PANDAS TAREA Isaac Fermin 2024-0397

In [6]:
```
import numpy as np
import pandas as pd
```

Lo primero que nos dice la documentación es cómo importar la libreria de Pandas y también la de Numpy.

In [18]:
```
s = pd.Series([1,3,5,np.nan,6,8])
s
```

Out[18]:
```
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

Aquí estamos creando una Serie pasando una lista de valores, pero dejando que Pandas cree un RangeIndex por defecto. Una Serie es un array de una sola dimensión y puede almacenar cualquier tipo de datos en su interior, por ejemplo: enteros, cadenas de texto, objetos de Python, etc.

In [22]:
```
dates = pd.date_range("20130101", periods = 6)
dates

df = pd.DataFrame(np.random.randn(6,4), index = dates, columns = list("ABDC"))
df
```

Out[22]:

	A	B	D	C
2013-01-01	-0.503532	-0.583556	-1.448177	-0.002650
2013-01-02	-0.702040	-0.574853	-1.083638	-0.995300
2013-01-03	-1.666242	0.513538	0.292049	-0.797981
2013-01-04	-0.131445	-1.846680	-0.249913	-0.867205
2013-01-05	-1.510444	0.406436	-0.742640	-2.374353
2013-01-06	0.851498	-0.718269	-0.786879	-1.089264

En el objeto dates, lo que estamos haciendo es usar la función date_range. Esto genera una cantidad de fechas partiendo de una fecha inicial. La fecha que utilizamos en el ejemplo es 2013/01/01, que equivale al 1 de enero de 2013, y le indicamos a Python que queremos que nos muestre 6 días. Esto lo hacemos al asignarle un valor a periods. Así que Python nos devuelve un rango de fechas desde el 1 de enero de 2013 hasta el 6 de enero de ese año. Esto se guarda como un tipo de dato que usa Pandas, que contiene un rango de fechas datetime64[ns] y nos dice que la frecuencia está en días, representado por una 'D'.

Luego, creamos un DataFrame con el datetime que creamos antes. Esto lo hacemos primero llamando a la función pd.DataFrame, que crea una estructura de datos similar a una tabla de Excel. Luego, usamos la librería Numpy, específicamente la función random.randn, que genera una matriz de números aleatorios de la distribución normal estándar. El primer número representa la cantidad de filas y el segundo, la cantidad de columnas. Después, asignamos un índice al DataFrame, esto lo hacemos con index=dates, que tomará las fechas que creamos antes y las usará como índice. Si no le pasamos ningún valor para el índice, Pandas utilizará por defecto los números del 0 al infinito. Para asignarle el nombre a las columnas, asignamos una lista con una serie de valores al campo columns.

In [40]:
```
df2 = pd.DataFrame(
    {
        "A": 1.0,
        "B": pd.Timestamp("20130102"),
        "C": pd.Series(1, index=list(range(4)), dtype="float32"),
        "D": np.array([3] * 4, dtype="int32"),
        "E": pd.Categorical(["test", "train", "test", "train"]),
        "F": "foo",
    }
)

