>	Diaria, entre semana ( <i>business day</i> )
<b>H</b>	Con frecuencia de horas
<b>T</b>	Con frecuencia de minutos
<b>S</b>	Con frecuencia de segundos

3

## Columnas temporales

Hasta ahora hemos trabajado con fechas en índices de series, pero es muy común encontrarnos con fechas en columnas de dataframes. Creemos un dataframe con una columna de fechas utilizando el método *date\_range*.

```
In [32]: 1 grps_tv_df = pd.DataFrame({  
2     "fecha": pd.date_range("2021-01-01", periods = 4,  
3                               freq = "W-MON"),  
4     "grps_tv": [300, 250, 315, 280]})  
5 grps_tv_df
```

Out[32]:

	fecha	grps_tv
0	2021-01-04	300
1	2021-01-11	250
2	2021-01-18	315
3	2021-01-25	280

Si consultamos el tipo de la columna *fecha*, veremos que es *datetime64[ns]* (igual que en el índice del apartado anterior).

In [33]: 1 grps\_tv\_df.dtypes

```
Out[33]: fecha      datetime64[ns]  
grps_tv           int64  
dtype: object
```

Concretamente, cada valor de la columna *fecha* es un objeto *Timestamp*.

In [34]: 1 grps\_tv\_df.iloc[0].fecha

Out[34]: Timestamp('2021-01-04 00:00:00')

## Periodos

Los periodos (o *Time Span* en la tabla anterior y en la documentación de Pandas) se corresponden con **intervalos de tiempo**, en contraposición a los instantes concretos que se representan en los *Timestamps*.

Podemos crearlos mediante la **función *Period*** de Pandas.

```
In [35]: 1 pd.Period("2021-01")
```

```
Out[35]: Period('2021-01', 'M')
```

Esta función también acepta un parámetro de frecuencia (*freq*), y parámetros adicionales para especificar el periodo (*year*, *month*, *quarter*, *day*, *hour*, *minute* y *second*).

```
In [36]: 1 pd.Period(year = 2021, month = 1, freq = "M")
```

```
Out[36]: Period('2021-01', 'M')
```

Por lo demás, su comportamiento es muy similar a los objetos *Timestamp* y sus derivados. Por ejemplo, podemos crear índices temporales con periodos donde veremos cómo aparece reflejada la frecuencia (mensual en el ejemplo).

```
In [37]: 1 indice_periodos = [pd.Period("2021-01"), pd.Period("2021-02"),
 2                               pd.Period("2021-03")]
 3 serie_periodos = pd.Series([200, 250, 180], index = indice_periodos)
 4 serie_periodos
```

```
Out[37]: 2021-01    200
2021-02    250
2021-03    180
Freq: M, dtype: int64
```

Si quisieramos pasar el índice con períodos a *Timestamps*, podemos utilizar el método *to\_timestamp*, que por defecto creará la fecha en el comienzo del periodo.

```
In [38]: 1 serie_periodos.to_timestamp()
Out[38]: 2021-01-01    200
          2021-02-01    250
          2021-03-01    180
Freq: MS, dtype: int64
```

De igual forma, para transformar *Timestamps* en períodos utilizaremos la función *to\_period*.

```
In [39]: 1 serie_periodos.to_timestamp().to_period()
Out[39]: 2021-01    200
          2021-02    250
          2021-03    180
Freq: M, dtype: int64
```

También tenemos disponible la función *period\_range* para crear rangos de períodos (concretamente, un *PeriodIndex*).

```
In [40]: 1 pd.period_range("2021-01", periods=4, freq = "M")
Out[40]: PeriodIndex(['2021-01', '2021-02', '2021-03', '2021-04'], dtype='period[M]', freq='M')
```

Y, por último, utilizar los períodos en columnas de un dataframe.

```
In [41]: 1 grps_tv_df = pd.DataFrame({
          2     "fecha": pd.period_range("2021-01", periods = 4, freq = "M"),
          3     "grps_tv": [300, 250, 315, 280]})

Out[41]:
      fecha  grps_tv
      0  2021-01    300
      1  2021-02    250
      2  2021-03    315
      3  2021-04    280
```

En este caso, la columna de fecha será de tipo period[M], donde la M hace referencia a la frecuencia mensual.

```
In [42]: 1 grps_tv_df.dtypes  
Out[42]: fecha      period[M]  
          grps_tv    int64  
          dtype: object
```

## Duraciones temporales

En Pandas, las diferencias temporales (o duraciones de tiempo) se representan mediante objetos *Timedelta*. Su equivalente en datetime es la clase *timedelta*.

