11_Series___indices_y_operaciones

November 28, 2023

0.1 Operando con Series

0.1.1 Operar con datos en Pandas

Una de las piezas esenciales de NumPy es la capacidad de realizar operaciones rápidas entre elementos, tanto con aritmética básica (suma, resta, multiplicación, etc.) como con operaciones más sofisticadas (funciones trigonométricas, exponenciales y logarítmicas, etc.). **Y Pandas hereda gran parte de esta funcionalidad de NumPy**

Pero lo importante es que las operaciones unitarias sobre objetos Pandas *preservarán las etiquetas de índice y columna* en la salida, y para las operaciones binarias como **la adición y la multiplicación, Pandas *alineará automáticamente los índices* al pasar los objetos a la ufunc.** Esto significa que mantener el contexto de los datos y combinar datos de diferentes fuentes -ambas tareas potencialmente propensas a errores con arrays NumPy en bruto- se convierten en tareas esencialmente infalibles con Pandas. Además, veremos que hay operaciones bien definidas entre estructuras unidimensionales `Series` y estructuras bidimensionales `DataFrame`.

0.2 Operaciones con Series

Se peuden emplear muchas funciones de numpay pero preservando indice y columna, pudiendo jugar con los resultados de las operaciones con pandas

0.3 Introducción

Como Pandas está diseñado para trabajar con NumPy, casi cualquier función de NumPy funcionará con los objetos `Series` y `DataFrame` de Pandas. Empecemos definiendo una simple `Serie` y `DataFrame` en la que demostrar esto:

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

np.random.seed(42)# establecemos un semilla , para que nos permita tener sempre␣
 ↪las mismas series random
```

[5]:
```
ser = pd.Series(np.random.randint(0,10,4))# hemos creado una serei panda␣
 ↪aleatoria unidimensional, entre el 0 y  10 y formada por 4 elementos
ser
```

[5]:
```
0    6
1    9
2    2
3    6
dtype: int32
```

[8]:
```
df = pd.DataFrame(np.random.randint(0,10,(3,4)), columns = ["A", "B", "C",␣
 ↪"D"])# creamos un DF con valores random entre 0 a 10m esta vec formando un␣
 ↪amatriz de 4 filas y 3 columnas,
# asignado a las columnas sus nombres (tb podrian haber asignado nombres a las␣
 ↪filas)
df
```

[8]:

	A	B	C	D
0	7	2	5	4
1	1	7	5	1
2	4	0	9	5

[ ]:
```

```

Si aplicamos funciones unitarias de NumPy tanto para `Series` como para `DataFrame`, el resultado será otro objeto Pandas *con los índices conservados:*

[10]:
```
np.sqrt(ser)# aqui calculamos la raiz cuadrada de toda las serie ser, operando␣
 ↪elemento a elemento
```

[10]:
```
0    2.449490
1    3.000000
2    1.414214
3    2.449490
dtype: float64
```

[14]:
```
np.sqrt(df)# tb se le puede aplicar al dataframe
```

[14]:

	A	B	C	D
0	2.645751	1.414214	2.236068	2.000000
1	1.000000	2.645751	2.236068	1.000000
2	2.000000	0.000000	3.000000	2.236068

[16]:
```
new_df = np.sqrt(df)
new_df
```

[16]:

	A	B	C	D
0	2.645751	1.414214	2.236068	2.000000
1	1.000000	2.645751	2.236068	1.000000
2	2.000000	0.000000	3.000000	2.236068

O, para un cálculo un poco más complejo:

[17]:
```
np. cos(df * np.pi/4)# realziamos el coseno del angulo al df, pero antes lo␣
 ↪hemos multiplicado por pi y divido por ya, cada elemento del DF
```

[17]:

	A	B	C	D
0	0.707107	6.123234e-17	-0.707107	-1.000000
1	0.707107	7.071068e-01	-0.707107	0.707107
2	-1.000000	1.000000e+00	0.707107	-0.707107

Algunos ejemplos de funciones numpy que puedes aplicar a las series: * np.add() -> suma las series que pases por argumento (aunque es más rápido usar el operador +, o el metodo ad) * np.subtract() -> resta las series que pases por argumento o un valor a todos los elementos de la serie * np.divide() -> Divide una serie entre otra (elemento a elemento) o cada elemento por un número

[18]:
```
np.add(ser,ser)# se sumara ser a si mismo
```

[18]:
```
0    12
1    18
2     4
3    12
dtype: int32
```

[19]:
```
ser + ser
```

[19]:
```
0    12
1    18
2     4
3    12
dtype: int32
```

[20]:
```
ser-ser
```

[20]:
```
0    0
1    0
2    0
3    0
dtype: int32
```

Las anteriores son más sencillas usando operadores, pero estás ya no tienen operador: * np.exp(), calcula e^x para cada elemento x de la serie * np.sin(), calcula el seno de cada elemento * np.log(), calcula el logaritmo

[21]:
```
np.log(ser)
```

[21]:
```
0    1.791759
1    2.197225
2    0.693147
3    1.791759
dtype: float64
```

0.4 Alineado de índices en Operaciones

Para operaciones binarias sobre dos objetos `Series` o `DataFrame`, Pandas alineará los índices en el proceso de realización de la operación.

Esto es muy conveniente cuando se trabaja con datos incompletos, como veremos en algunos de los ejemplos que siguen.

0.4.1 Alineado de índices en Series

Como ejemplo, supongamos que combinamos dos fuentes de datos diferentes y encontramos sólo los tres primeros estados de EE.UU. por *área* y los tres primeros estados de EE.UU. por *población*:

[22]:
```
area = pd.Series({'Alaska': 1723337,
                  'Texas': 695662,
                  'California': 423967},
                 name='superficie')

