



BASECAMP

Ciencia de Datos

Obtención y Preparación de Datos

Objetivo de la jornada

- Aplica técnicas de unión, combinación y redimensionamiento de estructuras de datos utilizando librerías de Python para el reacomodo de datos.

Indexación Jerárquica

La indexación jerárquica / multinivel es muy interesante, ya que abre la puerta a un análisis y manipulación de datos bastante sofisticados, especialmente para trabajar con datos de dimensiones superiores. En esencia, le permite almacenar y manipular datos con un número arbitrario de dimensiones en estructuras de datos de menor dimensión como Series(1d) y DataFrame(2d).

Crear objeto multi-índices

Podemos crear un multi-índice de cuatro formas distintas:

- A partir de una lista de arrays, usando el método `pd.MultiIndex.from_arrays()`
- A partir de un array de tuplas, usando el método `pd.MultiIndex.from_tuples()`
- A partir del producto cartesiano de los valores de dos iterables, usando el método `pd.MultiIndex.from_product()`
- A partir de un DataFrame, usando el método `pd.MultiIndex.from_frame()`

Veamos un ejemplo de cada tipo.

Multi-índice a partir de una lista de array

El primer método es aquel en el que creamos el multi-índice indicando cada una de las columnas que lo van a formar. Por ejemplo:

```
index = pd.MultiIndex.from_arrays(  
    [[2018, 2018, 2018, 2019, 2019],  
     ["Spain", "Portugal", "France", "Spain",  
      "Portugal", "France"]],  
    names = ["Year", "Country"])
```

index

```
MultiIndex([(2018, 'Spain'),  
            (2018, 'Portugal'),  
            (2018, 'France'),  
            (2019, 'Spain'),  
            (2019, 'Portugal'),  
            (2019, 'France')],  
           names=['Year', 'Country'])
```

Como vemos, el parámetro names nos permite especificar los nombres de los niveles del índice jerárquico.

Si llevamos este multi-índice a un dataframe obtenemos el siguiente resultado:

```
data = pd.DataFrame(data = [18, 20, 10, 15, 12, 18],  
                    index = index, columns = ["Sales"])
```

data

Sales		
Year	Country	
2018	Spain	18
	Portugal	20
	France	10
2019	Spain	15
	Portugal	12
	France	18

Multi-índice a partir de un array de tuplas

En este segundo método indicamos los valores del multi-índice valor por valor, siendo éstos tuplas:

```
index = pd.MultiIndex.from_tuples([
    (2018, "Spain"),
    (2018, "Portugal"),
    (2018, "France"),
    (2019, "Spain"),
    (2019, "Portugal"),
    (2019, "France")], names =
    ["year", "Country"])
```

index

```
MultiIndex([(2018, 'Spain'),
            (2018, 'Portugal'),
            (2018, 'France'),
            (2019, 'Spain'),
            (2019, 'Portugal'),
            (2019, 'France')],
            names=['Year', 'Country'])
```

Seguimos teniendo a nuestra disposición el parámetro **names** para especificar los nombres de los niveles.

Si creamos nuestro **DataFrame** vemos que el resultado es el mismo que el que habíamos obtenido:

```
df = pd.DataFrame(data = [18, 20, 10, 15, 12, 18],  
index = index, columns = ["sales"])
```

data

		Sales
Year	Country	
2018	Spain	18
	Portugal	20
	France	10
2019	Spain	15
	Portugal	12
	France	18

Multi-índice por productos cartesiano de array

El tercer método nos permite especificar los valores (únicos) de los diferentes niveles, creándose el índice jerárquico como resultado del producto escalar de los valores. Por ejemplo:

```
index = pd.MultiIndex.from_product([  
    [2018, 2019],  
    ["Spain", "Portugal", "France"],  
    names = ["Year", "Country"]])
```

index

```
MultiIndex([(2018, 'Spain'),  
            (2018, 'Portugal'),  
            (2018, 'France'),  
            (2019, 'Spain'),  
            (2019, 'Portugal'),  
            (2019, 'France')],  
           names=['Year', 'Country'])
```

Nuevamente, el parámetro **names** nos permite dar nombre a los niveles.

El resultado de llevar este índice a nuestro **DataFrame** es el ya conocido:

```
data = pd.DataFrame(data = [18, 20, 10, 15, 12, 18],  
                    index = index, columns = ["Sales"])
```

data

		Sales
Year	Country	
2018	Spain	18
	Portugal	20
	France	10
2019	Spain	15
	Portugal	12
	France	18

Multi-índice a partir de un DataFrame

Por último, podemos crear el multi-índice a partir de un DataFrame en el que cada columna coincide con una columna del multi-índice. Comencemos creando el DataFrame:

```
df = pd.DataFrame({  
    "Year": [2018, 2018, 2019, 2019, 2019],  
    "Country": ["Spain", "Portugal", "France",  
                "Spain", "Portugal", "France"]})  
  
df
```

	Year	Country
0	2018	Spain
1	2018	Portugal
2	2018	France
3	2019	Spain
4	2019	Portugal
5	2019	France

Ahora podemos crear el índice:

```
index = pd.MultiIndex.from_frame(df)  
index
```

```
MultiIndex([(2018,    'Spain'),  
            (2018, 'Portugal'),  
            (2018,    'France'),  
            (2019,    'Spain'),  
            (2019, 'Portugal'),  
            (2019,    'France')],  
           names=['Year', 'Country'])
```

...y nuestro DataFrame con índice jerárquico:

```
data = pd.DataFrame(data = [18, 20, 10, 15, 12, 18],  
index, columns = ["Sales"])  
data
```

		Sales
Year	Country	
2018	Spain	18
	Portugal	20
	France	10
2019	Spain	15
	Portugal	12
	France	18

Reordenamiento de niveles

Ejemplo:

Se tienen dos matrices de entrada **arr** y entero **N**.

Por lo tanto, el valor del elemento en la *enésima* posición es la posición donde estará en la matriz ordenada, y debe devolver la matriz.