### Creación

Para crear un objeto de esta clase, utilizaremos la función del mismo nombre. Podemos escribir la duración en formato de string, algo muy cómodo.

```
In [43]: 1 pd.Timedelta("1 day 2 hours")  
Out[43]: Timedelta('1 days 02:00:00')
```

Por supuesto, también podemos utilizar los mismos argumentos que en los objetos *Timedelta* de datetime.

```
In [44]: 1 pd.Timedelta(weeks = 4, hours = 48, minutes = 25*60)  
Out[44]: Timedelta('31 days 01:00:00')
```

Y al igual que en esos objetos, podemos tener diferencias negativas.

```
In [45]: 1 pd.Timedelta("-1 day 2 hours")  
Out[45]: Timedelta('-2 days +22:00:00')
```

De forma similar a las fechas con hora, podemos transformar objetos como cadenas o números a *Timedeltas*. Además, podemos indicar la unidad temporal con el parámetro *unit*.

```
In [46]: 1 pd.to_timedelta(range(1,5), unit="H")  
Out[46]: TimedeltaIndex(['01:00:00', '02:00:00', '03:00:00', '04:00:00'], dtype='timedelta64[ns]', freq=None)
```

## Operaciones con Timedeltas

Al igual que en sus homólogos de *datetime*, la mayor utilidad de estos objetos es realizar operaciones con fechas. Y precisamente aquí es donde más partido vamos a sacar al hecho de utilizar la librería Pandas, ya que haremos estas operaciones sobre columnas de dataframes.

```
In [47]: 1 grps_tv_df = pd.DataFrame({  
2     "fecha": pd.date_range("2021-01-01", periods = 4,  
3         freq = "W-MON"),  
4     "grps_tv": [300, 250, 315, 280]})  
5 grps_tv_df = grps_tv_df.assign(fecha_fin = lambda x: x.fecha +\n6                                     pd.Timedelta(days = 6))  
7 grps_tv_df
```

```
Out[47]:  
      fecha  grps_tv  fecha_fin  
0  2021-01-04      300  2021-01-10  
1  2021-01-11      250  2021-01-17  
2  2021-01-18      315  2021-01-24  
3  2021-01-25      280  2021-01-31
```

Recuperando el ejemplo anterior, hemos creado una nueva columna que calcula el día final de la semana sumando 6 días a los valores de la columna `fecha`.

```
In [48]: 1 grps_tv_df = grps_tv_df.assign(diferencia = lambda x: x.fecha_fin - x.fecha)
          2
          3 grps_tv_df
```

Out[48]:

	fecha	grps_tv	fecha_fin	diferencia
0	2021-01-04	300	2021-01-10	6 days
1	2021-01-11	250	2021-01-17	6 days
2	2021-01-18	315	2021-01-24	6 days
3	2021-01-25	280	2021-01-31	6 days

Lógicamente, si restamos dos objetos de tipo `datetime64`, obtendremos un objeto `timedelta64`.

```
In [49]: 1 grps_tv_df.dtypes
```

```
Out[49]: fecha      datetime64[ns]
          grps_tv     int64
          fecha_fin   datetime64[ns]
          diferencia  timedelta64[ns]
          dtype: object
```

## Offsets

Además de los objetos `Timedelta`, Pandas incluye una serie de incrementales de tiempo, como `Day`, `Hour`, `Minute` o `Week`. Antes de utilizarlos, deberéis importarlos de `pandas.tseries.offsets`. El funcionamiento es muy similar al que veíamos en los `Timedeltas`.

```
In [50]: 1 from pandas.tseries.offsets import Week
          2 grps_tv_df.assign(siguiente_anyo = lambda x: x.fecha_fin + Week(52))
```

