In [2]:

```
import pandas as pn
import numpy as np
```

DATA WRANGLING

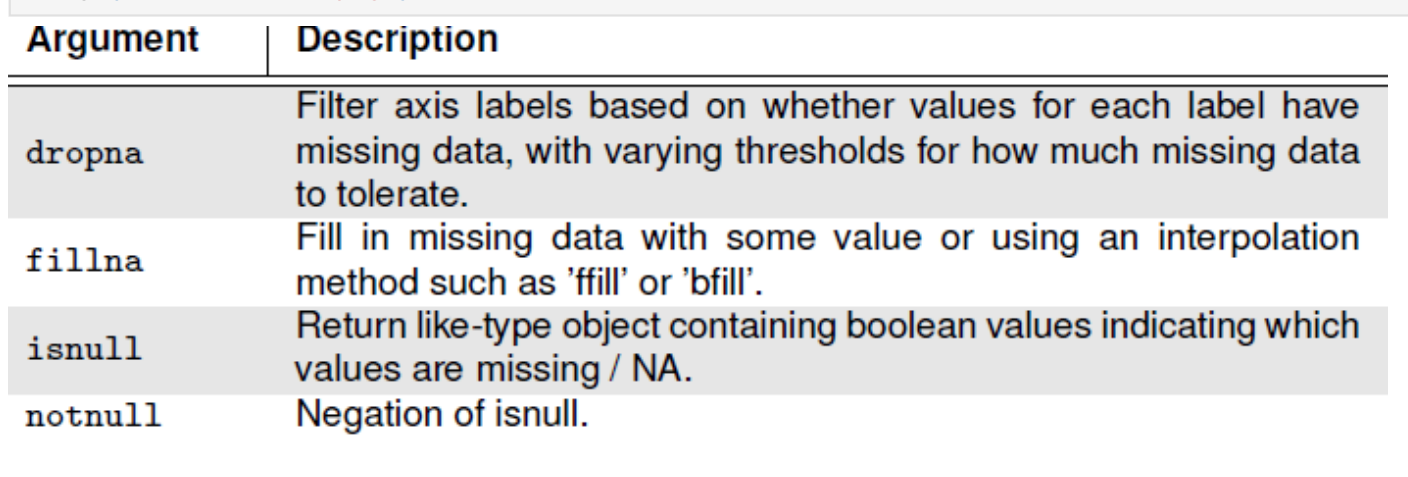
Filtering out missing data

Distinguimos distintos métodos para filtrar los datos nulos de una set de datos:

In [3]:

```
from IPython.display import Image
Image("null_methods.png")
```

Out[3]:



Argument	Description
`dropna`	Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.
`fillna`	Fill in missing data with some value or using an interpolation method such as 'ffill' or 'bfill'.
`isnull`	Return like-type object containing boolean values indicating which values are missing / NA.
`notnull`	Negation of isnull.

df.dropna()

Como su nombre indica permite eliminar (por defecto) las filas que contengan al menos un valor nulo (NaN o np.nan). Funciona tanto para series como para dataframes:

In [4]:

```
Serie_ejemplo=pn.Series([1,np.nan,3.5,np.nan,7])
df=pn.DataFrame({0:[1,np.nan,6.5,3,0],1:[1,np.nan,np.nan,np.nan,1.4],2:[np.nan,np.nan,np.nan,np.nan,2.7],3:[np.nan,np.nan,6.5,3,3.5],4:[1,np.nan,3.4,2.6,1]})
print(Serie_ejemplo)
print(df)
```

```
0    1.0
1    NaN
2    3.5
3    NaN
4    7.0
dtype: float64
     0    1    2    3    4
0  1.0  1.0  NaN  NaN  1.0
1  NaN  NaN  NaN  NaN  NaN
2  6.5  NaN  NaN  6.5  3.4
3  3.0  NaN  NaN  3.0  2.6
4  0.0  1.4  2.7  3.5  1.0
```