df2
```

Out[40]:

	A	B	C	D	E	F
0	1.0	2013-01-02	1.0	3	test	foo
1	1.0	2013-01-02	1.0	3	train	foo
2	1.0	2013-01-02	1.0	3	test	foo
3	1.0	2013-01-02	1.0	3	train	foo

Aquí creamos un dataframe pasando un diccionario donde las llaves son los labels de la columna y los valores son los valores de la columna.

In [41]:
```
df2.dtypes
```

Out[41]:
```
A          float64
B    datetime64[s]
C          float32
D            int32
E         category
F           object
dtype: object
```

Las columnas resultantes del dataframe tienes diferentes Dtypes.

In [52]:
```
df.head()
```

Out[52]:

	A	B	D	C
2013-01-01	-0.503532	-0.583556	-1.448177	-0.002650
2013-01-02	-0.702040	-0.574853	-1.083638	-0.995300
2013-01-03	-1.666242	0.513538	0.292049	-0.797981
2013-01-04	-0.131445	-1.846680	-0.249913	-0.867205
2013-01-05	-1.510444	0.406436	-0.742640	-2.374353

In [51]:
```
df.tail(3)
```

Out[51]:

	A	B	D	C
2013-01-04	-0.131445	-1.846680	-0.249913	-0.867205
2013-01-05	-1.510444	0.406436	-0.742640	-2.374353
2013-01-06	0.851498	-0.718269	-0.786879	-1.089264

Usando las funciones Dataframe.head() y Dataframe.tail() podemos ver las primeras o las últimas filas del dataframe, le podemos indicar cuantas filas podemos ver introduciendo el numero exacto en el parentesis Ej: df.tail(3)

In [53]:
```
df.index
```

Out[53]:
```
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')
```

In [54]:
```
df.columns
```

Out[54]:
```
Index(['A', 'B', 'D', 'C'], dtype='object')
```

Podemos er el indice o las columnas usando las funciones Dataframe.index o Dataframe.columns

In [55]:
```
df.to_numpy()
```

Out[55]:
```
array([[-0.50353212, -0.58355591, -1.44817682, -0.00264965],
       [-0.70203988, -0.57485336, -1.083638  , -0.99530048],
       [-1.66624165,  0.51353844,  0.29204927, -0.79798074],
       [-0.13144458, -1.84668041, -0.24991343, -0.86720493],
       [-1.51044404,  0.40643625, -0.74264031, -2.37435321],
       [ 0.85149776, -0.71826949, -0.78687868, -1.08926414]])
```

Podemos retornar una representacion de la data con Dataframe.to_numpy() pero esto no retornará ni el indice ni los labels de las columnas

In [56]:
```
df.describe()
```

Out[56]:

	A	B	D	C
count	6.000000	6.000000	6.000000	6.000000
mean	-0.610367	-0.467231	-0.669866	-1.021126
std	0.928573	0.862116	0.616053	0.767765
min	-1.666242	-1.846680	-1.448177	-2.374353
25%	-1.308343	-0.684591	-1.009448	-1.065773
50%	-0.602786	-0.579205	-0.764759	-0.931253
75%	-0.224466	0.161114	-0.373095	-0.815287
max	0.851498	0.513538	0.292049	-0.002650

describe() nos enseña un pequeño resumen estadistico de nuestro dataframe.

In [57]:
```
df.T
```

Out[57]:

	2013-01-01	2013-01-02	2013-01-03	2013-01-04	2013-01-05	2013-01-06
A	-0.503532	-0.702040	-1.666242	-0.131445	-1.510444	0.851498
B	-0.583556	-0.574853	0.513538	-1.846680	0.406436	-0.718269
D	-1.448177	-1.083638	0.292049	-0.249913	-0.742640	-0.786879
C	-0.002650	-0.995300	-0.797981	-0.867205	-2.374353	-1.089264

Para transponer la data (Que las filas sean columnas y viceversa) se usa df.T

In [58]:
```
df.sort_index(axis=1,ascending=False)
```

Out[58]:

	D	C	B	A
2013-01-01	-1.448177	-0.002650	-0.583556	-0.503532
2013-01-02	-1.083638	-0.995300	-0.574853	-0.702040
2013-01-03	0.292049	-0.797981	0.513538	-1.666242
2013-01-04	-0.249913	-0.867205	-1.846680	-0.131445
2013-01-05	-0.742640	-2.374353	0.406436	-1.510444
2013-01-06	-0.786879	-1.089264	-0.718269	0.851498

Con df.sort_index puedes ordenar tu dataframe por indices. El axis puede tomar el valor de 1 o de 0, 1 para las columnas y 0 para las filas. y podemos decir si queremos que lo ordene de manera ascendente o descendente con ascending. Entonces lo que le estamos diciendo es que nos ordene las columnas del indice de manera descendente.

In [59]:
```
df.sort_values(by="B")
```

Out[59]:

	A	B	D	C
2013-01-04	-0.131445	-1.846680	-0.249913	-0.867205
2013-01-06	0.851498	-0.718269	-0.786879	-1.089264
2013-01-01	-0.503532	-0.583556	-1.448177	-0.002650
2013-01-02	-0.702040	-0.574853	-1.083638	-0.995300
2013-01-05	-1.510444	0.406436	-0.742640	-2.374353
2013-01-03	-1.666242	0.513538	0.292049	-0.797981

Esta función permite ordenar las filas del DataFrame en función de los valores de una columna específica. aquí le estamos indicando que queremos ordenar los valores del dataframe según la columna B, esta se organniza de manera ascendente por default.

In [60]:
```
df["A"]
```

Out[60]:
```
2013-01-01   -0.503532
2013-01-02   -0.702040
2013-01-03   -1.666242
2013-01-04   -0.131445
2013-01-05   -1.510444
2013-01-06    0.851498
Freq: D, Name: A, dtype: float64
```

Para seleccionar algun elemtento específico utilizamos los corchetes [], solo pasandole un valor de devuelve toda la columna.

In [61]:
```
df[0:3]
```

Out[61]:

	A	B	D	C
2013-01-01	-0.503532	-0.583556	-1.448177	-0.002650
2013-01-02	-0.702040	-0.574853	-1.083638	-0.995300
2013-01-03	-1.666242	0.513538	0.292049	-0.797981

Para seleccionar un slice lo podemos hacer con los dos puntos :, esto nos devolvera un rango del Dataframe. En este caso nos devuelve las 3 primeras filas del dataframe.

In [64]:
```
df["20130102":"20130104"]
```

Out[64]:

	A	B	D	C
2013-01-02	-0.702040	-0.574853	-1.083638	-0.995300
2013-01-03	-1.666242	0.513538	0.292049	-0.797981
2013-01-04	-0.131445	-1.846680	-0.249913	-0.867205

Aquí le específicamos un rango de valores que queremos que nos devuelva.

In [65]:
```
df.loc[dates[0]]
```

Out[65]:
```
A   -0.503532
B   -0.583556
D   -1.448177
C   -0.002650
Name: 2013-01-01 00:00:00, dtype: float64
```

Loc es una funcion que nos permite seleccionar datos utilizando las etiquetas, en este caso, los indices de las filas.

In [66]:
```
df.loc[:, ["A","B"]]
```

Out[66]:

	A	B
2013-01-01	-0.503532	-0.583556
2013-01-02	-0.702040	-0.574853
2013-01-03	-1.666242	0.513538
2013-01-04	-0.131445	-1.846680
2013-01-05	-1.510444	0.406436
2013-01-06	0.851498	-0.718269

Aquí seleccionamos todas las filas de las columnas A y B.

In [67]:
```
df.loc["20130102":"20130104", ["A","B"]]
```

Out[67]:

	A	B
2013-01-02	-0.702040	-0.574853
2013-01-03	-1.666242	0.513538
2013-01-04	-0.131445	-1.846680

Aquí le decimos cuales filas y cuales columnas queremos que nos pase, las filas van primero, las columnas después.

In [68]:
```
df.loc[dates[0], "A"]
```

Out[68]:
```
-0.5035321178325527
```

Seleccionar solo una fila y una columna nos retorna un escalar

In [69]:
```
df.at[dates[0], "A"]
```

Out[69]:
```
-0.5035321178325527
```

Este es otro metodo mas rapido para acceder a un escalar

In [70]:
```
df.iloc[3]
```

Out[70]:
```
A   -0.131445
B   -1.846680
D   -0.249913
C   -0.867205
Name: 2013-01-04 00:00:00, dtype: float64
```

Esta función se utiliza para seleccionar datos de un DataFrame basándose en la posición numérica de las filas y columnas. Es decir, en lugar de basarse en etiquetas, utiliza el número de la fila o de la columna. En este caso, estamos visualizando la fila 4 de todas las columnas.

In [71]:
```
df.iloc[3:5, 0:2]
```

Out[71]:

	A	B
2013-01-04	-0.131445	-1.846680
2013-01-05	-1.510444	0.406436

Aquí seleccionamos un slice de las filas 3 hasta la 4 y las columas 1 y 2.

In [72]:
```
df.iloc[[1,2,4], [0,2]]
```

Out[72]:

	A	D
2013-01-02	-0.702040	-1.083638
2013-01-03	-1.666242	0.292049
2013-01-05	-1.510444	-0.742640

Seleccionamos una serie de filas puntuales basándonos en su posición numérica

In [73]:
```
df.iloc[1:3,:]
```

Out[73]:

	A	B	D	C
2013-01-02	-0.702040	-0.574853	-1.083638	-0.995300
2013-01-03	-1.666242	0.513538	0.292049	-0.797981

Esto es para hacer un slice a las filas solamente

In [74]:
```
df.iloc[:, 1:3]
```

Out[74]:

	B	D
2013-01-01	-0.583556	-1.448177
2013-01-02	-0.574853	-1.083638
2013-01-03	0.513538	0.292049
2013-01-04	-1.846680	-0.249913
2013-01-05	0.406436	-0.742640
2013-01-06	-0.718269	-0.786879

Esto es para hacer un slice a las columnas solamente

In [75]:
```
df.iloc[1,1]
```

Out[75]:
```
-0.5748533631886706
```

Obtener un valor en específico

In [76]:
```
df.iat[1,1]
```

Out[76]:
```
-0.5748533631886706
```

Esto es lo mismo pero mas rápido

In [77]:
```
df[df["A"] > 0]
```

Out[77]:

	A	B	D	C
2013-01-06	0.851498	-0.718269	-0.786879	-1.089264

Esto es para seleccionar cuando se comple una condición, en este caso, seleciona los valores de columna A donde los valores sean

mayores a 0

In [78]:
```
df[df > 0]
```

Out[78]:

	A	B	D	C
2013-01-01	NaN	NaN	NaN	NaN
2013-01-02	NaN	NaN	NaN	NaN
2013-01-03	NaN	0.513538	0.292049	NaN
2013-01-04	NaN	NaN	NaN	NaN
2013-01-05	NaN	0.406436	NaN	NaN
2013-01-06	0.851498	NaN	NaN	NaN

Esto es para seleccionar los valores del dataframe que son mayores a 0

In [80]:
```
df2 = df.copy()
df2["E"] = ["one", "one", "two", "three", "four", "three"]
df2
```

Out[80]:

	A	B	D	C	E
2013-01-01	-0.503532	-0.583556	-1.448177	-0.002650	one
2013-01-02	-0.702040	-0.574853	-1.083638	-0.995300	one
2013-01-03	-1.666242	0.513538	0.292049	-0.797981	two
2013-01-04	-0.131445	-1.846680	-0.249913	-0.867205	three
2013-01-05	-1.510444	0.406436	-0.742640	-2.374353	four
2013-01-06	0.851498	-0.718269	-0.786879	-1.089264	three

In [81]:
```
df2[df2["E"].isin(["two","four"])]
```

Out[81]:

	A	B	D	C	E
2013-01-03	-1.666242	0.513538	0.292049	-0.797981	two
2013-01-05	-1.510444	0.406436	-0.742640	-2.374353	four

Esto se usa para filtra valores, en este caso le decimos que nos pase los valores que son iguales a two y a four en la columna e

In [82]:
```
s1 = pd.Series([1,2,3,4,5,6], index=pd.date_range("20130102", periods = 6))
s1
```

Out[82]:
```
2013-01-02    1
2013-01-03    2
2013-01-04    3
2013-01-05    4
2013-01-06    5
2013-01-07    6
Freq: D, dtype: int64
```

In [85]:
```
df["F"] = s1
```

Out[85]:
```
2013-01-02    1
2013-01-03    2
2013-01-04    3
2013-01-05    4
2013-01-06    5
2013-01-07    6
Freq: D, dtype: int64
```

Al configurar una nueva columna, los datos se alinean automáticamente por los índices

In [89]:
```
df.at[dates[0],"A"] = 0
```

Esto usa la función at para asignar un nuevo valor. estamos asignando el valor 0 a la celda correspondiente a la primera fecha de dates.

In [90]:
```
df.iat[0, 1] = 0
```

Esto usa la función at para asignar un nuevo valor. En este caso, por posición en vez de por labels.

In [92]:
```
df.loc[:, "D"] = np.array([5] * len(df))
```

Estamos asignando valores a una columna completa de un dataframe usando un array de NumPy. Primero, seleccionamos todas las

filas de la columna D; luego, creamos una lista con el número 5 repetido tantas veces como fila tenga el dataframe.

In [94]:
```
df
```

Out[94]:

	A	B	D	C	F
2013-01-01	0.000000	0.000000	5.0	-0.002650	NaN
2013-01-02	-0.702040	-0.574853	5.0	-0.995300	1.0
2013-01-03	-1.666242	0.513538	5.0	-0.797981	2.0
2013-01-04	-0.131445	-1.846680	5.0	-0.867205	3.0
2013-01-05	-1.510444	0.406436	5.0	-2.374353	4.0
2013-01-06	0.851498	-0.718269	5.0	-1.089264	5.0

Esto es el resultado de todas las operaciones pasadas

In [97]:
```
df2 = df.copy()
df2[df2 > 0] = -df2
df2
```

Out[97]:

	A	B	D	C	F
2013-01-01	0.000000	0.000000	-5.0	-0.002650	NaN
2013-01-02	-0.702040	-0.574853	-5.0	-0.995300	-1.0
2013-01-03	-1.666242	-0.513538	-5.0	-0.797981	-2.0
2013-01-04	-0.131445	-1.846680	-5.0	-0.867205	-3.0
2013-01-05	-1.510444	-0.406436	-5.0	-2.374353	-4.0
2013-01-06	-0.851498	-0.718269	-5.0	-1.089264	-5.0

Lo que hacemos aquí es crear una copia del dataframe original, luego seleccionamos los valores que sean mayores a 0 y les asignamos valores negativos a todas las celdas seleccionadas.

In [99]:
```
df1 = df.reindex(index=dates[0:4], columns=list(df.columns) + ["E"])
df1.loc[dates[0]: dates[1],"E"] = 1
df1
```

Out[99]:

	A	B	D	C	F	E
2013-01-01	0.000000	0.000000	5.0	-0.002650	NaN	1.0
2013-01-02	-0.702040	-0.574853	5.0	-0.995300	1.0	1.0
2013-01-03	-1.666242	0.513538	5.0	-0.797981	2.0	NaN
2013-01-04	-0.131445	-1.846680	5.0	-0.867205	3.0	NaN

Reaindexar te permite cambiar, añadir o borrar el índice en un eje específico; esto devuelve una copia de la data.
index=dates[0:4]: Esto nos dice que se está creando un nuevo DataFrame solo con las primeras cuatro fechas de dates.
columns=list(df.columns) + ["E"]: Aquí le indicamos que el nuevo DataFrame tendrá las mismas columnas que el DataFrame original, pero le estamos agregando una nueva columna E.

In [100…
```
df1.dropna(how="any")
```

Out[100…

	A	B	D	C	F	E
2013-01-02	-0.70204	-0.574853	5.0	-0.9953	1.0	1.0

Esta función te devuelve todas las filas en donde no hay ningún valor faltante

In [101…
```
df1.fillna(value=5)
```

Out[101…

	A	B	D	C	F	E
2013-01-01	0.000000	0.000000	5.0	-0.002650	5.0	1.0
2013-01-02	-0.702040	-0.574853	5.0	-0.995300	1.0	1.0
2013-01-03	-1.666242	0.513538	5.0	-0.797981	2.0	5.0
2013-01-04	-0.131445	-1.846680	5.0	-0.867205	3.0	5.0

Esta función llena los valores faltantes con el valor que tu le indiques

In [102…
```
pd.isna(df1)
```

Out[102…

	A	B	D	C	F	E
2013-01-01	False	False	False	False	True	False
2013-01-02	False	False	False	False	False	False
2013-01-03	False	False	False	False	False	True
2013-01-04	False	False	False	False	False	True

Esta función hace que los valores faltantes sean TRUE.

In [104…
```
df.mean()
```

Out[104…
```
A   -0.526445
B   -0.369971
D    5.000000
C   -1.021126
F    3.000000
dtype: float64
```

Calcula la media de cada columna

In [105…
```
df.mean(axis=1)
```

Out[105…
```
2013-01-01    1.249338
2013-01-02    0.745561
2013-01-03    1.009863
2013-01-04    1.030934
2013-01-05    1.104328
2013-01-06    1.808793
Freq: D, dtype: float64
```

Calcula la media de cada fila

In [109…
```
s = pd.Series([1,3,5,np.nan,6,8], index=dates).shift(2)
s
```

Out[109…
```
2013-01-01    NaN
2013-01-02    NaN
2013-01-03    1.0
2013-01-04    3.0
2013-01-05    5.0
2013-01-06    NaN
Freq: D, dtype: float64
```

In [110…
```
df.sub(s,axis="index")
```

Out[110…

	A	B	D	C	F
2013-01-01	NaN	NaN	NaN	NaN	NaN
2013-01-02	NaN	NaN	NaN	NaN	NaN
2013-01-03	-2.666242	-0.486462	4.0	-1.797981	1.0
2013-01-04	-3.131445	-4.846680	2.0	-3.867205	0.0
2013-01-05	-6.510444	-4.593564	0.0	-7.374353	-1.0
2013-01-06	NaN	NaN	NaN	NaN	NaN

Al trabajar con otra Serie o DataFrame que tenga un índice o columnas diferentes, el resultado se alineará con la unión de las etiquetas de índice o columnas. Además, pandas automáticamente hace un broadcast a lo largo de la dimensión especificada y llenará las etiquetas desalineadas con np.nan.

In [111…
```
df.agg(lambda x: np.mean(x) *5.6)
```

Out[111…
```
A    -2.948094
B    -2.071840
D    28.000000
C    -5.718303
F    16.800000
dtype: float64
```

la función agg (Aggregate) se usa para aplicar una función a lo largo de un eje (una fila o una columna), en este caso la columna y luego se usa una función anónima que toma como entrada una columna, le saca la media y la multiplica por 5.6

In [112…
```
df.transform(lambda x: x * 101.2)
```

Out[112…

	A	B	D	C	F
2013-01-01	0.000000	0.000000	506.0	-0.268145	NaN
2013-01-02	-71.046436	-58.175160	506.0	-100.724409	101.2
2013-01-03	-168.623655	51.970090	506.0	-80.755651	202.4
2013-01-04	-13.302192	-186.884058	506.0	-87.761139	303.6
2013-01-05	-152.856937	41.131348	506.0	-240.284545	404.8
2013-01-06	86.171573	-72.688873	506.0	-110.233531	506.0

transform, a diferencia de agg, aplica la operación a cada valor del DataFrame, devolviendo un DataFrame con la misma cantidad de filas y columnas.

Estas funciones permiten aplicar funciones definidas por el usuario a un dataframe.

In [114…
```
s = pd.Series(np.random.randint(0,7, size = 10))
s
```

Out[114…
```
0    5
1    5
2    4
3    4
4    2
5    1
6    2
7    6
8    3
9    4
dtype: int32
```

aquí creamos una serie de 10 numeros aleatorios que pueden ir del 0 al 6.

In [115…
```
s.value_counts()
```

Out[115…
```
4    3
5    2
2    2
1    1
6    1
3    1
Name: count, dtype: int64
```

y luego con la función value_counts() contamos cuántas veces aparece cada valor en la serie. Básicamente te dice la frecuencia de cada número en la serie.

In [118…
```
s = pd.Series(["A", "B", "C", "Aaba", "Baca", np.nan, "CABA", "dog", "cat"])
s.str.lower()
```

Out[118…
```
0       a
1       b
2       c
3    aaba
4    baca
5     NaN
6    caba
7     dog
8     cat
dtype: object
```

Primero creamos una serie, y luego usamos str.lower; str se refiere a que estamos accediendo a una operación de cadenas y lower convierte todas las letras a minúsculas.

In [122…
```
df = pd.DataFrame(np.random.randn(10,4))
df
```

Out[122…

	0	1	2	3
0	0.308345	2.853698	0.781155	-0.235751
1	-1.430324	1.096822	0.046209	-1.142979
2	-0.958521	-0.464481	-1.558605	-1.280579
3	1.617494	-1.917454	1.954722	-0.800582
4	0.450336	0.315228	-0.831033	-1.767681
5	1.266892	1.466659	-0.536196	-2.145111
6	1.368807	-0.462822	0.709767	0.451165
7	0.918029	1.204791	0.778312	1.698545
8	0.524341	0.341980	0.280387	1.254091
9	0.472502	-0.266787	-0.815563	1.810458

Creamos un Dataframe con numeros aleatorios de 10 filas y 4 columnas

In [128…