El número entero es más pequeño que el valor del elemento en la *enésima* posición antes de que se mueva, y todos los números enteros que son más grandes que este último. Cuatro ejemplos son los siguientes:

```
1  
2 Reordenar ([6,5,8,1,7,2,9,3,4], 2) == [1,2,3,6,7,5,9,8,4]  
3 Reordenar ([7,3,9,6,2,5,1,8,4], 5) == [3,2,1,4,5,6,7,8,9]  
4 Reordenar ([2,9,1,5,7,3,6,4,8], 5) == [2,1,3,4,5,6,7,9,8]  
5 Reordenar ([6,2,4,9,1,3,7,8,5], 5) == [2,1,4,3,5,6,7,8,9]  
6
```


Debe utilizar las funciones de NumPy para completar este ejercicio.

```
def reorder(arr, n):  
    pass
```

Prueba:

```
Test.describe('Basic tests')  
  
Test.assert_equals(reorder([6, 5, 8, 1, 7, 2, 9, 3, 4],2), [1,2, , , 7, 5, 9, 8, 4])  
  
Test.assert_equals(reorder([7, 3, 9, 6, 2, 5, 1, 8, 4],5), [1,2, , , 7, 5, 9, 8, 4])  
  
Test.assert_equals(reorder([2, 9, 1, 5, 7, 3, 6, 4, 8],5), [1,2, , , 7, 5, 9, 8, 4])  
  
Test.assert_equals(reorder([6, 2, 4, 9, 1, 3, 7, 8, 5],5), [1,2, , , 7, 5, 9, 8, 4])
```

Posibles Respuestas.

Respuesta 1:

```
def re_ordering(name):  
    return ' '.join(sorted(name.split(), key =  
str.islower))
```

Respuesta 2:

```
def re_ordering(text):  
    return ' '.join(sorted(name.split(), key =  
str.islower))
```

Respuesta 3:

```
import re  
def re_ordering(text):  
    x, y = re.search(r'[A-Z][A-Za-z]*',  
test).span()  
    return (text[x:y] + ' ' + text[:x] +  
text[y:]).rstrip().replace(' ', ' ')
```

Respuesta 4:

```
def re_ordering(text)  
    words = text.split(" ")  
    for i in range(len(words)):  
        if words[i][0].isupper():
```

```
        words.insert(0, words.pop(i),)
        break
    return " ".join(words)
```

Columnas como Índices

Normalmente, en un Pandas **DataFrame**, tenemos números de serie desde 0 hasta la longitud del objeto como índice por defecto. También podemos hacer que una columna específica de un **dataframe** sea su índice. Para ello, podemos usar el **set_index()** proporcionado en pandas, y también podemos especificar el índice de la columna mientras importamos un **dataframe** de un archivo Excel o CSV.

Usando **set_index** para hacer columnas como índices.

set_index() puede aplicarse a listas, series o cuadros de datos para alterar su índice. Para los **Dataframes**, **set_index()** también puede hacer múltiples columnas como su índice.

Ejemplo:

```
import pandas as pd
import numpy as np

colnames = ['Name', 'Time', 'Course']
df = pd.DataFrame([[ 'Jay', 10, 'B.Tech'],
                   [ 'Raj', 12, 'B.Tech'],
                   [ 'Jack', 11, 'B.Sc']], columns = colnames)
print(df)
```

Resultado:

	Name	Time	Course
0	Jay	10	B.Tech
1	Raj	12	BBA
2	Jack	11	B.Sc

La sintaxis para hacer columnas como índice:

```
dataframe.set_index(Column_name, inplace = True)
```

Hacer una sola columna como índice usando **set_index()**:

```
import pandas as pd
import numpy as np

colnames = ['Name', 'Time', 'Course']
df = pd.DataFrame([['Jay', 10, 'B.Tech'],
                  ['Raj', 12, 'B.Tech'],
                  ['Jack', 11, 'B.Sc']], columns = colnames)

df.set_index('Name', inplace = True)
print(df)
```

Resultado

	Time	Course
Name		
Jay	10	B.Tech
Raj	12	BBA
Jack	11	B.Sc

Hacer varias columnas como índice:

```
import pandas as pd
import numpy as np

colnames = ['Name', 'Time', 'Course']
df = pd.DataFrame([['Jay', 10, 'B.Tech'],
                   ['Raj', 12, 'B.Tech'],
                   ['Jack', 11, 'B.Sc']], columns = colnames)

df.set_index(['Name', 'Course'], inplace = True)
print(df)
```

Resultado:

		Time
Name	Course	
Jay	B.Tech	10
Raj	BBA	12
Jack	B.Sc	11

Usando el parámetro **index_col** en **read_excel** o **read_csv** para establecer la columna como índice.

Mientras leemos un **dataframe** de un archivo **Excel** o **CSV**, podemos especificar la columna que queremos como el índice del **DataFrame**.

Ejemplo:

```
import pandas as pd
import numpy as np

df = pd.read_excel("data.xlsx", index_col = 2)
print(df)
```

Resultado:

	Name	Time
Course		
B.Tech	Mark	12
BBA	Jack	10
B.Sc	Jay	11

Combinación y merge de Datos.

Ésta es otra de las áreas en las que la variedad de opciones puede resultar confusa. A modo de resumen, digamos que pandas ofrece dos principales funciones con este objetivo: **pandas.concat** y **pandas.merge**.

- La función **concat** permite concatenar dataframes a lo largo de un determinado eje.
- La función **merge** permite realizar uniones (*joins*) entre dataframes tal y como se realizan en bases de datos. Esta función también está disponible como método: **pandas.DataFrame.merge**

Hay una tercera función que está disponible solo como método: **pandas.DataFrame.append**. El método **append** ofrece una funcionalidad semejante a la de la función **concat** pero reducida. Así, por ejemplo, sólo permite realizar concatenaciones a lo largo del eje 0 (es decir, verticalmente).

La función concat

La función **pandas.concat** es la responsable de concatenar dos o más **dataframes** (y de todas las estructuras proveídas por pandas) a lo largo de un eje, con soporte a lógica de conjuntos a la hora de gestionar etiquetas en ejes no coincidentes. Veamos un primer caso, el más sencillo posible, para el que partimos de los siguientes dos **dataframes**:

```
df1 = pd.DataFrame(np.arange(9).reshape([3, 3]),  
                    index = ["a", "b", "d"],  
                    columns = ["A", "B", "C"])  
df1
```

	A	B	C
a	0	1	2
b	3	4	5
d	6	7	8


```
df2 = pd.DataFrame(np.arange(12).reshape([4, 3]),  
                    index = ["a", "b", "c", "e"],  
                    columns = ["B", "C", "D"])  
df2
```

	B	C	D
a	0	1	2
b	3	4	5
c	6	7	8
e	9	10	11

En algunos de los ejemplos que se muestran a continuación se utiliza el argumento `sort = False` para evitar que se muestre un aviso al respecto de cierto cambio de funcionalidad que se producirá en futuras versiones de esta librería.

Si pasamos a la función `concat` ambos dataframes como primer argumento (en forma de lista), obtenemos el siguiente resultado:

```
pd.concat([df1, df2], sort = False)
```




	A	B	C	D
a	0.0	1	2	NaN
b	3.0	4	5	NaN
d	6.0	7	8	NaN
a	NaN	0	1	2.0
b	NaN	3	4	5.0
c	NaN	6	7	8.0
e	NaN	9	10	11.0

Vemos cómo, por defecto, la concatenación se ha realizado a lo largo del eje 0 (eje vertical), uniendo los índices de fila de ambos **dataframes**, y alineando las columnas por su etiqueta. Los valores para los que no hay datos se han rellenado con NaN (opción correspondiente al argumento por defecto join: "outer").

Si especificamos que la concatenación se realice a lo largo del eje 1 (eje horizontal), el resultado es el siguiente

```
pd.concat([df1, df2], axis = 1, sort = False)
```



	A	B	C	B	C	D
a	0.0	1.0	2.0	0.0	1.0	2.0
b	3.0	4.0	5.0	3.0	4.0	5.0
d	6.0	7.0	8.0	NaN	NaN	NaN
c	NaN	NaN	NaN	6.0	7.0	8.0
e	NaN	NaN	NaN	9.0	10.0	11.0

De modo semejante al primer ejemplo, se han introducido NaN's allí donde no había datos, y se han alineado las filas por su etiqueta.

Estos dos ejemplos vistos son tipo "Outer" (opción por defecto), considerando todas las etiquetas de los dos dataframes aun cuando no sean comunes a ambos. Pero si especificamos el argumento `join = "Inner"`, los resultados pasan a considerar sólo las etiquetas comunes. Así, para el primer ejemplo visto tenemos:

```
pd.concat([df1, df2], join = "inner")
```

	B	C
a	1	2
b	4	5
d	7	8
a	0	1
b	3	4
c	6	7
e	9	10

...incluyendo solo las columnas B y C comunes a ambos dataframes. Y para el segundo ejemplo tenemos:

```
pd.concat([df1, df2], axis = 1, join = "inner")
```

	A	B	C	B	C	D
a	0	1	2	0	1	2
b	3	4	5	3	4	5

..Incluyendo solo las filas a y b comunes a ambos dataframes.

El parámetro **ignore_index** controla el índice a asignar al eje a lo largo del cual se realiza la concatenación. Si este parámetro toma el valor `False` (por defecto), el eje de concatenación mantiene las etiquetas de los dataframes originales. Si toma el valor `True`, se ignoran dichas

etiquetas y el resultado de la concatenación recibe un nuevo índice automático numérico. Por ejemplo, si añadimos a ln [176] el argumento **ignore_index=True**, obtenemos el siguiente resultado:

```
pd.concat([df1,df2], axis = 1, join = "inner",
ignore_index = True)
```

```

   0  1  2  3  4  5
a  0  1  2  0  1  2
b  3  4  5  3  4  5
```

El método append

El método `pandas.DataFrame.append` es un atajo de la función `concat` que ofrece funcionalidad semejante pero limitada: no permite especificar el eje de concatenación (siempre es el eje 0) ni el tipo de "join" (siempre es tipo "Outer").

Si seguimos con los mismos **dataframes** de anteriores:

```
df1 = pd.DataFrame(np.arange(9).reshape([3, 3]),
                    index = ["a", "b", "d"],
                    columns = ["A", "B", "C"])
df2 = pd.DataFrame(np.arange(12).reshape([4, 3]),
                    index = ["a", "b", "c", "e"],
                    columns = ["B", "C", "D"])
print(df1)
print(df2)
```

	A	B	C		B	C	D
a	0	1	2	a	0	1	2
b	3	4	5	b	3	4	5
d	6	7	8	c	6	7	8
				e	9	10	11

...podemos ver cuál es el resultado de aplicar este método a df1:

```
df1.append(df2, sort = False)
```

	A	B	C	D
a	0.0	1	2	NaN
b	3.0	4	5	NaN
d	6.0	7	8	NaN
a	NaN	0	1	2.0
b	NaN	3	4	5.0
c	NaN	6	7	8.0
e	NaN	9	10	11.0

Al igual que ocurría con la función **concat**, el parámetro **ignore_index** nos permite controlar las etiquetas que recibe el índice del resultado: las de los dataframes originales (con *ignore_index = False*, opción por defecto), o uno nuevo automático (con *ignore_index = True*).

La función merge

La función **pandas.merge** nos permite realizar "joins" entre tablas. El join es realizado sobre las columnas o sobre las filas. En el primer caso, las etiquetas de las filas son ignoradas. En cualquier otro caso (joins realizados entre etiquetas de filas, o entre etiquetas de filas y de columnas), las etiquetas de filas se mantienen.

Veamos un primer ejemplo. Partimos de dos tablas conteniendo las ventas y costes de producción para varios meses:

```
df1 = pd.DataFrame({
    "Month": ["ene", "feb", "mar", "may"],
    "Sales": [14, 8, 12, 17]})
df1
```

	Month	Sales
0	ene	14
1	feb	8
2	mar	12
3	may	17

```
df2 = pd.DataFrame({
    "Month": ["ene", "feb", "mar", "abr"],
    "Sales": [7, 6, 8, 5]})
df2
```

	Month	Cost
0	feb	7
1	ene	6
2	mar	8
3	abr	5

Vemos que ambos dataframes tienen una columna común ("Month") y varias filas comunes ("ene", "feb" y "mar"). Obsérvese que en df2 las filas no están ordenadas y que, en df1, el mes de enero tiene índice 0 mientras que, en df2, el mes de enero tiene índice 1.

Si aplicamos la función merge a estos dataframes con los valores por defecto, obtenemos el siguiente resultado:

```
pd.merge(df1, df2)
```

	Month	Sales	Cost
0	ene	14	6
1	feb	8	7
2	mar	12	8

Esos valores por defecto suponen que el join se realiza sobre las columnas comunes y tipo "inner" (considerando solo las filas con etiquetas comunes).

Si especificamos que el join sea de tipo "outer", lo que definimos con el parámetro how, el resultado considerará todas las etiquetas presentes en ambos dataframes:

```
pd.merge(df1, df2, how = "outer")
```

	Month	Sales	Cost
0	ene	14.0	6.0
1	feb	8.0	7.0
2	mar	12.0	8.0
3	may	17.0	NaN
4	abr	NaN	5.0

Como vemos, se ha rellenado con NaN's los valores inexistentes. Otras opciones para el parámetro how son "left" y "right" (además de la opción por defecto, "outer").

Ya se ha comentado que, por defecto, el join se realiza entre las columnas comunes. Esto es, sin embargo, controlable usando el parámetro on y especificando la columna o columnas a usar. Por ejemplo, consideremos los siguientes dataframes:

```
df1 = pd.DataFrame({
    "Month": ["ene", "feb", "mar", "may"],
    "Product": ["A","B","A","B"],
    "Sales": [14, 8 12, 17]})
df1
```

```
Month  Product  Sales
0     ene      A     14
1     ene      B      8
2     feb      A     12
3     feb      B     17
```

```
df2 = pd.DataFrame({
    "Month": ["ene", "ene", "feb", "feb"],
    "Product": ["A","B","A","B"],
    "Sales": [7, 6, 8, 5]})
df2
```

```
Month  Product  Cost
0     ene      A      7
1     ene      B      6
2     feb      A      8
3     feb      B      5
```

Hay dos columnas comunes, lo que supone que el resultado de un merge por defecto sería el siguiente:

```
pd.merge(df1, df2)
```

Out[10]:

	Month	Product	Sales	Cost
0	ene	A	14	7
1	ene	B	8	6
2	feb	A	12	8
3	feb	B	17	5

Es decir, para cada combinación de Mes-Producto se añadirían los valores de los campos de ventas y coste. Si quisiéramos que el join se realizase solo por uno de los campos, *Product*, por ejemplo, bastaría con especificarlo con el parámetro *on*:

```
pd.merge(df1, df2, on = "Product")
```

Out[11]:

	Month_x	Product	Sales	Month_y	Cost
0	ene	A	14	ene	7
1	ene	A	14	feb	8
2	feb	A	12	ene	7
3	feb	A	12	feb	8
4	ene	B	8	ene	6
5	ene	B	8	feb	5
6	feb	B	17	ene	6
7	feb	B	17	feb	5

Además del campo utilizado para realizar el join ("Product"), al existir un campo común a ambos dataframes ("Month") que no se desea usar para el join, pandas añade un sufijo (configurable) a este campo en ambas tablas para poder diferenciarlo.

También podría ocurrir que ambos dataframes no tuviesen columnas comunes (columnas con el mismo nombre) pero que, aun así, quisiéramos realizar el join por algunas de ellas. Por ejemplo:

```
df1 = pd.DataFrame({
    "Month": ["ene", "feb", "mar", "may"],
    "Sales": [14, 8, 12, 17]})
df2 = pd.DataFrame({
    "MonthName": ["feb", "ene", "mar", "abr"],
    "Cost": [7, 6, 8, 5]})
```

Al no haber columnas comunes, la ejecución de la función *merge* devolvería un error. En este caso podemos usar los parámetros **left_on** y **right_on** para especificar el campo a usar en la tabla de la izquierda del join y en la de la derecha, respectivamente:

```
pd.merge(df1, df2, left_on = "Month", right_on = "MonthName")
```

```
Out[105]:
```

	Month	Sales	MonthName	Cost
0	ene	14	ene	6
1	feb	8	feb	7
2	mar	12	mar	8

Vemos cómo se realiza el join correctamente y se mantienen las columnas originales.

Join por índices de filas

Si queremos que el join considere los índices de las filas -y no los valores de las columnas- de alguno de los dataframes para realizar el join, podemos usar los parámetros `left_index` y `right_index`.

Supongamos, por ejemplo, que partimos de los siguientes dataframes:

```
df1 = pd.DataFrame({
    "Month": ["ene", "feb", "mar", "may"],
    "Sales": [14, 8, 12, 17]})
df2 = pd.DataFrame({
    "Purchases": [5, 9, 11, 2, 6]},
    index = ["ene", "feb", "mar", "abr", "may"])

display(df1)
display(df2)
```

			Purchases	
	Month	Sales		
0	ene	14	ene	5
1	feb	8	feb	9
2	mar	12	mar	11
3	may	17	abr	2
			may	6

La ejecución de la función `merge` no sería posible -devolvería un error- pues no hay columnas comunes. En este caso querríamos que para el

dataframe df1 se considerase la columna "Month" -usando el parámetro left_on- y para el dataframe df2 el índice -usando el parámetro right_index-, de la siguiente forma:

```
pd.merge(df1, df2, left_on = "Month", right_index = True)
```