In [5]:

```
Serie_ejemplo.dropna()
```

Out[5]:

```
0    1.0
2    3.5
4    7.0
dtype: float64
```

In [6]:

```
df.dropna()
```

Out[6]:

	0	1	2	3	4
4	0.0	1.4	2.7	3.5	1.0

.dropna() por defecto elimina **las filas** que tienen **al menos** un valor NaN, pero podemos especificarle que únicamente elimine aquellas filas que tengan **todos sus valores NaN** con **.dropna(how='all')**:

.dropna(how='all')

Elimina las filas cuyos todos sus valores son NaN:

In [7]:

```
Serie_ejemplo.dropna(how='all')
```

Out[7]:

```
0    1.0
2    3.5
4    7.0
dtype: float64
```

In [8]:

```
df.dropna(how='all')
```

Out[8]:

	0	1	2	3	4
0	1.0	1.0	NaN	NaN	1.0
2	6.5	NaN	NaN	6.5	3.4
3	3.0	NaN	NaN	3.0	2.6
4	0.0	1.4	2.7	3.5	1.0

.dropna(axis=1)

En vez de eliminar las columnas que tengan al menos un valor NaN, ahora elimina las **columnas** que tengan al menos un valor NaN:

In [9]:

```
df.dropna(axis=1)
```

Out[9]:

0
1
2
3
4

.fillna()

Como su nombre indica, permite rellenar los huecos que tienen valor nulo (NaN) por el valor que especifiques en el paréntesis (variable, int, string, float ,boolean, serie...) en cualquier serie o dataframe:

In [10]:

```
Serie_ejemplo.fillna('valor_de_relleno')
```

Out[10]:

```
0                   1
1    valor_de_relleno
2                 3.5
3    valor_de_relleno
4                   7
dtype: object
```

In [11]:

```
df.fillna('valor_de_relleno')
```

Out[11]:

	0	1	2	3	4
0	1	1	valor_de_relleno	valor_de_relleno	1
1	valor_de_relleno	valor_de_relleno	valor_de_relleno	valor_de_relleno	valor_de_relleno
2	6.5	valor_de_relleno	valor_de_relleno	6.5	3.4
3	3	valor_de_relleno	valor_de_relleno	3	2.6
4	0	1.4	2.7	3.5	1

.fillna({'nombre_columna':valor_por_el_que_quieras_sustituir_los_NaN})

Al introducir un diccionario en el paréntesis, te permite esque cificar que en la columna X te sustituya los valores NaN de esa columna por el valor que hayas especificado en el value:

In [12]:

```
a=df.fillna({1:'valor_sust_columna_1',2:'valor_sust_columna_2'})
```

In [13]:

```
Serie_ejemplo=pn.Series([1,np.nan,3.5,np.nan,7])
df=pn.DataFrame({0:[1,np.nan,6.5,3,0],1:[1,np.nan,np.nan,np.nan,1.4],2:[np.nan,np.nan,np.nan,np.nan,2.7],3:[np.nan,np.nan,6.5,3,3.5],4:[1,np.nan,3.4,2.6,1]})
df
```

Out[13]:

	0	1	2	3	4
0	1.0	1.0	NaN	NaN	1.0
1	NaN	NaN	NaN	NaN	NaN
2	6.5	NaN	NaN	6.5	3.4
3	3.0	NaN	NaN	3.0	2.6
4	0.0	1.4	2.7	3.5	1.0

.fillna(method='ffill/bfill', limit=X)

Con .fillna(limit=X) lo que haces es **indicar el número de sustituciones que hará por columna**. Por ejemplo .fillna(limit=3) significa que solo sustituirá hasta 3 NaN por columna, el resto lo deja como está. Es imporescindible especificar **el método**, ffill te sustituye empezando por arriba y bfill empezando por abajo :

In [14]:

```
df.fillna(method='ffill',limit=3)
```

Out[14]:

	0	1	2	3	4
0	1.0	1.0	NaN	NaN	1.0
1	1.0	1.0	NaN	NaN	1.0
2	6.5	1.0	NaN	6.5	3.4
3	3.0	1.0	NaN	3.0	2.6
4	0.0	1.4	2.7	3.5	1.0

In [15]:

```
df.fillna(method='bfill',limit=2)
```

Out[15]:

	0	1	2	3	4
0	1.0	1.0	NaN	6.5	1.0
1	6.5	NaN	NaN	6.5	3.4
2	6.5	1.4	2.7	6.5	3.4
3	3.0	1.4	2.7	3.0	2.6
4	0.0	1.4	2.7	3.5	1.0

Grouping data

In [16]:

```
df_gd=pn.DataFrame({'data1':[-0.204708,0.478943,-0.519439,-0.555730,1.965781],'data2':[1.393406,0.092908,0.281746,0.769023,1.246435],'key1':['a','a','b','b','a'],'key2':['one','two','one','two','one']})
df_gd
```

Out[16]:

	data1	data2	key1	key2
0	-0.204708	1.393406	a	one
1	0.478943	0.092908	a	two
2	-0.519439	0.281746	b	one
3	-0.555730	0.769023	b	two
4	1.965781	1.246435	a	one

Como su nombre indica, agrupar datos nos permite crear a partir de una dataframe un elemento llamado **objeto agrupado**, que es una especie de serie en el que los datos de una columna del dataframe original se agrupan en base a los datos de otra columna del dataframe orignial, y para crearlo necesitamos el elemento **.groupby()**:

df['columna_con_los_datos_que_queremos_usar'].groupby(df['columna_por_la_que_queremos_agrupar_los_datos'])

In [17]:

```
grouped_object=df_gd['data1'].groupby(df_gd['key1']) #Creamos el grouped_object
grouped_object
```

Out[17]:

```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x0693ECA0>
```

Ahora todos los cálculos que se hagan sobre el objeto agrupado se harán de forma individual por cada agrupamiento, por ejemplo si queremos calcular la media de 'data1', lo hará según las letras de 'key1', es decir, la media de las filas que tengan como key1 la letra a por un lado y por otro las b:

In [18]:

```
grouped_object.mean()
```

Out[18]:

```
key1
a    0.746672
b   -0.537584
Name: data1, dtype: float64
```

Merging data

Merging consiste en fusionar 2 columnas según una columna o índice que los relacione. Hay 4 métodos para hacer merging: inner,outer,right y left:

- **Inner**: Es aquel que se hace por defecto. Es un merging uno a muchos, en el que no habrá valores NaN, es de cir, los valores coincidentes se rellenan en todas las posibilidades y los que no coincidan se eliminan.
- **Outer**: Es un merging 1 a 1, es decir, los valores coincidentes se unen mientras que los no coincidentes conservan su valor original pero en la nueva columna ponen un NaN porque no hay ningún valor coincidente (es un inner pero los valores no coincidentes de ambas tablas se muestran en la nueva con un NaN en la columna que no tienen pareja).
- **Left/Right**: Se hace un merging muchos a muchos, es decir, se generan todas las posibles combinaciones entre los datos coincidentes de las 2 tablas, y los que no coincidan pues se pone un valor NaN en la columna que no coincida (es como un 'outter' pero si pones 'right' solo se conservan los valores no coincidentes de la segunda tabla/serie, rellenando con NaN la columna sin pareja, y si pones 'left' se conservan los de la primera tabla/serie.

pn.merge(df_1, df_2, on='columna_comun', how='outer'/'inner'/'right')

Partimos de 2 dataframes:

In [19]:

```
df_merge_1=pn.DataFrame({'data1':[0,1,2,3,4,5,6],'key':['b','b','a','c','a','a','b']})
df_merge_2=pn.DataFrame({'data2':[0,1,2,0],'key':['a','b','d','b']})
print(df_merge_1)
print('')
print(df_merge_2)
```

```
   data1 key
0      0   b
1      1   b
2      2   a
3      3   c
4      4   a
5      5   a
6      6   b

   data2 key
0      0   a
1      1   b
2      2   d
3      0   b
```

In [20]:

```
df_juntos=pn.merge(df_merge_1,df_merge_2, on='key', how='inner')
df_juntos
```

Out[20]:

	data1	key	data2
0	0	b	1
1	0	b	0
2	1	b	1
3	1	b	0
4	6	b	1
5	6	b	0
6	2	a	0
7	4	a	0
8	5	a	0

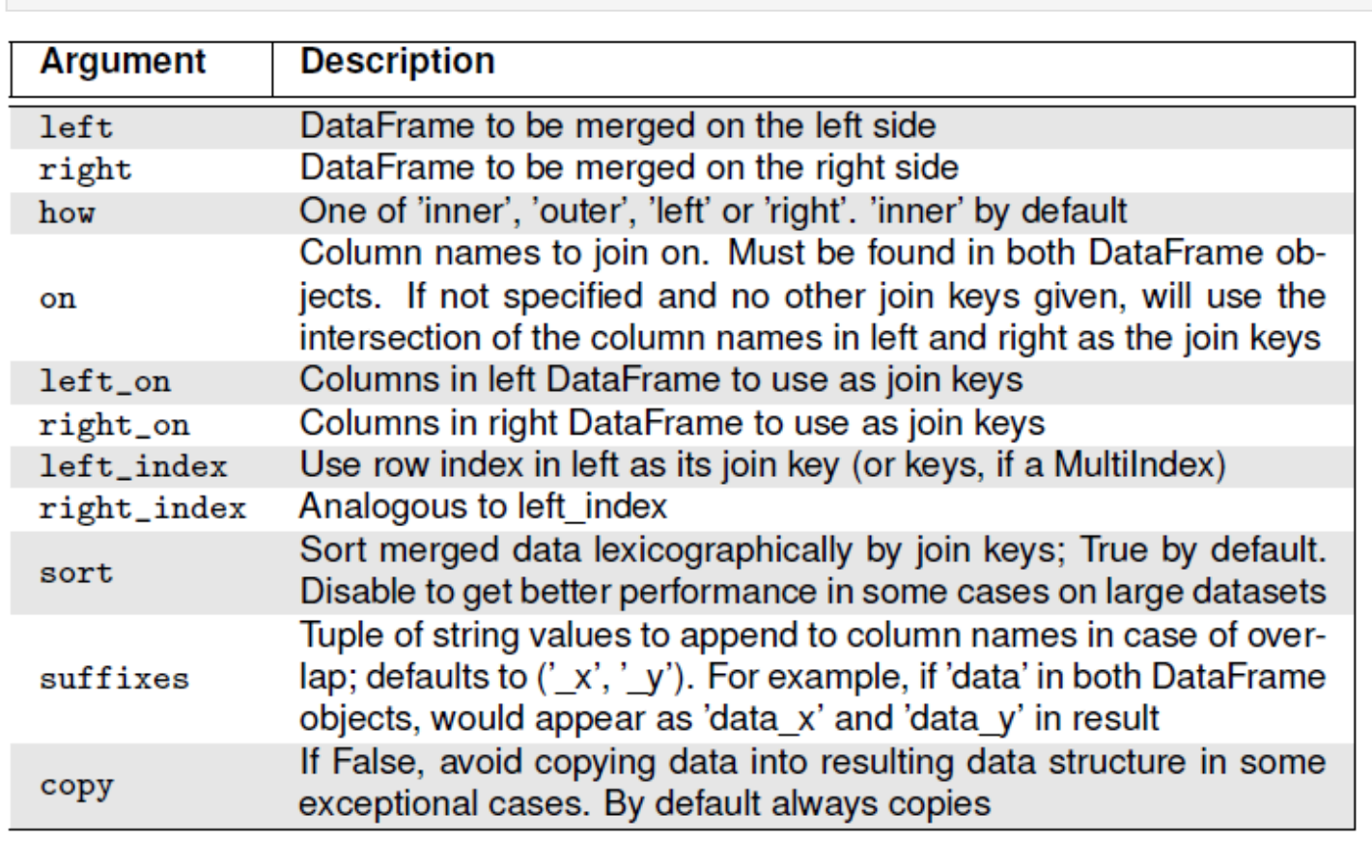
Podemos ver que las tablas se han unido mediante un **innerjoin** por la columna común que en este caso es **'key'**