```
#break it into pieces
pieces = [df[:3], df[3:7], df[7:]]
pd.concat(pieces)
```

Out[128…

	0	1	2	3
0	0.308345	2.853698	0.781155	-0.235751
1	-1.430324	1.096822	0.046209	-1.142979
2	-0.958521	-0.464481	-1.558605	-1.280579
3	1.617494	-1.917454	1.954722	-0.800582
4	0.450336	0.315228	-0.831033	-1.767681
5	1.266892	1.466659	-0.536196	-2.145111
6	1.368807	-0.462822	0.709767	0.451165
7	0.918029	1.204791	0.778312	1.698545
8	0.524341	0.341980	0.280387	1.254091
9	0.472502	-0.266787	-0.815563	1.810458

Estamos usando la funcion concat para concatenar varias partes de un Dataframe. Primero dividimos el Dataframe en piezas df[3] selecciona las 3 primeras filas del dataframe df[3:7] selecciona las filas desde la posición 3 hasta la 6 df[7:] selecciona las filas de la posicion 7 en adelante y esto se guarda en una lista que se llama pieces

Despues usamos la funcion concat para concatenar la lista por filas, creando un solo dataframe. pandas usa axis= 0 por defecto, osea que por defecto concatena por filas

In [135…

```
left = pd.DataFrame({"key": ["foo", "foo"], "lval": [1, 2]})
right = pd.DataFrame({"key": ["foo", "foo"], "rval": [4, 5]})
left
```

Out[135…

	key	lval
0	foo	1
1	foo	2

Creamos dos dataframes

In [134…