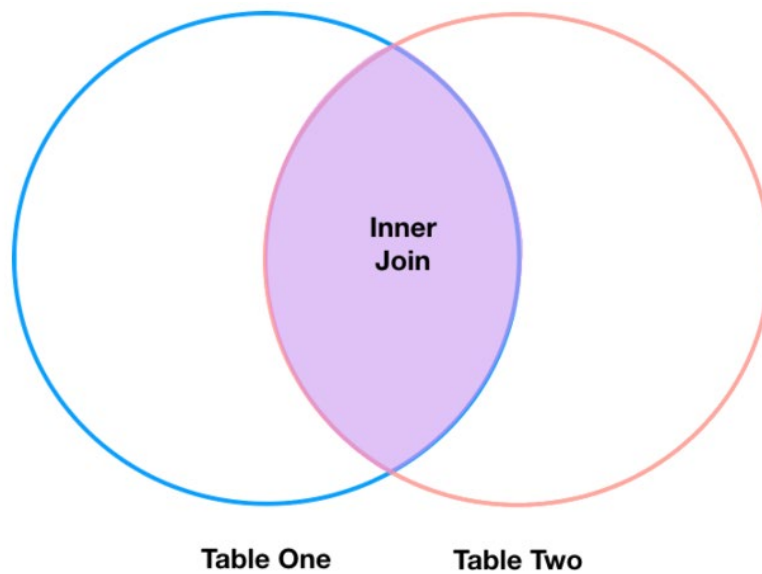
	Month	Sales	Purchases
0	ene	14	5
1	feb	8	9
2	mar	12	11
3	may	17	6

Operaciones Join al estilo de bases de datos.

Uniones Internas (inner join)

El tipo más común de unión se llama **inner join** (unión interna). Una combinación interna combina dos **DataFrames** basados en una clave de unión y devuelve un nuevo **DataFrame** que contiene **solo** aquellas filas que tienen valores coincidentes entre los dos **DataFrames** originales.

Las uniones internas producen un **DataFrame** que contiene solo filas donde el valor que es el sujeto de la unión existe en las dos tablas. Un ejemplo de una unión interna, adaptado de esta página se encuentra a continuación:



La función en pandas para realizar uniones se llama merge y una unión interna es la opción por defecto:

```
merged_inner = pd.merge(left=survey_sub, right=species_sub,  
left_on='species_id', right_on='species_id')
```

#En este caso, 'species_id' es el único nombre de la columna en los dos __DataFrames__, entonces si omitimos los argumentos 'left_on' y 'right_on' todavía obtendríamos el mismo resultado

#¿Cuál es el tamaño de los datos en el resultado?

```
merged_inner.shape
```

```
merged_inner
```

	record_id	month	day	year	plot_id	species_id	sex	hindfoot_length	\
0	1	7	16	1977	2	NL	M	32	
1	2	7	16	1977	3	NL	M	33	
2	3	7	16	1977	2	DM	F	37	
3	4	7	16	1977	7	DM	M	36	
4	5	7	16	1977	3	DM	M	35	
5	8	7	16	1977	1	DM	M	37	
6	9	7	16	1977	1	DM	F	34	
7	7	7	16	1977	2	PE	F	NaN	

	weight	genus	species	taxa
0	NaN	Neotoma	albigula	Rodent
1	NaN	Neotoma	albigula	Rodent
2	NaN	Dipodomys	merriami	Rodent
3	NaN	Dipodomys	merriami	Rodent
4	NaN	Dipodomys	merriami	Rodent
5	NaN	Dipodomys	merriami	Rodent
6	NaN	Dipodomys	merriami	Rodent
7	NaN	Peromyscus	eremicus	Rodent

El resultado de una unión interna de `survey_sub` y `species_sub` es un nuevo DataFrame que contiene el conjunto combinado de columnas de `survey_sub` y `species_sub`. Solo contiene filas que tienen códigos de dos letras de especies que son iguales en el `survey_sub` y el `species_sub` DataFrames. En otras palabras, si una fila en `survey_sub` tiene un valor de `species_id` que no aparece en la `species_id` columna de `species`, no será incluída en el DataFrame devuelto por una unión interna. Del mismo modo, si una fila en `species_sub` tiene un valor de `species_id` que no aparece en la columna `species_id` de `survey_sub`, esa fila no será incluída en el DataFrame devuelto por una unión interna.

Los dos DataFrames a los que queremos unir se pasan a la función `merge` usando el argumento de `left` y `right`. El argumento `left_on = 'species'` le dice a `merge` que use la columna `species_id` como la clave de unión de `survey_sub` (el `left` DataFrame). De manera similar, el argumento `right_on = 'species_id'` le dice a `merge` que use la columna `species_id` como la clave de unión de `species_sub` (el `right` DataFrame). Para uniones internas, el orden de los argumentos `left` y `right` no importa.

El resultado `merged_inner` DataFrame contiene todas las columnas de `survey_sub` (ID de registro, mes, día, etc.), así como todas las columnas de `species_sub` (`species_id`, género, especie y taxa).

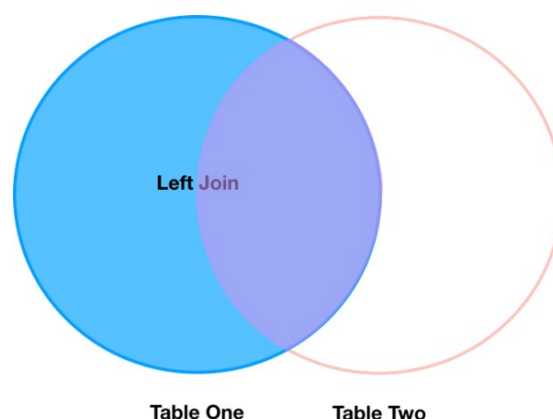
Date cuenta que `merged_inner` tiene menos filas que `survey_sub`. Esto es una indicación de que había filas en `survey_df` con valor(es) para `species_id` que no existen como valor(es) para `species_id` en `species_df`.

Unión Izquierda (left join)

¿Qué pasa si queremos agregar información de `species_sub` a `survey_sub` sin perdiendo información de `survey_sub`? En este caso, utilizamos un diferente tipo de unión llamada “left outer join (unión externa izquierda)”, or a “left join (unión izquierda)”.

Como una combinación interna, una unión izquierda utiliza las claves de unión para combinar dos DataFrames. Diferente a una unión interna, una unión izquierda devolverá todas las filas del left DataFrame, hasta aquellas filas cuyas claves de unión no tienen valores en el right DataFrame. Filas en el left DataFrame que faltan valores para las clave(s) de unión en el right DataFrame simplemente tendrán valores nulos (es decir, NaN o Ninguno) para las columnas en el resultante DataFrame unido.

Una unión izquierda descartará las filas del right DataFrame que no tienen valores para la(s) clave(s) de unión en el left DataFrame.



Una unión izquierda se realiza en pandas llamando a la misma función merge utilizada para unión interna, pero usando el argumento how = 'left':

```
merged_left = pd.merge(left=survey_sub, right=species_sub,
                        how='left', left_on='species_id', right_on='species_id')
merged_left
```

```
**OUTPUT:**

   record_id  month  day  year  plot_id  species_id  sex  hindfoot_length  \
0          1     7   16  1977         2         NL    M             32
1          2     7   16  1977         3         NL    M             33
2          3     7   16  1977         2         DM    F             37
3          4     7   16  1977         7         DM    M             36
4          5     7   16  1977         3         DM    M             35
5          6     7   16  1977         1         PF    M             14
6          7     7   16  1977         2         PE    F             NaN
7          8     7   16  1977         1         DM    M             37
8          9     7   16  1977         1         DM    F             34
9         10     7   16  1977         6         PF    F             20

   weight  genus  species  taxa
0     NaN  Neotoma  albigula  Rodent
1     NaN  Neotoma  albigula  Rodent
2     NaN  Dipodomys  merriami  Rodent
3     NaN  Dipodomys  merriami  Rodent
4     NaN  Dipodomys  merriami  Rodent
5     NaN     NaN     NaN     NaN
6     NaN  Peromyscus  eremicus  Rodent
7     NaN  Dipodomys  merriami  Rodent
8     NaN  Dipodomys  merriami  Rodent
9     NaN     NaN     NaN     NaN
```

El resultado **DataFrame** de una unión izquierda (merged_left) se parece mucho al resultado **DataFrame** de una unión interna (merged_inner) en

términos de las columnas que contiene. Sin embargo, a diferencia de `merged_inner`, `merged_left` contiene el **mismo número de filas** como el **DataFrame** original `survey_sub`. Cuando inspeccionamos `merged_left`, encontramos que hay filas donde la información debería haber venido de `species_sub` (es decir, `species_id`, `genus` y `taxa`) hace falta (contienen valores de `NaN`):

```
merged_left[ pd.isnull(merged_left.genus) ]
**OUTPUT:**
   record_id  month  day  year  plot_id species_id sex  hindfoot_length \
5          6     7   16  1977         1         PF  M              14
9         10     7   16  1977         6         PF  F              20

   weight genus species taxa
5     NaN   NaN    NaN   NaN
9     NaN   NaN    NaN   NaN
```

Estas filas son aquellas en las que el valor de `species_id` de `survey_sub` (en este caso, `PF`) no ocurre en `species_sub`.

Otros tipos de unión

La función `merge` de `pandas` admite otros dos tipos de unión:

- **Right (outer) join** unión derecha (exterior): se invoca al pasar `how = 'right'` como argumento. Similar a una unión izquierda, excepto que se guardan todas las filas del `right DataFrame`, mientras que las filas del `left DataFrame` sin coincidir con los valores de las claves de unión son descartadas.
- **Full (outer) join** unión completa (externa): se invoca al pasar `how = 'outer'` como argumento. Este tipo de unión devuelve todas las combinaciones de filas de los dos `DataFrames`; es decir., el `DataFrame` resultante estará `NaN` donde faltan datos en uno de los `DataFrames`. Este tipo de unión es muy raramente utilizado.

Merge sobre índices.

Cuando se fusionan dos `DataFrames` en el índice, el valor de los parámetros `left_index` y `right_index` de la función `merge()` debe ser `True`.

El siguiente ejemplo de código combinará dos DataFrames con inner como tipo de combinación:

```
import numpy as np

import pandas as pd

df1 = pd.DataFrame(['a','b','d', 'e', 'h'],index = [1, 2, 4, 5, 7], columns = ['C1'])

df2 = pd.DataFrame(['AA','BB','CC', 'EE', 'FF'],index = [1, 2, 3, 5, 6], columns = ['C2'])

df_inner = df1.merge(df2, how = 'inner', left_index = True, right_index = True)

df_inner
```

Resultado:

	C1	C2
1	a	AA
2	b	BB
5	e	EE

El siguiente código combinará los DataFrames con el tipo de unión como outer:


```
import numpy as np

import pandas as pd

df1 = pd.DataFrame(['a','b','d', 'e', 'h'],index = [1, 2, 4, 5, 7], columns = ['C1'])

df2 = pd.DataFrame(['AA','BB','CC', 'EE', 'FF'],index = [1, 2, 3, 5, 6], columns = ['C2'])

df_inner = df1.merge(df2, how = 'outer', left_index = True, right_index = True)

print(df_inner)
```

Resultado:

	C1	C2
1	a	AA
2	b	BB
3	NaN	CC
4	d	NaN
5	e	EE
6	NaN	FF
7	h	NaN

Como puedes ver, el DataFrame fusionado con el tipo join como inner sólo tiene registros coincidentes de ambos DataFrames, mientras que el que tiene el tipo join outer tiene todos los elementos presentes en ellos, llenando los registros que faltan con NaN. Ahora usando left join:

```
import numpy as np
```

```
import pandas as pd

df1 = pd.DataFrame(['a','b','d', 'e', 'h'],index = [1, 2, 4, 5, 7], columns = ['C1'])

df2 = pd.DataFrame(['AA','BB','CC', 'EE', 'FF'],index = [1, 2, 3, 5, 6], columns = ['C2'])

df_inner = df1.merge(df2, how = 'left', left_index = True, right_index = True)

print(df_inner)
```

Resultado:

	C1	C2
1	a	AA
2	b	BB
4	d	NaN
5	e	EE
7	h	NaN

El DataFrame fusionado anterior tiene todos los elementos del DataFrame izquierdo, y sólo los registros coincidentes del DataFrame derecho. Su opuesto exacto es la unión right, como se muestra abajo:

```
import numpy as np

import pandas as pd
```

```
df1 = pd.DataFrame(['a','b','d', 'e', 'h'],index = [1, 2, 4, 5, 7], columns = ['C1'])

df2 = pd.DataFrame(['AA','BB','CC', 'EE', 'FF'],index = [1, 2, 3, 5, 6], columns = ['C2'])

df_inner = df1.merge(df2, how = 'right', left_index = True, right_index = True)

print(df_inner)
```

Resultado:

	C1	C2
1	a	AA
2	b	BB
3	NaN	CC
5	e	EE
6	NaN	FF

Concatenación sobre un eje

Acción: Lectura de dos csv (data.csv y label.csv) en un solo marco de datos.
`import numpy as np`

```
df = dd.read_csv(data_files, delimiter=' ', header=None, names=['x', 'y', 'z', 'intensity', 'r', 'g', 'b'])

df_label = dd.read_csv(label_files, delimiter=' ', header=None, names=['label'])
```

Problema: La concatenación de columnas requiere divisiones conocidas. Sin embargo, establecer un índice ordenará los datos, que explícitamente no quiero, porque el orden de ambos archivos es su coincidencia.

```
df = dd.concat([df, df_label], axis=1)

-----

ValueError                                Traceback (most recent call
last)

<python-input-11-e6c2e1bdde55> in <module>()

----> 1 df = dd.concat([df, df_label], axis=1)

/uhome/hemmet/.local/lib/python3.5/site-
packages/dask/dataframe/multi.py in concat(dfs, axis,
join, interleave_partitions)

    573             return
concat_unindexed_dataframes(dfs)

    574         else:

-->    575             raise ValueError('Unable to
concatenate DataFrame with unknown '

    576                 'division specifying axis=1')

    577     else:

ValueError: Unable to concatenate DataFrames with unknown
division specifying axis=1
```

Intentado Agregar un 'id' columna

```
df['id'] = pd.Series(range(len(df)))
```

Sin embargo, la longitud de Dataframe da como resultado una serie más grande que la memoria.

Python sabe que ambos Dataframe tienen la misma longitud:

```
df.index.compute()
```

```
Int64Index([    0,     1,     2,     3,     4,     5,     6,
              7,     8,     9,
              ...
            1120910, 1120911, 1120912, 1120913, 1120914, 1120915, 1120916,
            1120917, 1120918, 1120919],
            dtype='int64', length=280994776)

In [16]:
df_label1.index.compute()
Out[16]:
Int64Index([1, 5, 5, 2, 2, 2, 2, 2, 2, 2,
              ...
            3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
            dtype='int64', length=280994776)
```

Redimensionamiento, Agrupamiento y Pivoteo

Redimensionamiento

NumPy tiene dos funciones (y también métodos) para cambiar las formas de los arrays - reshape y resize. Tienen una diferencia significativa que será el centro de atención de este capítulo.

Numpy.reshape()

Comencemos con la función para cambiar la forma del array - reshape().

```
import numpy as np

arrayA = np.arange(8)
# arrayA = array([0, 1, 2, 3, 4, 5, 6, 7])

np.reshape(arrayA, (2, 4))
# array([[0, 1, 2, 3],
#        [4, 5, 6, 7]])
```

Convierte un vector de 8 elementos a el array de la forma de (4, 2). Puede ser ejecutada con éxito porque la cantidad de elementos antes y después de reshape es idéntica. Aumenta el ValueError si las cantidades son diferentes.

```
In [1]: np.reshape(arrayA, (3,4))

-----

ValueError                                Traceback (most recent call
last)

ValueError: cannot reshape array of size 8 into shape
(3,4)
```

Echemos un vistazo más de cerca al array de la remodelación. La primera fila contiene los primeros 4 datos del **arrayA** y la segunda fila contiene los últimos 4. Rellena los datos en el orden de la fila en esta conversión de remodelación.

Necesitas cambiar el parámetro **order** si quieres que el orden de llenado de los datos sea de columna.

```
np.reshape(arrayA, (2,4), order='F')  
  
# array([[0,2,4,6],  
        [1,3,5,7]])
```

El valor por defecto de orden es C, que significa leer o escribir datos en un orden de índice similar a C, o en palabras simples, en el orden de la fila. F significa leer o escribir datos en un orden de índice similar al Fortan, o digamos, en el orden de la column.

Ndarray.reshape()

Además de la función reshape, NumPy tiene también el método reshape en el objeto ndarray. El método tiene los mismos parámetros que la función pero sin el array como parámetro.

```
arrayB = arrayA.reshape((2,4))  
arrayB  
  
# array([[0,1,2,3],  
        [4,5,6,7]])  
  
arrayA  
  
# array([0,1,2,3,4,5,6,7])
```

Puedes ver que el método reshape es similar a la función reshape. Y también debería ser consciente de que el método ndarray.reshape() no cambia los datos y la forma del array original, sino que devuelve una nueva instancia ndarray.

Agrupación

El agrupamiento de datos o binning en inglés, es un método de preprocesamiento de datos y consiste en agrupar valores en compartimentos. En ocasiones este agrupamiento puede mejorar la precisión de los modelos predictivos y, a su vez, puede mejorar la comprensión de la distribución de los datos. El método nos lo proporciona pandas y se llama “pd. cut”.