Se puede dar el caso de que la columna común a 2 tablas no se llame igual, o que la columna común sea el índice, por lo que existen métodos específicos para indicarlo:

In [21]:

```
from IPython.display import Image
Image("merging_methods.png")
```

Out[21]:



Argument	Description
`left`	DataFrame to be merged on the left side
`right`	DataFrame to be merged on the right side
`how`	One of 'inner', 'outer', 'left' or 'right'. 'inner' by default
`on`	Column names to join on. Must be found in both DataFrame objects. If not specified and no other join keys given, will use the intersection of the column names in left and right as the join keys
`left_on`	Columns in left DataFrame to use as join keys
`right_on`	Columns in right DataFrame to use as join keys
`left_index`	Use row index in left as its join key (or keys, if a MultiIndex)
`right_index`	Analogous to left_index
`sort`	Sort merged data lexicographically by join keys; True by default. Disable to get better performance in some cases on large datasets
`suffixes`	Tuple of string values to append to column names in case of overlap; defaults to ('_x', '_y'). For example, if 'data' in both DataFrame objects, would appear as 'data_x' and 'data_y' in result
`copy`	If False, avoid copying data into resulting data structure in some exceptional cases. By default always copies

Por ejemplo si tenemos las siguientes tablas y queremos hacer un **leftjoining** usando **como key** la **columna de key** de la tabla izquierda y **el índice** como **key** de la tabla derecha

In [22]:

```
left1=pn.DataFrame({'key':['a','b','a','a','b','c'],'value':[0,1,2,3,4,5]})
right1=pn.DataFrame({'group_val':[3.5,7.]},index=['a','b'])
print (left1)
print('')
print(right1)
```

```
  key  value
0   a      0
1   b      1
2   a      2
3   a      3
4   b      4
5   c      5

   group_val
a        3.5
b        7.0
```

In [23]:

```
df_merged=pn.merge(left1, right1, left_on='key', right_index=True, how='inner')
df_merged
```

Out[23]:

	key	value	group_val
0	a	0	3.5
2	a	2	3.5
3	a	3	3.5
1	b	1	7.0
4	b	4	7.0

Concatenating data sets

pn.concat([df1,df2,serie1,serie2...])

Concatenar es una forma de unir "a lo bruto" 2 o más dataframes o series entre sí. Podemos unir dataframes, series, o los dos. Por defecto, se unirán en vertical, es decir, el dataframe 2 quedará debajo del dataframe 1, y se conservará el índice de cada uno de ellos. Partimos de 2 dataframes:

In [24]:

```
df1=pn.DataFrame(np.arange(6).reshape(3,2), index=['a','b','c'], columns=['one','two'])

df2=pn.DataFrame(5 + np.arange(4).reshape(2,2),index=['a','c'],columns=['three','four'])

print(df1)
print('')
print(df2)
```

```
   one  two
a    0    1
b    2    3
c    4    5

   three  four
a      5     6
c      7     8
```

Ahora las concatenamos:

In [25]:

```
df1_2=pn.concat([df1,df2])
df1_2
```

Out[25]:

	one	two	three	four
a	0.0	1.0	NaN	NaN
b	2.0	3.0	NaN	NaN
c	4.0	5.0	NaN	NaN
a	NaN	NaN	5.0	6.0
c	NaN	NaN	7.0	8.0