```
right
```

Out[134…

	key	rval
0	foo	4
1	foo	5

In [133…

```
pd.merge(left, right, on="key")
```

Out[133…

	key	lval	rval
0	foo	1	4
1	foo	1	5
2	foo	2	4
3	foo	2	5

pd.merge realiza una operacion de join entre dos dataframes en función de una columna compratida. left y right son los dataframes que queremos unir, y on=key es la columna que ambos dataframes tienen en común y será utilizada como clave para hacer el join

In [137…
```
left = pd.DataFrame({"key": ["foo", "bar"], "lval": [1, 2]})
right = pd.DataFrame({"key": ["foo", "bar"], "rval": [4, 5]})
left
```

Out[137…

	key	lval
0	foo	1
1	bar	2

In [ ]:
```
creamos dos dataframes
```

In [138…
```
right
```

Out[138…

	key	rval
0	foo	4
1	bar	5

In [139…
```
pd.merge(left, right, on="key")
```

Out[139…

	key	lval	rval
0	foo	1	4
1	bar	2	5

left y right son los dataframes que deseamos unir.
on=key es la columna común que ambos dataframes tienen, y que será utilizada para emparejar las filas.
el resultado es un dataframe donde se combinan las columnas de ambos dataframes y dado que las claves son únicas, el resultado es una unión uno a uno.

In [141…
```
df = pd.DataFrame(
    {
        "A": ["foo", "bar", "foo", "bar", "foo", "bar", "foo", "foo"],
        "B": ["one", "one", "two", "three", "two", "two", "one", "three"],
        "C": np.random.randn(8),
        "D": np.random.randn(8),
    }
)

df
```

Out[141…

	A	B	C	D
0	foo	one	-0.714348	0.420894
1	bar	one	0.072580	1.123379
2	foo	two	0.680969	0.171687
3	bar	three	-1.213341	-0.006287
4	foo	two	-0.147130	-2.988119
5	bar	two	0.856072	-0.765795
6	foo	one	1.546776	-0.849379
7	foo	three	1.962669	-0.037483

Creamos un dataframe

In [142…
```
df.groupby("A")[["C", "D"]].sum()
```

Out[142…

A	C	D
bar	-0.284689	0.351297
foo	3.328937	-3.282400

df.groupby("A"): esto agrupa los valores de la columna A
[["C", "D"]]: luego de agrupar seleccionamos las columnas C y D para aplicar la operación. Básicamente solo se sumarán los valores en estas dos columnas dentro de cada grupo de A.
finalmente la función sum() se aplica a los dos grupos, esto suma los valores de las columnas C y D para cada valor de A.

In [143…
```
df.groupby(["A", "B"]).sum()
```

Out[143…

A	B	C	D
bar	one	0.072580	1.123379
	three	-1.213341	-0.006287
	two	0.856072	-0.765795
foo	one	0.832429	-0.428485
	three	1.962669	-0.037483
	two	0.533840	-2.816432

Aquí estamos agrupando datos por múltiples columnas, lo que genera un MultiIndex, seguido de una función para sumar los valores en las columnas seleccionadas. Agrupar por múltiples columnas significa que los datos se agrupan primero por los valores de la columna A, y luego, dentro de esos grupos, se agrupan por los valores de la columna B. El resultado es un datagrama con un multiindex formado por las columnas A y B, y con las sumas correspondientes en las columnas C y D.

In [146…
```
arrays = [
   ["bar", "bar", "baz", "baz", "foo", "foo", "qux", "qux"],
   ["one", "two", "one", "two", "one", "two", "one", "two"],
]
index = pd.MultiIndex.from_arrays(arrays, names=["first", "second"])
df = pd.DataFrame(np.random.randn(8, 2), index=index, columns=["A", "B"])
df2 = df[:4]
df2
```

Out[146…

first	second	A	B
bar	one	2.710851	1.065379
	two	0.689675	0.316865
baz	one	-0.489989	0.081920
	two	1.762749	0.974972

pd.MultiIndex.from_arrays(): Esta función convierte las listas en un MultiIndex, es decir, un índice que tiene varios niveles. names=["first", "second"]: Aquí se asignan nombres a los dos niveles del índice. El primer nivel se llamará "first" y el segundo nivel se llamará "second". El resultado es un MultiIndex con dos niveles, donde el primer nivel es "bar", "baz", "foo", "qux", y dentro de cada uno hay un subnivel con "one" y "two"

In [147…
```
stacked = df2.stack(future_stack=True)
stacked
```

Out[147…
```
first  second
bar    one     A    2.710851
                B    1.065379
       two     A    0.689675
                B    0.316865
baz    one     A   -0.489989
                B    0.081920
       two     A    1.762749
                B    0.974972
dtype: float64
```

df2.stack(): El método stack() toma las columnas del DataFrame df2 y las mueve a un nivel más bajo del índice (es decir, convierte las columnas en parte del índice). Esto transforma las columnas en un nivel jerárquico del índice Las columnas han desaparecido y el contenido ahora está representado como valores con tres niveles de índice. Cada combinación única de los niveles del índice tiene un valor numérico asociado, que antes pertenecía a la columna correspondiente ("A" o "B").

In [149…
```
stacked.unstack()
stacked.unstack(1)
stacked.unstack(0)
```

Out[149…

		first	bar	baz
second				
one	A	2.710851	-0.489989	
	B	1.065379	0.081920	
two	A	0.689675	1.762749	
	B	0.316865	0.974972	

El método unstack() toma un nivel del índice de un DataFrame (o Series) que tiene un MultiIndex y lo convierte en columnas, deshaciendo el proceso de apilamiento hecho por stack().

In [150…
```
df = pd.DataFrame(
    {
        "A": ["one", "one", "two", "three"] * 3,
        "B": ["A", "B", "C"] * 4,
        "C": ["foo", "foo", "foo", "bar", "bar", "bar"] * 2,
        "D": np.random.randn(12),
        "E": np.random.randn(12),
    }
)
df
```

Out[150…

	A	B	C	D	E
0	one	A	foo	0.115141	1.283074
1	one	B	foo	1.034008	0.829312
2	two	C	foo	-0.044649	0.368180
3	three	A	bar	-1.597521	-0.133375
4	one	B	bar	1.415118	-0.506155
5	one	C	bar	-0.683194	-0.900029
6	two	A	foo	1.641915	-1.086659
7	three	B	foo	-0.848818	-0.625513
8	one	C	foo	0.964477	0.453435
9	one	A	bar	-0.566575	1.367182
10	two	B	bar	-0.776374	2.268069
11	three	C	bar	-1.211132	1.089595

Creamos un dataframe

In [151…
```
pd.pivot_table(df, values="D", index=["A", "B"], columns=["C"])
```

Out[151…

	C	bar	foo
A	B		
one	A	-0.566575	0.115141
	B	1.415118	1.034008
	C	-0.683194	0.964477
three	A	-1.597521	NaN
	B	NaN	-0.848818
	C	-1.211132	NaN
two	A	NaN	1.641915
	B	-0.776374	NaN
	C	NaN	-0.044649

pd.pivot_table(): Este es el método que crea una tabla dinámica, te permite reorganizar los datos de un DataFrame en base a ciertas columnas que se usarán como índice, columnas y valores. values="D": Indica que los datos de la columna D son los que se utilizarán para rellenar las celdas de la tabla dinámica. index=["A", "B"]: Definimos A y B como el índice de la tabla dinámica, o sea, que A y B serán las filas en la tabla resultante. columns=["C"]: Aquí especificamos que la columna C se usará para crear las columnas en la tabla resultante. Los valores únicos en la columna C serán las cabeceras de las columnas de la tabla dinámica.

In [152…
```
rng = pd.date_range("1/1/2012", periods=100, freq="s")
ts = pd.Series(np.random.randint(0, 500, len(rng)), index=rng)
ts.resample("5Min").sum()
```

Out[152…
```
2012-01-01    24689
Freq: 5min, dtype: int32
```

pd.date_range(): Esta función crea una serie de fechas con una frecuencia específica.
np.random.randint(0, 500, len(rng)): Esto genera números enteros aleatorios entre 0 y 500. La cantidad de números aleatorios generados es igual al número de fechas.
pd.Series(): Esto crea una serie.
ts.resample("5Min"): Esta función es un remuestreo que agrupa los datos en intervalos de tiempo más grandes, en este caso, el argumento 5Min indica que estamos reagrupando los datos en intervalos de 5 minutos.
Después de remuestrear los datos en intervalos de 5 minutos, sumamos los valores en cada intervalo.

El resultado de esta operación es una nueva serie donde los datos originales (que estaban en intervalos de 1 segundo) han sido agrupados en intervalos de 5 minutos, y los valores dentro de cada intervalo se han sumado.

In [153…
```
rng = pd.date_range("3/6/2012 00:00", periods=5, freq="D")

ts = pd.Series(np.random.randn(len(rng)), rng)

ts
```

Out[153…
```
2012-03-06   -0.509421
2012-03-07    0.367999
2012-03-08    0.311466
2012-03-09   -0.465065
2012-03-10   -0.792779
Freq: D, dtype: float64
```

In [154…
```
ts_utc = ts.tz_localize("UTC")

ts_utc
```

Out[154…
```
2012-03-06 00:00:00+00:00   -0.509421
2012-03-07 00:00:00+00:00    0.367999
2012-03-08 00:00:00+00:00    0.311466
2012-03-09 00:00:00+00:00   -0.465065
2012-03-10 00:00:00+00:00   -0.792779
Freq: D, dtype: float64
```

Aquí estamos usando la función tz_localize() para asignar una zona horaria a una serie temporal.
ts.tz_localize("UTC"): Esta función asigna una zona horaria a una serie temporal que no tiene una asignada previamente.
"UTC": Aquí, estamos asignando la zona horaria UTC (Coordinated Universal Time) a la serie ts.

In [155…
```
ts_utc.tz_convert("US/Eastern")
```

Out[155…
```
2012-03-05 19:00:00-05:00   -0.509421
2012-03-06 19:00:00-05:00    0.367999
2012-03-07 19:00:00-05:00    0.311466
2012-03-08 19:00:00-05:00   -0.465065
2012-03-09 19:00:00-05:00   -0.792779
Freq: D, dtype: float64
```

Aquí usamos la función tz_convert() para convertir una serie temporal con una zona horaria a otra zona horaria diferente.
tz_convert("US/Eastern"): Esta función convierte la serie temporal de la zona horaria actual (UTC) a la zona horaria US/Eastern.

In [156…
```
rng
rng + pd.offsets.BusinessDay(5)
```

Out[156…
```
DatetimeIndex(['2012-03-13', '2012-03-14', '2012-03-15', '2012-03-16',
               '2012-03-16'],
              dtype='datetime64[ns]', freq=None)
```

Aquí estamos usando BusinessDay una utilidad para trabajar con series temporales que consideren solo días hábiles, es decir, excluyendo los fines de semana y días festivos. pd.offsets.BusinessDay(5): Esto crea un desplazamiento de 5 días hábiles. Los días hábiles son generalmente de lunes a viernes, por lo que los fines de semana se saltan. rng + pd.offsets.BusinessDay(5): Aquí, sumamos 5 días hábiles a cada una de las fechas en el rango rng. Como el desplazamiento es de días hábiles, las fechas se ajustan, excluyendo los fines de semana.

In [157…
```
df = pd.DataFrame(
    {"id": [1, 2, 3, 4, 5, 6], "raw_grade": ["a", "b", "b", "a", "a", "e"]}
)
```

Creamos un dataframe

In [158…
```
df["grade"] = df["raw_grade"].astype("category")
```

```
df["grade"]
```

Out[158…
```
0    a
1    b
2    b
3    a
4    a
5    e
Name: grade, dtype: category
Categories (3, object): ['a', 'b', 'e']
```

Estamos convirtiendo una serie de datos string a un tipo de datos muy útil llamado category, que es más eficiente para manejar variables categóricas.
df["raw_grade"]: Aquí estamos accediendo a la columna que queremos cambiar a category.
astype("category"): Este método convierte los datos de la columna a un tipo de datos categórico.
df["grade"] = Simplemente estamos creando una nueva columna llamada grade que contendrá los datos convertidos a categóricos.

In [159…
```
new_categories = ["very good", "good", "very bad"]

df["grade"] = df["grade"].cat.rename_categories(new_categories)
```

Estamos renombrando las categorías para que tengan nombres más significativos. Primero definimos los nuevos nombres de las categorías con new_categories = ["very good", "good", "very bad"] Luego las renombramos: df["grade"] selecciona la columna grade.
.cat.rename_categories(new_categories): .cat: Accede a las propiedades categóricas de la columna.
.rename_categories(new_categories): Cambia los nombres de las categorías actuales a los que has definido en la lista new_categories..

In [161…
```
df["grade"] = df["grade"].cat.set_categories(
    ["very bad", "bad", "medium", "good", "very good"]
)
df["grade"]
```

Out[161…
```
0    very good
1         good
2         good
3    very good
4    very good
5     very bad
Name: grade, dtype: category
Categories (5, object): ['very bad', 'bad', 'medium', 'good', 'very good']
```

Estamos usando la función set_categories() para redefinir las categorías de la columna grade. df["grade"]: Selecciona la columna
.cat.set_categories(): .cat: Accedes a las propiedades categóricas de la columna. set_categories(): Esta función permite redefinir el conjunto de categorías que puede tener la columna.

Ahora las categorías de la columna grade tienen 5 niveles.

In [162…
```
df.sort_values(by="grade")
```

Out[162…

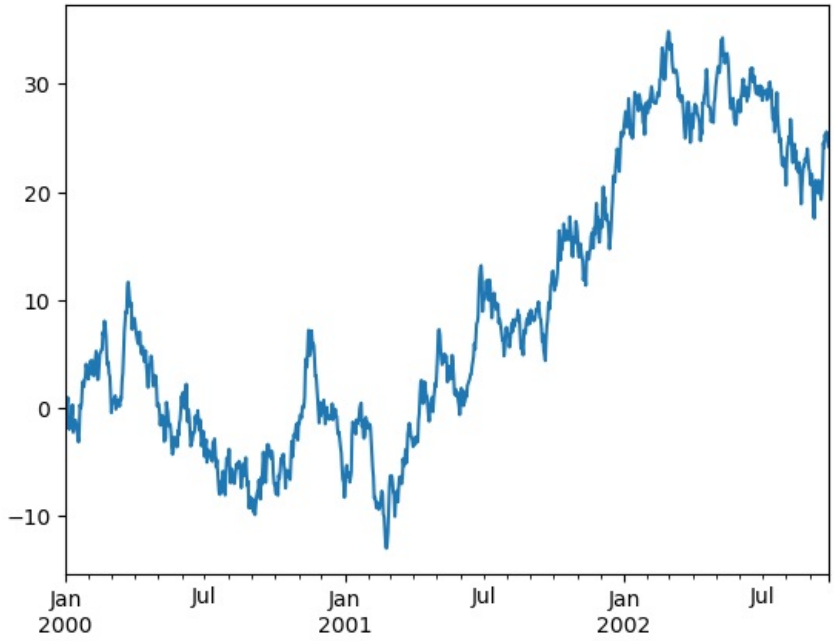
	id	raw_grade	grade
5	6	e	very bad
1	2	b	good
2	3	b	good
0	1	a	very good
3	4	a	very good
4	5	a	very good

Estamos ordenando las categorías pero no por el orden léxico, sino por el orden de las categorías. df.sort_values(): Esta es la función que se utiliza para ordenar un DataFrame en función de una o más columna by="grade": Especifica que deseamos ordenar el DataFrame en función de la columna grade.

In [165…
```
import matplotlib.pyplot as plt

plt.close("all")
```

Aquí importamos la librería Matplotlib plt.close("all"): Esta función cierra todas las figuras abiertas o activas en Matplotlib.

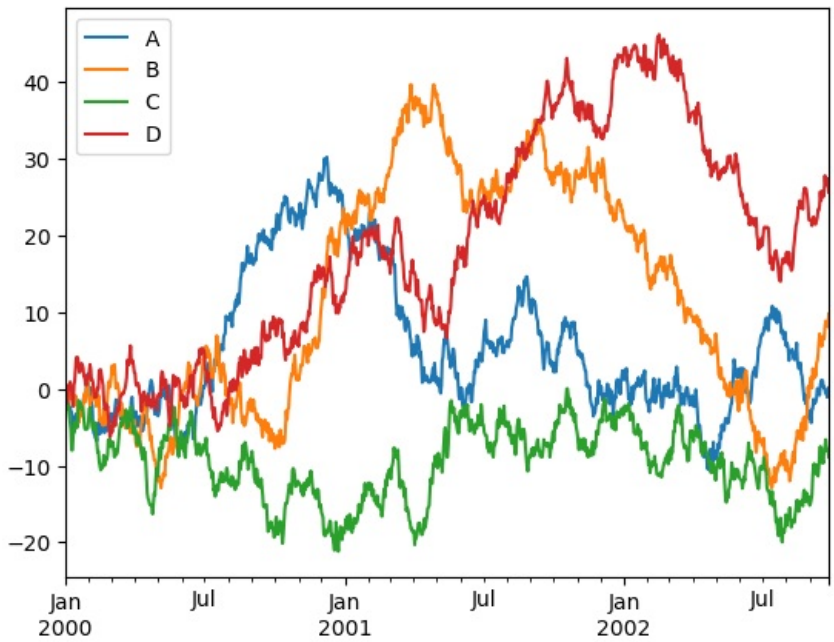
In [166…
```
ts = pd.Series(np.random.randn(1000), index=pd.date_range("1/1/2000", periods=1000))

ts = ts.cumsum()

ts.plot();
```



Primero creamos una serie temporal aleatoria y luego creamos un gráfico con Matplotlib para vizualizar los datos. .cumsum(): Este metodo calcula la suma acumulada de los valores en la serie. la suma acumulada es que cada valor en la serie es la suma del valor actual mas todos los valores anteriores. ts.plot(); Esta función crea un gráfico de líneas de la serie temporal utilizando Matplotlib de manera interna, ya que Pandas tiene una integración con Matplotlib.

In [167…
```
df = pd.DataFrame(
    np.random.randn(1000, 4), index=ts.index, columns=["A", "B", "C", "D"]
)
df = df.cumsum()

plt.figure();

df.plot();

plt.legend(loc='best');
```

```
<Figure size 640x480 with 0 Axes>
```



df = df.cumsum() .cumsum(): Calculamos la suma acumulada de los valores en cada columna del DataFrame. Esto significa que cada valor es la suma del valor actual y todos los valores anteriores en la misma columna.

plt.figure(); Crea una nueva figura en Matplotlib. Es útil para crear un espacio separado para un gráfico y permite personalizarlo antes de trazar los datos.

df.plot(); Esta función de Pandas crea automáticamente un gráfico de líneas para todas las columnas del DataFrame. Cada columna se dibuja como una línea separada en el gráfico.

plt.legend(loc="best"): Añade una leyenda al gráfico que indica a qué columna corresponde cada línea.

El argumento loc="best" le indica a Matplotlib que coloque la leyenda en la mejor ubicación automática.

In [168…
```
df = pd.DataFrame(np.random.randint(0, 5, (10, 5)))

df.to_csv("foo.csv")
```

Aquí usamos la funcion to_csv para escribir un dataframe en un archivo csv

In [170…
```
pd.read_csv("foo.csv")
```

Out[170…

	Unnamed: 0	0	1	2	3	4
0	0	4	2	0	3	2
1	1	0	4	3	0	1
2	2	1	0	0	3	1
3	3	3	1	0	4	1
4	4	2	1	2	3	1
5	5	1	3	0	3	1
6	6	0	1	1	2	4
7	7	0	2	0	1	1
8	8	4	2	3	3	0
9	9	1	4	2	2	4

In [ ]:
```
aquí usamos la función read_csv para leer el archivo creado anteriormente
```

In [171…
```
df.to_parquet("foo.parquet")
```

In [ ]:
```
Aquí usamos la funcion to_parquet para escribir un dataframe en un archivo Parquet
```

In [172…
```
pd.read_parquet("foo.parquet")
```

Out[172…

	0	1	2	3	4
0	4	2	0	3	2
1	0	4	3	0	1
2	1	0	0	3	1
3	3	1	0	4	1
4	2	1	2	3	1
5	1	3	0	3	1
6	0	1	1	2	4
7	0	2	0	1	1
8	4	2	3	3	0
9	1	4	2	2	4

In [ ]:
```
aquí usamos la función read_parquet para leer el archivo creado anteriormente
```

In [173…
```
df.to_excel("foo.xlsx", sheet_name="Sheet1")
```

In [ ]:
```
Aquí usamos la funcion to_excel para escribir un dataframe en un archivo Excel
```

In [174…
```
pd.read_excel("foo.xlsx", "Sheet1", index_col=None, na_values=["NA"])
```

Out[174…

Unnamed: 0	0	1	2	3	4
0	0	4	2	0	3	2
1	1	0	4	3	0	1
2	2	1	0	0	3	1
3	3	3	1	0	4	1
4	4	2	1	2	3	1
5	5	1	3	0	3	1
6	6	0	1	1	2	4
7	7	0	2	0	1	1
8	8	4	2	3	3	0
9	9	1	4	2	2	4

"Sheet1": Especifica la hoja de cálculo dentro del archivo de Excel que deseas leer.

index_col=None:Este argumento indica que no se debe usar ninguna columna como índice. Pandas, por defecto, intentará usar la primera columna como índice, pero al establecer index_col=None, le estás diciendo que trate todas las columnas como datos y que cree un índice numérico por defecto.

na_values=["NA"]: Este argumento le dice a Pandas qué valores debe interpretar como valores nulos o NaN. En este caso, cualquier valor en el archivo Excel que sea "NA" se interpretará como un valor faltante.