Out[50]:

	fecha	grps_tv	fecha_fin	diferencia	siguiente_anyo
0	2021-01-04	300	2021-01-10	6 days	2022-01-09
1	2021-01-11	250	2021-01-17	6 days	2022-01-16
2	2021-01-18	315	2021-01-24	6 days	2022-01-23
3	2021-01-25	280	2021-01-31	6 days	2022-01-30

Si trabajamos con zonas horarias o calendarios de verano/invierno, también nos puede interesar utilizar los objetos de duración relativa *DateOffset*.

Por ejemplo, si nos situamos en la zona horaria europea (CET, *Central European time*) justo antes del cambio al horario de verano de 2021 (la noche del sábado 27 al domingo 28 de marzo), y sumamos y calculamos el incremento de 1 día mediante un objeto *Timedelta*, veremos que la hora resultante son las 2 a. m. del siguiente día (en lugar de las 1 a. m. de nuestro objeto original).

```
In [51]: 1 fecha_cambio_hora = pd.Timestamp('2021-03-28 01:00:00', tz='CET')
          2 fecha_cambio_hora + pd.Timedelta(days=1)

Out[51]: Timestamp('2021-03-29 02:00:00+0200', tz='CET')
```

Si por el contrario utilizamos un objeto *DateOffset*, se respeta esta aritmética de calendario a pesar del cambio de hora, obteniendo las 1 a. m. del día posterior.

```
In [52]: 1 fecha_cambio_hora + pd.DateOffset(days=1)

Out[52]: Timestamp('2021-03-29 01:00:00+0200', tz='CET')
```

## Missing values en fechas

Pandas representa los *missing values* en fechas, duraciones temporales y periodos mediante el objeto *NaT* (*not a time*).

```
In [53]: 1 fecha_na = pd.Timestamp(pd.NaT)
          2 fecha_na
```

Out[53]: NaT

Al igual que con los *missing values* tradicionales, deberemos utilizar el **método *isna*** para comprobar si se trata de un objeto *NaT*.

```
In [54]: pd.isna(fecha_na)
```

Out[54]: True

Recordad que si utilizáis el operador de comparación en lugar del método *isna* no obtendréis el resultado esperado.

```
In [55]: 1 print(fecha_na== pd.NaT)
          2 print(fecha_na== np.NaN)
```

False  
False

## Componentes de un objeto *DateTime*

Ya hemos visto que los objetos de la librería *datetime* tenían atributos (por ejemplo, *month* o *year*). No es menos en el caso de sus análogos de Pandas, y además podremos disponer de ellos como series. A continuación, tenéis una tabla con los principales.

Atributo	Significado
<i>year</i>	Año
<i>month</i>	Mes
<i>day</i>	Día
<i>hour</i>	Hora
<i>minute</i>	Minuto
<i>second</i>	Segundo
<i>date</i>	Objeto <i>date</i> con la fecha
<i>time</i>	Objeto <i>time</i> con la hora
<i>dayofyear</i>	Día del año
<i>week</i>	Semana del año
<i>weekday</i>	Día de la semana (desde 0 para el lunes, hasta 6 para el domingo)



Podéis consultar la lista completa en [este enlace](#).

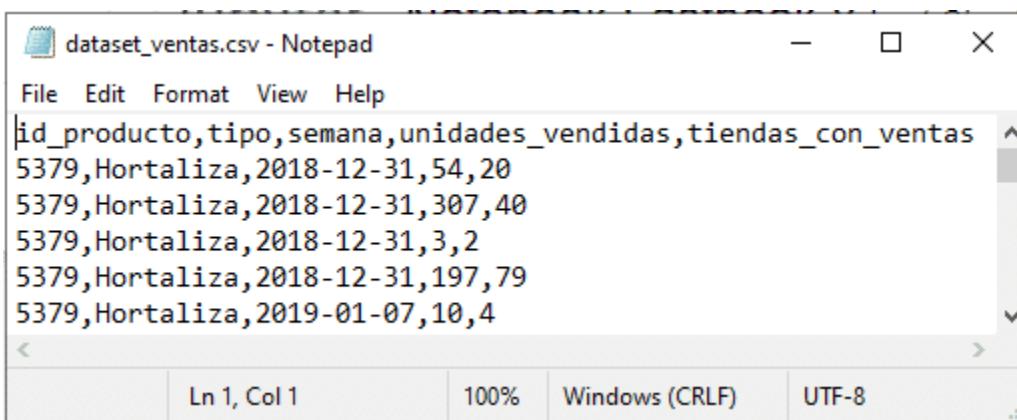
Para acceder a ellos, utilizaremos el atributo `dt` de la serie, y sobre este ya podremos seleccionar el atributo que queramos.

```
In [56]: 1 grps_tv_df\  
2     .assign(year = lambda x: x.fecha.dt.year)  
  
Out[56]:  
      fecha  grps_tv  fecha_fin  diferencia  year  
0  2021-01-04       300  2021-01-10    6 days  2021  
1  2021-01-11       250  2021-01-17    6 days  2021  
2  2021-01-18       315  2021-01-24    6 days  2021  
3  2021-01-25       280  2021-01-31    6 days  2021
```

## Carga de datos

En el fastbook 05 aprendimos a cargar datos con Pandas, usando las funciones `read_csv` o `read_excel`. En este apartado vamos a ver algunos de los parámetros de estas funciones relacionados con las fechas.

Para ello cargaremos el fichero `dataset_ventas.csv`.



En primer lugar, carguemoslo sin utilizar ningún parámetro adicional.

```
In [57]: 1 dataset_ventas = pd.read_csv("../data/dataset_ventas.csv")
2 dataset_ventas.head(5)
```

Out[57]:

	id_producto	tipo	semana	unidades_vendidas	tiendas_con_ventas
0	112	Hortaliza	2018-12-31	802	71
1	112	Hortaliza	2019-01-07	2351	195
2	112	Hortaliza	2019-01-14	1430	137
3	112	Hortaliza	2019-01-21	464	64
4	112	Hortaliza	2019-01-28	2069	195

```
In [58]: 1 dataset_ventas.dtypes
```

```
Out[58]: id_producto          int64
         tipo                object
         semana               object
         unidades_vendidas    int64
         tiendas_con_ventas   int64
         dtype: object
```

Si utilizamos el parámetro *parse\_dates* para indicar que la columna semana contiene datos de formato fecha, Pandas los procesará como tal.

```
In [59]: 1 dataset_ventas = pd.read_csv("../data/dataset_ventas.csv",
2                               parse_dates= ["semana"])
3 dataset_ventas.dtypes
```

```
Out[59]: id_producto          int64
         tipo                object
         semana              datetime64[ns]
         unidades_vendidas    int64
         tiendas_con_ventas   int64
         dtype: object
```

Otra opción es realizar la conversión tras leer el fichero. Para ello, simplemente usaremos la función `to_datetime` de Pandas sobre la columna `semana`, con lo que obtendremos el mismo resultado. Además, podemos indicar el formato con el parámetro `format`, utilizando el estándar que hemos visto en la sección anterior.

```
In [60]: 1 dataset_ventas = pd.read_csv("../data/dataset_ventas.csv")\
2         .assign(semana = lambda x: pd.to_datetime(x.semana))
3 dataset_ventas.dtypes
```

```
Out[60]: id_producto           int64
          tipo                  object
          semana            datetime64[ns]
          unidades_vendidas    int64
          tiendas_con_ventas   int64
          dtype: object
```

Lesson 3 of 4

# Operaciones con series temporales

 Edix Educación

---

Ahora que ya hemos aprendido cómo crear series temporales en Pandas, es hora de ver algunas funciones que nos permitirán trabajar cómodamente con ellas.

## Desplazamientos

Una de las operaciones más comunes en series temporales es realizar desplazamientos atrás y adelante en el tiempo (también conocidos como *lags* y *leads*).

---

**En Pandas podemos realizar ambas operaciones mediante la función `shift`.**

---

Para el siguiente ejemplo, vamos a tomar un producto del dataframe que hemos cargado en el anterior apartado, y establecer la columna temporal como el índice.

```
In [61]: 1 dataset_ventas_producto = dataset_ventas\  
2     .loc[lambda x: x.id_producto == 5379]\\  
3     .set_index("semana")  
4 dataset_ventas_producto
```

Out[61]:

	id_producto	tipo	unidades_vendidas	tiendas_con_ventas
semana				
2018-12-31	5379	Hortaliza	54	20
2019-01-07	5379	Hortaliza	10	4
2019-01-14	5379	Hortaliza	5	2
2019-01-21	5379	Hortaliza	309	32
2019-01-28	5379	Hortaliza	202	69

Sobre este dataframe podemos ejecutar el método *shift* para desplazar todas las columnas una semana.

Observad que, al realizar este desplazamiento, la primera semana se queda sin dato, y por lo tanto, aparece como NA.

```
In [62]: 1 dataset_ventas_producto.shift()
```

Out[62]:

	id_producto	tipo	unidades_vendidas	tiendas_con_ventas
semana				
2018-12-31	NaN	NaN	NaN	NaN
2019-01-07	5379.0	Hortaliza	54.0	20.0
2019-01-14	5379.0	Hortaliza	10.0	4.0
2019-01-21	5379.0	Hortaliza	5.0	2.0
2019-01-28	5379.0	Hortaliza	309.0	32.0

Con el parámetro *periods* podemos variar el número de periodos que queremos desplazar el dato, pudiendo ser también desplazamientos negativos (o *leads*). Además, con el parámetro *fill\_value* podemos llenar estos valores NA.

En este caso, lo que habría quedado con valores NA son las últimas dos semanas del histórico, pero les hemos asignado el valor 0. Observad que se utiliza el mismo valor para todas las columnas, por lo que deberéis usarlo con cuidado en el caso de que haya columnas no numéricas.

```
In [63]: 1 dataset_ventas_producto.shift(-2, fill_value = 0)
```

Out[63]:

semana	id_producto	tipo	unidades_vendidas	tiendas_con_ventas
2018-12-31	5379	Hortaliza	5	2
2019-01-07	5379	Hortaliza	309	32
2019-01-14	5379	Hortaliza	202	69
2019-01-21	5379	Hortaliza	296	36
2019-01-28	5379	Hortaliza	16	6
...	...	...	...	...
2020-11-30	5379	Hortaliza	63	19
2020-12-07	5379	Hortaliza	16	8
2020-12-14	5379	Hortaliza	3	2
2020-12-21	0	0	0	0
2020-12-28	0	0	0	0

El parámetro *freq* nos permite introducir un objeto temporal (como un *Timedelta* o un *offset*) para desplazar las series.

Observad que este desplazamiento es muy similar al primero de este apartado, pero al utilizar el parámetro *freq* no obtenemos la primera fila de NAs, sino que desplazamos el índice (en lugar de desplazar todas las columnas).

```
In [64]: 1 dataset_ventas_producto.shift(freq = pd.Timedelta("7 days"))
```

Out[64]:

	id_producto	tipo	unidades_vendidas	tiendas_con_ventas
semana				
2019-01-07	5379	Hortaliza	54	20
2019-01-14	5379	Hortaliza	10	4
2019-01-21	5379	Hortaliza	5	2
2019-01-28	5379	Hortaliza	309	32
2019-02-04	5379	Hortaliza	202	69

Por último, también tenemos la función *shift* disponible para series. De esta forma, no necesitaremos tener el dataframe indexado por una columna temporal para realizar desplazamientos en una columna.

```
In [65]: 1 dataset_ventas_producto.reset_index()\n2     .assign(unidades_lag = lambda x: x.unidades_vendidas.shift(1))
```

Out[65]:

	semana	id_producto	tipo	unidades_vendidas	tiendas_con_ventas	unidades_lag
0	2018-12-31	5379	Hortaliza	54	20	NaN
1	2019-01-07	5379	Hortaliza	10	4	54.0
2	2019-01-14	5379	Hortaliza	5	2	10.0
3	2019-01-21	5379	Hortaliza	309	32	5.0
4	2019-01-28	5379	Hortaliza	202	69	309.0

## Cambios de frecuencia

Es habitual que manejemos series temporales con diferentes frecuencias y queramos combinarlas en un solo dataframe. Para ello, necesitamos aprender a cambiar la granularidad o frecuencia de las series.