Como podemos ver, se han unido en vertical y han conservado el índice. Si no queremos que conserven el índice podemos poner **ignore_index=True**:

In [26]:

```
df1_2=pn.concat([df1,df2], ignore_index=True)
df1_2
```

Out[26]:

	one	two	three	four
0	0.0	1.0	NaN	NaN
1	2.0	3.0	NaN	NaN
2	4.0	5.0	NaN	NaN
3	NaN	NaN	5.0	6.0
4	NaN	NaN	7.0	8.0

También podemos querer que se unan en horizontal y no en vertical, por lo que debemos poner **axis=1**:

In [27]:

```
df1_2=pn.concat([df1,df2], axis=1)
df1_2
```

Out[27]:

	one	two	three	four
a	0	1	5.0	6.0
b	2	3	NaN	NaN
c	4	5	7.0	8.0

Se puede dar el caso de que querramos **poder acceder y saber a qué tabla original pertenecían cada uno de los elementos que forman parte del concat**, y para eso tenemos **keys=[key_df_1,key_df_2...]**:

In [28]:

```
df1_2=pn.concat([df1,df2], keys=['df1','df2'])
df1_2
```

Out[28]:

		one	two	three	four
df1	**a**	0.0	1.0	NaN	NaN
	b	2.0	3.0	NaN	NaN
	c	4.0	5.0	NaN	NaN
df2	**a**	NaN	NaN	5.0	6.0
	c	NaN	NaN	7.0	8.0

Por tanto podremos acceder rápidamente con .loc['key'] a los datos del df original que queramos:

In [29]:

```
df1_2.loc['df1']
```

Out[29]:

	one	two	three	four
a	0.0	1.0	NaN	NaN
b	2.0	3.0	NaN	NaN
c	4.0	5.0	NaN	NaN

Podemos también fusionar df con series, por ejemplo:

In [30]:

```
series=pn.Series([1,3,5],index=['a','b','c'],name='serie_ejemplo')
print(series)
```

```
a    1
b    3
c    5
Name: serie_ejemplo, dtype: int64
```

In [31]:

```
df_1_2_serie=pn.concat([df1,series], axis=1)
df_1_2_serie
```

Out[31]:

	one	two	serie_ejemplo
a	0	1	1
b	2	3	3
c	4	5	5

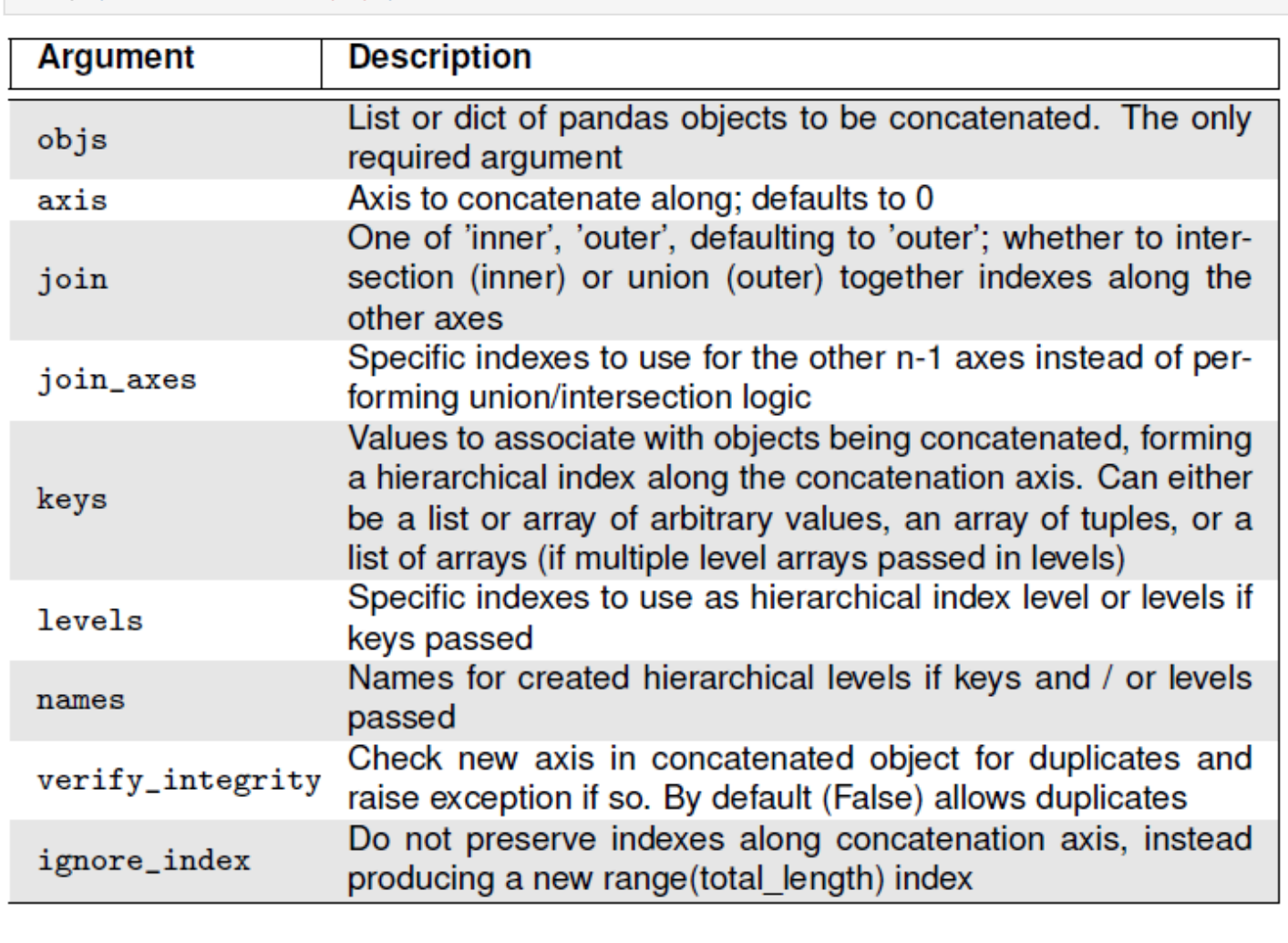
Importante poner en la serie un nombre con name='XXXXX' para que se le pueda asignar un nombre de columna a la serie, porque si no no funciona. Tambien hay que tener en cuenta que el concat horizontal SE HARÁ EN FUNCIÓN DE LOS ÍNDICES , POR LO QUE SI NO COINCIDEN NO SE UNEN

Hay una gran cantidad de argumentos para pn.concat():

In [32]:

```
from IPython.display import Image
Image("concat_methods.png")
```

Out[32]:



Argument	Description
`objs`	List or dict of pandas objects to be concatenated. The only required argument
`axis`	Axis to concatenate along; defaults to 0
`join`	One of 'inner', 'outer', defaulting to 'outer'; whether to intersection (inner) or union (outer) together indexes along the other axes
`join_axes`	Specific indexes to use for the other n-1 axes instead of performing union/intersection logic
`keys`	Values to associate with objects being concatenated, forming a hierarchical index along the concatenation axis. Can either be a list or array of arbitrary values, an array of tuples, or a list of arrays (if multiple level arrays passed in levels)
`levels`	Specific indexes to use as hierarchical index level or levels if keys passed
`names`	Names for created hierarchical levels if keys and / or levels passed
`verify_integrity`	Check new axis in concatenated object for duplicates and raise exception if so. By default (False) allows duplicates
`ignore_index`	Do not preserve indexes along concatenation axis, instead producing a new range(total_length) index

Combinar datasets

set1.combine_first(set2)

Combinar datasets permite enfrentar 2 sets de datos (series o dataframes) y rellenar los datos nulos del primero con el correspondiente valor del segundo:

Por ejemplo si tenemos 2 series:

In [33]:

```
serie1=pn.Series([np.nan, 2.5, np.nan, 3.5, 4.5, np. nan], index=['f', 'e', 'd', 'c', 'b', 'a'])
serie2=pn.Series(np. arange ( len(serie1), dtype=np.float64),index=['f', 'e', 'd', 'c', 'b', 'a'])

print(serie1 , serie2)
```

```
f    NaN
e    2.5
d    NaN
c    3.5
b    4.5
a    NaN
dtype: float64 f    0.0
e    1.0
d    2.0
c    3.0
b    4.0
a    5.0
dtype: float64
```

Queremos rellenar los datos nulos de serie1 con los datos correspondientes de serie2, por lo que usamos .combine_first():

In [34]:

```
serie1.combine_first(serie2)
```

Out[34]:

```
f    0.0
e    2.5
d    2.0
c    3.5
b    4.5
a    5.0
dtype: float64
```

Lo mismo se aplica a dataframes:

In [35]:

```
df1=pn.DataFrame({'a':[1,np.nan,5,np.nan],'b':[np.nan,2,np.nan,6],'c':range(2,18,4)})
df2= pn.DataFrame({'a':[5,4,np.nan,3,7],'b':[np.nan,3,4,6,8]})
print(df1)
print(df2)
```

```
     a    b   c
0  1.0  NaN   2
1  NaN  2.0   6
2  5.0  NaN  10
3  NaN  6.0  14
     a    b
0  5.0  NaN
1  4.0  3.0
2  NaN  4.0
3  3.0  6.0
4  7.0  8.0
```

In [36]:

```
df1.combine_first(df2)
```

Out[36]:

	a	b	c
0	1.0	NaN	2.0
1	4.0	2.0	6.0
2	5.0	4.0	10.0
3	3.0	6.0	14.0
4	7.0	8.0	NaN

Reshaping datasets

df.stack() --> Convierte las columnas de un dataframe en índices, generando un dataframe con índice jerárquico

df.unstack() --> Convierte los índices del dataframe del índice jerárquico en columnas, opuesto a df.stack()

In [37]:

```
df1_stack=df1.stack()
df1_stack
```

Out[37]:

```
0  a     1.0
   c     2.0
1  b     2.0
   c     6.0
2  a     5.0
   c    10.0
3  b     6.0
   c    14.0
dtype: float64
```

Podemos ver que las columnas ahora son un nuevo índice

Para deshacerlo:

In [38]:

```
df1_stack.unstack()
```

Out[38]:

	a	b	c
0	1.0	NaN	2.0
1	NaN	2.0	6.0
2	5.0	NaN	10.0
3	NaN	6.0	14.0

Añadir fila a partir de una serie en un dataframe

Podemos añadir una fila más a una lista con el comando **df.append(lista, ignore_index=True)**, obviamente la serie debe seguir la estructura de las columnas de la tabla y debe tener índice para quitarlo.

In [40]:

```
df1=pn.DataFrame({'student_id':['S1','S2','S3','S4','S5'],'name':['Daniella Fenton','Ryder Storey','Bryce Jensen','Ed Bernal','Kwame Morin'],'marks':[200,210,190,222,199]})
df2=pn.Series(['S6','Scarlette Fisher',205],index=['student_id','name','marks'],name='New Row')


df1.append(df2, ignore_index=True)
```

Out[40]:

	student_id	name	marks
0	S1	Daniella Fenton	200
1	S2	Ryder Storey	210
2	S3	Bryce Jensen	190
3	S4	Ed Bernal	222
4	S5	Kwame Morin	199
5	S6	Scarlette Fisher	205