Veamos un ejemplo utilizando los datos del Titanic, vamos agrupar en compartimentos los datos de la columna de la “Edad”, en este caso vamos a crear seis grupos de edades, divididos de la siguiente forma:

- El primer grupo lo comprenda las personas con edades entre 0 a 5,
- El segundo grupo serán las personas con edades entre 6 a 12,
- En el tercer grupo estarán las personas entre 13 a 18 años,
- En el cuarto grupo estará formado por las personas con edades comprendidas entre 19 a 35 años,
- El quinto lo forman las personas entre 36 años a 60, y
- El último grupo está comprendido por las personas entre 61 año a 100 años.

Toma en cuenta que estos rangos son seleccionados al azar. tú puedes seleccionar tus propios rangos de edades, de acuerdo a tu análisis.



Una vez definido nuestros rangos vamos a crear una variable llamada “bins” en donde colocaremos nuestros rangos, solamente se debe colocar desde donde inicia el primer rango y en donde finaliza el resto de los rangos, por esa razón colocamos de primero el cero que es donde inicia nuestro primer rango, y seguidamente colocamos en donde termina el resto de los rangos.


```
bins = [0, 5, 12, 18, 35, 60, 100]
```

Seguidamente creamos otra variable llamada “names” en donde colocamos los nombres que le vamos a poner a cada uno de los compartimientos o en este caso rango de edades, podríamos colocarle, bebe, niño, adolescente, adulto, etc, pero como te lo he comentado en varias ocasiones los algoritmos de Machine Learning por lo general solamente aceptan números entonces es preferible colocarles número a estos compartimientos para que posteriormente no se tenga que hacer un cambio en los nombres. Entonces simplemente le colocamos 1 al primer rango correspondiente de 0 a 5, le colocamos 2 al siguiente rango y así sucesivamente.

```
names = ["1", "2", "3", "4", "5", "6"]
```

Definidos ya los bins o rangos y los nombres que van a llevar ya podemos implementar el método “cut”, para crear los grupos de datos. Lo único que debemos hacer es definir la columna a editar en este caso “Edad” y le decimos al método los rangos en que lo vamos a dividir, en este caso son los que definimos en la variable “bins” por tal razón solamente escribimos este nombre y seguidamente le colocamos los nombres de cada rango que lo definimos en la variable “names”.

Esto es todo con este método se hace la agrupación respectiva y ahora en vez de tener cada persona con su respectiva edad, ahora los tenemos agrupados en rangos de edades.

```
df["Edad"] = pd.cut(df["Edad"], bins, labels = names)
```

Esta función es muy útil en muchos ejercicios, por ejemplo, en este del Titanic, al finalizar del respectivo análisis podríamos ver a qué grupo correspondía las personas que sobrevivieron el desastre, si corresponde a niños, adolescente o por el contrario son personas mayores.

Pivoteo

Visualizamos datos con la función pivote de DataFrame.

Ejemplo:

	select_me	estimator	score
0	relief_f	knn	0.806452
1	relief_f_n	knn	0.870968
2	relief_f	svm	0.967742
3	relief_f_n	svm	0.935484
4	relief_f	dt	0.967742
5	relief_f_n	dt	0.870968
6	relief_f	gnb	0.967742
7	relief_f_n	gnb	0.935484
8	relief_f	lr	0.935484
9	relief_f_n	lr	0.83871

1. Se leen los datos en formato dataframe

```
import pandas as pd

import matplotlib.pyplot as plt

df = pd.read_csv('score(1).csv,' header = None)

data_original = {'algorithm': df.iloc[1:,1].values,
'target': df.iloc[1:,2].values,
'number':df.iloc[1:,3].values}

data_original = pd.DataFrame(data_original)

data_original
```

	algorithm	target	number
0	relief_f	knn	0.806451613
1	relief_f_n	knn	0.870967742
2	relief_f	svm	0.967741935
3	relief_f_n	svm	0.935483871
4	relief_f	dt	0.967741935
5	relief_f_n	dt	0.870967742
6	relief_f	gnb	0.967741935
7	relief_f_n	gnb	0.935483871
8	relief_f	lr	0.935483871
9	relief_f_n	lr	0.838709677

2. Utilice pivot Función para pivotar DataFrame

```
data = data_original.pivot(index = 'algorithm',columns =
'target',values='number')
```

```
data
```

	target	dt	gnb	knn	lr	svm
algorithm						
relief_f	0.967741935	0.967741935	0.806451613	0.935483871	0.967741935	
relief_f_n	0.870967742	0.935483871	0.870967742	0.838709677	0.935483871	

3. Solo necesitamos datos, por lo que:

```
data.values
```

```
array([[ '0.967741935', '0.967741935', '0.806451613', '0.935483871',  
        '0.967741935'],  
       ['0.870967742', '0.935483871', '0.870967742', '0.838709677',  
        '0.935483871']], dtype=object)
```

```
plt.plot(range(5), data.values[0,:], 'r-',label =  
        'relief_f')  
  
plt.plot(range(5), data.values[1,:], 'g-',label =  
        'relief_f_n')  
  
plt.xticks(range(5),data.columns)  
  
plt.xlabel('Target')  
  
plt.ylabel('Precision')  
  
plt.title("The Precision")  
  
  
ax = plt.gca()  
ax.invert_yaxis () #y eje inverso  
  
  
plt.legend () # Muestra la leyenda en la esquina inferior  
izquierda  
  
plt.show()
```


Referencias

[1] Indexación Jerárquica

<https://living-sun.com/es/python/689481-pandas-matching-on-level-of-hierarchical-index-python-pandas-indexing.html>

[2] Listas Indexadas

<https://www.youtube.com/watch?v=2-uXLQbsHIA>

[3] Multi Index

<https://www.youtube.com/watch?v=bWjB4089EbA>

[4] Combinación y merge de datos

https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html

[5] Fusionar Pandas Dataframes en el índice.

<https://www.delftstack.com/es/howto/python-pandas/merge-dataframes-on-index-in-pandas/>

[6] Agrupar datos

<https://www.youtube.com/watch?v=ZhrJrrXrpwo>