Hay dos opciones principales en el cambio de frecuencia:

- Agregar datos de mayor frecuencia en una frecuencia más baja (por ejemplo, pasar de dato semanal a mensual), también llamado *downsampling*.
- Transformar datos de menor frecuencia a mayor granularidad (el cambio contrario, pasar de dato mensual a semanal), que se denomina *upsampling*.

---

**En Pandas, realizaremos los cambios de frecuencia mediante la función *resample*.**

---

Le deberemos indicar la regla (parámetro *rule*) mediante un objeto *Timedelta* o incluso un string como los que veíamos en la tabla de frecuencias.

### Downsampling

Como decíamos, el proceso de *downsampling* consiste en reducir la granularidad de nuestros datos temporales. Aplicándolo a nuestro dataframe de ventas, podríamos pasar de la actual granularidad semanal a mensual.

```
In [66]: 1 dataset_ventas_producto.resample("M")
```

```
Out[66]: <pandas.core.resample.DatetimeIndexResampler object at 0x0000029A91F48048>
```

Como veis, el resultado de la función es un objeto *DatetimeIndexResampler*, similar a los *DataFrameGroupBy* en cuanto a que necesitamos aplicar una función de agrupación para completar el cambio de frecuencia. En nuestro ejemplo, estábamos pasando a una frecuencia mensual, con lo que una buena forma de agrupar los datos de unidades vendidas sería mediante la suma.

```
In [67]: 1 dataset_ventas_producto.resample("M").sum()
```

Out[67]:

	id_producto	unidades_vendidas	tiendas_con_ventas
semana			
2018-12-31	5379	54	20
2019-01-31	21516	526	107
2019-02-28	21516	438	79
2019-03-31	21516	496	162
2019-04-30	26895	536	104

Debéis tener cuidado con las agrupaciones que hagáis en los cambios de frecuencia: por ejemplo, las tiendas con ventas no se deberían sumar, ya que probablemente haya tiendas en común en las ventas de las semanas que estamos agregando.

En el ejemplo anterior, el dato de cada mes aparece representado por el último día. Si queremos utilizar un periodo en lugar de una fecha para tener mayor congruencia con el significado del dato, *resample* tiene un parámetro *kind* que nos realiza esta transformación de forma automática.

```
In [68]: 1 dataset_ventas_producto.resample("M", kind = "period").sum()
```

Out[68]:

	id_producto	unidades_vendidas	tiendas_con_ventas
semana			
2018-12	5379	54	20
2019-01	21516	526	107
2019-02	21516	438	79
2019-03	21516	496	162
2019-04	26895	536	104

## Upsampling

Para obtener mayor granularidad de una serie temporal, en lugar de agregar los datos, necesitaremos una función que nos proporcione valores para ‘rellenar’ los nuevos índices temporales.

Por ejemplo, pasemos nuestra serie anterior, de granularidad semanal, a una serie diaria. Si no queremos realizar ninguna transformación, podemos utilizar el método `asfreq`, que simplemente transforma el índice de la serie a la frecuencia que le proporcionemos.

Como veis, se mantiene el dato de nuestro dataframe original en el 31 de diciembre (lunes), pero no tenemos dato para las nuevas filas correspondientes al resto de días de la semana.

```
In [69]: 1 dataset_venta_producto.asfreq("D")
```

Out[69]:

	id_producto	tipo	unidades_vendidas	tiendas_con_ventas
semana				
2018-12-31	5379.0	Hortaliza	54.0	20.0
2019-01-01	NaN	NaN	NaN	NaN
2019-01-02	NaN	NaN	NaN	NaN
2019-01-03	NaN	NaN	NaN	NaN
2019-01-04	NaN	NaN	NaN	NaN

Como recordaréis del fastbook anterior, podemos utilizar las técnicas de `ffill` o `bfill` para llenar *missing values*. Además de introducirlos como parámetro `method` en `fillna`, también disponen de sus propios métodos.

```
In [70]: 1 dataset_venta_producto.asfreq("D").ffill()
```

Out[70]:

	id_producto	tipo	unidades_vendidas	tiendas_con_ventas
semana				
2018-12-31	5379.0	Hortaliza	54.0	20.0
2019-01-01	5379.0	Hortaliza	54.0	20.0
2019-01-02	5379.0	Hortaliza	54.0	20.0
2019-01-03	5379.0	Hortaliza	54.0	20.0
2019-01-04	5379.0	Hortaliza	54.0	20.0

Si dividimos los valores entre 7, estaríamos repartiendo el dato semanal entre cada día de la semana. De nuevo recordad que la columna de tiendas con ventas no deberíamos transformarla mediante este método.

No obstante, este método de repartición puede ser insuficiente en muchas situaciones, por lo que en el siguiente apartado explicaremos técnicas más sofisticadas.

## ***Rolling windows***

Las operaciones de ventana deslizante (o *rolling window*) son transformaciones sobre una ventana temporal de longitud fija que se va desplazando a lo largo de la serie. Retomando el ejemplo de nuestra serie semanal, esta ventana podría ser, por ejemplo, de 4 semanas.

Una vez que hayamos definido la amplitud de la ventana, deberemos indicar qué operación queremos realizar con los datos pertenecientes a ella. Una de las más utilizadas es la media, dando lugar así a la serie conocida como media móvil (*moving average*).

---

**En Pandas disponemos de la función *rolling* para realizar estas transformaciones.**

---

Recuperemos nuestro dataset de ventas de un producto para realizar unos ejemplos.

In [71]: 1 dataset\_ventas\_producto

Out[71]:

	id_producto	tipo	unidades_vendidas	tiendas_con_ventas
semana				
2018-12-31	5379	Hortaliza	54	20
2019-01-07	5379	Hortaliza	10	4
2019-01-14	5379	Hortaliza	5	2
2019-01-21	5379	Hortaliza	309	32
2019-01-28	5379	Hortaliza	202	69

Probemos ahora a realizar la media móvil con una ventana de 4 semanas.

In [72]: 1 dataset\_ventas\_producto.rolling(4).mean()

Out[72]:

	id_producto	unidades_vendidas	tiendas_con_ventas
semana			
2018-12-31	NaN	NaN	NaN
2019-01-07	NaN	NaN	NaN
2019-01-14	NaN	NaN	NaN
2019-01-21	5379.0	94.50	14.50
2019-01-28	5379.0	131.50	26.75

Si aplicamos la función *rolling* sobre el dataframe, se realizará la operación de ventana deslizante sobre todas las columnas. Si lo preferimos, podemos hacer la media móvil directamente en una columna, con lo que conservaremos el dato original.

```
In [73]: dataset_ventas_producto_rolling = dataset_ventas_producto\
           [["unidades_vendidas"]]\n           .assign(unidades_vendidas_rolling = lambda x:\n                  x.unidades_vendidas.rolling(4).mean())\n\n           dataset_ventas_producto_rolling
```

Out[73]:

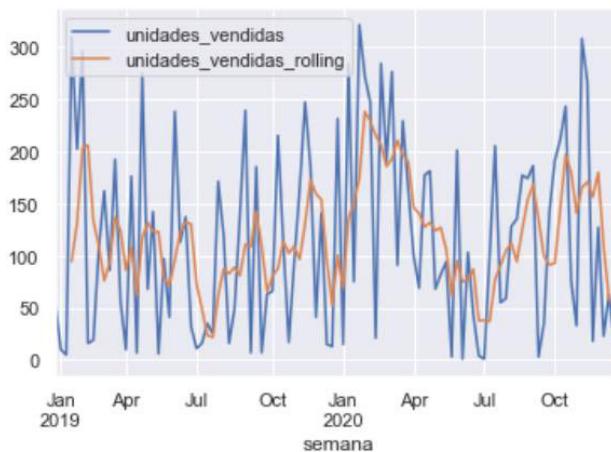
	unidades_vendidas	unidades_vendidas_rolling
semana		
2018-12-31	54	NaN
2019-01-07	10	NaN
2019-01-14	5	NaN
2019-01-21	309	94.50
2019-01-28	202	131.50

Veamos gráficamente cómo ha cambiado la variable al aplicar la media móvil.

Como veis en el gráfico, la serie original tiene muchos picos, y con la media móvil hemos conseguido suavizarlos y obtener una serie más estable.

```
In [74]: dataset_ventas_producto_rolling\
           .reset_index()\n           .plot(x = "semana",\n                 y = ["unidades_vendidas", "unidades_vendidas_rolling"])
```

Out[74]: <matplotlib.axes.\_subplots.AxesSubplot at 0x22231882dd8>



## Interpolación

Aunque ya hemos visto cómo rellenar *missing values* con los valores previos o posteriores, si trabajamos con series temporales habitualmente preferiremos un término intermedio entre ambos valores.

En Pandas disponemos de la función *interpolate*, que nos permitirá realizar diferentes tipos de interpolación para llenar esos valores perdidos.

Creemos una serie temporal con valores perdidos para ver su funcionamiento.

```
In [75]: ► 1  dataset_ventas_missing = pd.DataFrame(  
2      {"semana": pd.date_range("2021-05-31", periods = 4,  
3          freq = "w-mon"),  
4      "ventas": [250, np.nan, 265, 245]  
5      })  
6  dataset_ventas_missing
```

Out[75]:

	semana	ventas
0	2021-05-31	250.0
1	2021-06-07	NaN
2	2021-06-14	265.0
3	2021-06-21	245.0

Como veis, no tenemos disponible el segundo dato de la serie. Utilizando el método *interpolate*, podemos estimarlo mediante interpolación lineal. En este caso, al solo haber un valor perdido, se rellenará con la media de los valores anterior y posterior.

```
In [76]: 1 dataset_ventas_missing.interpolate()
```

Out[76]:

	semana	ventas
0	2021-05-31	250.0
1	2021-06-07	257.5
2	2021-06-14	265.0
3	2021-06-21	245.0

Aunque por defecto *interpolate* usa la interpolación lineal, acepta otros métodos de interpolación (podéis ver la lista completa en la documentación de la función). Por ejemplo, utilicemos interpolación cuadrática.

En este caso hemos obtenido un dato diferente, ligeramente superior al valor de la tercera semana.

```
In [77]: 1 dataset_ventas_missing.interpolate("quadratic")
```

Out[77]:

	semana	ventas
0	2021-05-31	250.000000
1	2021-06-07	266.666667
2	2021-06-14	265.000000
3	2021-06-21	245.000000

# Conclusiones

X Edix Educación

---

En este fastbook hemos aprendido a manejarnos con fechas en Python, utilizando tanto el módulo *datetime* como las funciones propias de la librería Pandas. Además, hemos realizado operaciones con fechas y períodos, y utilizado datos temporales en columnas e índices de dataframes.

Asimismo, hemos aprendido que las series temporales son sucesiones de datos a lo largo de una escala temporal, y que podemos realizar desplazamientos sobre ellas para calcular datos posteriores o anteriores. Finalmente, hemos realizado cambios de frecuencias, para pasar tanto a series más agrupadas como más desagregadas, y hemos concluido el fastbook introduciendo las operaciones de ventana deslizante o *rolling window*.

---

**Nuestro camino de aprendizaje sobre la programación en Python y Pandas ya está casi completado, y en el siguiente fastbook nos centraremos en el análisis exploratorio de datos, repasando y ampliando conceptos para poder hacer un pequeño modelo que explique las ventas de un producto.**

¡Enhorabuena! Fastbook superado

edix

Creamos Digital Workers