poblacion = pd.Series({'California': 38332521,
                       'Texas': 26448193,
                       'New York': 19651127},
                      name='poblacion')
```

Veamos qué ocurre cuando los dividimos para calcular la densidad de población:

[23]:
```
s_poblaccion_area = poblacion/area# calculamos la densidad de poblacion
s_poblaccion_area
# al no poder tener valores para realziar los calculos rellena los huecos NaN(␣
 ↪datos nulo o faltantes)
```

[23]:
```
Alaska              NaN
California    90.413926
New York            NaN
Texas         38.018740
dtype: float64
```

[ ]:
```

```

[ ]:
```

```

Cualquier elemento para el que uno u otro no tenga una entrada se marca con `NaN`, o "Not a Number", que es como Pandas marca los datos que faltan.[Y que mencionamos en alguna pildora

anterior, ojo como puedes ver empiezan a ser omnipresentes, y lo seguirán siendo los NaN o nulos] Esta coincidencia de índices se implementa de esta manera para cualquiera de las expresiones aritméticas incorporadas de Python; cualquier valor que falte se rellena con NaN por defecto:

[28]:
```
A = pd.Series([2,4,6], index = ["andalucia", "aragon", "madrid"])# series␣
 ↪indexadas
B = pd.Series([1,3,5], index = ["aragon", "madrid","asturias"])
A + B
```

[28]:
```
andalucia    NaN
aragon       5.0
asturias     NaN
madrid       9.0
dtype: float64
```

¿Y como se tratan los NaN posteriormente en otras operaciones? Pues hagamos un ejemplo...

[30]:
```
serie_1 = A + B
serie_2 = serie_1 +B
serie_2# si operamos con NaN, sieguen dando NaN, por lo que lo que se debe␣
 ↪hacer es eliminar los NaN o limpiar los NaN danodle algun valor
```

[30]:
```
andalucia     NaN
aragon        6.0
asturias      NaN
madrid       12.0
dtype: float64
```

Si el uso de valores NaN no es el comportamiento deseado, el valor de llenado puede ser modificado usando métodos de objetos apropiados en lugar de los operadores. Por ejemplo, llamar a `A.add(B)` es equivalente a llamar a `A + B`, pero permite la especificación explícita opcional del valor de relleno para cualquier elemento de `A` o `B` que pueda faltar:

[31]:
```
B.add(A, fill_value= 0)# con fill_value le hemos dado valores a los NaN, por lo␣
 ↪que realzia todas las operaciones y nos modifica el indice con valores de␣
 ↪menor a mayor
```

[31]:
```
andalucia    2.0
aragon       5.0
asturias     5.0
madrid       9.0
dtype: float64
```

[33]:
```
A
```

[33]:
```
andalucia    2
aragon       4
madrid       6
dtype: int64
```

[34]:
```
B
```

[34]:
```
aragon      1
madrid      3
asturias    5
dtype: int64
```

Y ahora ya sí que podriamos seguir operando:

[ ]:

[ ]: