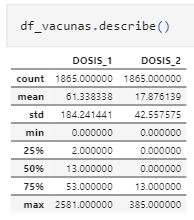
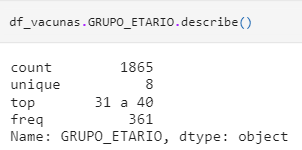
**Clase 11: Pandas II**

Un **paso importante** para el **análisis de datos** es **agregar la información** **en forma eficiente**. Calcular su suma, promedio, máximo, mínimo de los valores de una columna, nos puede dar una primera mirada a los datos. Pero si necesitamos **agrupamientos más complejos**, las operaciones **groupby** que implementan el concepto **split-apply-combine** nos pueden dar respuesta a nuestra necesidad.

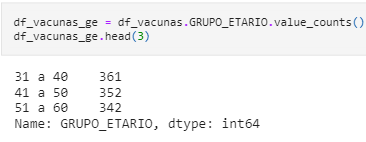
**Estadísticas Descriptivas:** Conjunto de métodos que realizan operaciones matemáticas y estadísticas (*Pandas Aggregated Methods*). A partir de una serie o un DataFrame devuelven un valor o un resumen de valores. IE: **.count** (cuenta los valores no nulos); **.min**; **.max**; **.sum**, **.describe()** genera estadísticas de las columnas del DataFrame. Por default, sólo para las columnas numéricas:



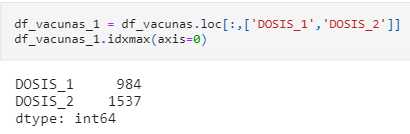
También da algunas estadísticas para datos no numéricos:



**.value\_counts()** da la cantidad de apariciones de cada valor distinto de una columna, ordenados por frecuencia de mayor a menor. Es parecido a un histograma y súper útil a la hora de entender qué tenemos dentro de una columna y verificar si todos los valores se encuentran en el formato esperado.

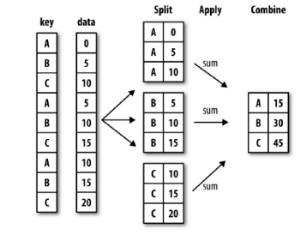


**.idxmax**: Devuelve el índice de fila que contiene el máximo valor de cada columnas.

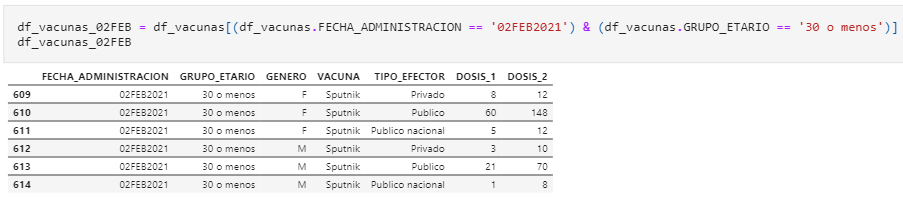


Para agrupamientos más complejos, se puede usar la operación **Groupby**, queimplementa el concepto ***Split-apply-combine***.

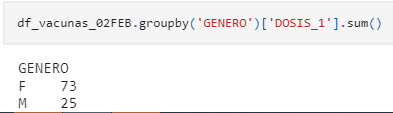
1. Los datos en un objeto Panda (Serie o DataFrame) **se dividen en grupos** en base a una o más keys que nosotros definimos (**split**).
2. Se le **aplica una función a cada** uno de los **grupos** (**apply**).
3. Los **resultados se combinan** en un único resultado (**combine**)



Ejemplo: Split: Selecciona los vacunados para una fecha determinada (2/2/2021) y un grupo étareo determinado (menores de 30 años).



Divide el DataFrame por género (Split); suma para estos 2 grupos el valor de la columna Dosis\_1 (Apply) y por último combina los resultados (Combine).

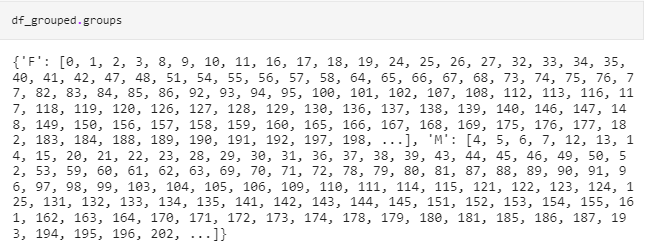


El resultado de aplicar el método .groupby() sobre un DataFrame da como resultado un objeto **DataFrameGroupBy.** Una vez que tenemos el objeto DataFrameGroupBy, podemos aplicarle operaciones y nos va a dar el resultado de aplicar dichas operaciones sobre cada grupo.



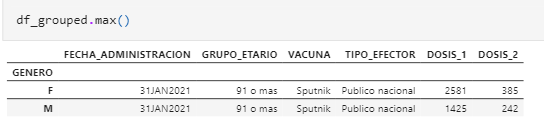
Los grupos se forman con los diferentes valores que tiene la columna usada en el groupby.

Con el método **.groups** se puede ver cada grupo, su valor y los índices de las filas en donde se encuentra:

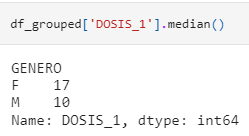


Si **aplicamos operaciones** sobre un objeto **DataFrameGroupBy**, el **resultado** sale en una **Serie o** en un **DataFrame** **indexado** por los **valores únicos de la clave del GroupBy**.

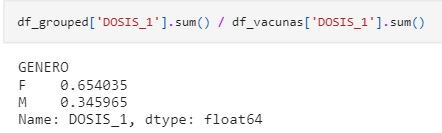
Ejemplo: Usar max en un DataFrameGroupBy que agrupó por género, da como resultado un DataFrame con el máximo de cada columna, midiendo en filas por separado dicho valor para cada valor único de la columna de agrupación género (o sea, si se tienen 3 columnas con valores máximos en distintas filas, va a mostrar 3 columnas con los valores máximos de todas las filas, aunque sea un registro que “no existe”; y tantas filas como valores únicos tenga la columna de agrupamiento):



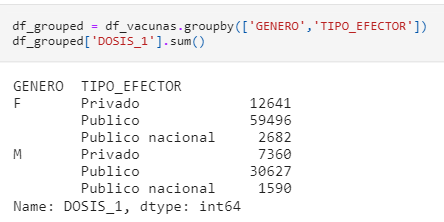
Es posible aplicar la operación sobre una sola columna. Al igual que en el caso anterior, va a dar un resultado en tantas filas como valores únicos tenga la columna de agrupación del Groupby.



Es posible usar el DataFrame original. En este caso, lo usa para calcular el porcentaje de aplicaciones por género. Al aplicar una operación en la que participa un DataFrameGroupBy, el resultado vuelve medido para cada una de las categorías de la columna “Género”, que es la columna de agrupación del GroupBy.



Se puede usar más de una columna como clave: En el siguiente ejemplo, agrupa por Género y por Tipo Efector. Entonces da un resultado para cada una de las combinaciones de valores únicos entre las columnas de agrupación utilizadas:



Otra forma de agrupar es indizar el dataframe por la columna que nos interese. Si tenemos valores asociados a los valores de dicha columna de interés (IE: Tenemos países y tenemos una serie o diccionario con la relación país-continente), podemos usar como argumento del groupby una serie o un diccionario que tenga como índices los valores del dataframe indizado y como columna los valores de agrupamiento deseados asociados a dichos índices.

**En Python:**

# Creo un diccionario que asocia países a continentes:

pais\_en\_continente = { 'Chile': 'America', 'Francia': 'Europa', 'México': 'America', 'Colombia': 'America', 'Uruguay': 'America', 'Estados Unidos': 'America', 'España': 'Europa', 'Italia': 'Europa', 'India': 'Asia', 'Ecuador': 'America', 'Brasil': 'America', 'Australia': 'Oceania', 'Bolivia': 'America', 'Reino Unido': 'Europa', 'Alemania': 'Europa', 'Israel': 'Asia', 'China': 'Asia', 'Venezuela': 'America', 'Países Bajos': 'Europa', 'Canadá': 'America', 'Suiza': 'Europa', 'Turquía': 'Europa', 'Noruega': 'Europa', 'Corea del Sur': 'Asia', 'Polonia': 'Europa', 'Perú': 'America', 'Paraguay': 'America', 'Costa Rica': 'America', 'Japón': 'Asia', 'Marruecos': 'Africa', 'Bélgica': 'Europa', 'Malasia': 'Asia', 'Rusia': 'Europa', 'Sudáfrica': 'Africa', 'Nueva Zelanda': 'Oceania'}

# Tengo una columna con los países en mi DataFrame de interés. Uso esta columna para indizar el DataFrame:

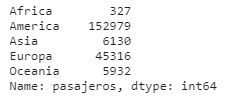
data.index = data.pais\_residencia\_si\_extranjero

# Ahora puedo aplicar el Groupby usando el diccionario:

data\_grouped\_continente = data.groupby(pais\_en\_continente)

# Y puedo estudiar, por ejemplo, cuántos pasajeros vinieron de cada continente:

data\_grouped\_continente[‘pasajeros’].sum()



# Haciendo lo mismo pero en una serie. Primero creo la serie en la cual voy a aplicar el GroupBy:

serie\_pasajeros = data.pasajeros

# Ahora la indizo por país:

Serie\_pasajeros.index = data.pais\_residencia\_si\_extranjero

# Ahora puedo aplicar el GroupBy y sumar:

serie.pasajeros.groupby(pais\_en\_continente).sum()

# Puedo hacer lo mismo, pero usando una serie:

pais\_en\_continente\_serie = pd.Series(pais\_en\_continente)

data\_grouped\_continente\_2 = data.groupby(pais\_en\_continente\_serie)

# Se le puede dar una vuelta de tuerca más generando una función que vaya iterando los valores del índice y devolviendo la etiqueta de agrupamiento asociada a cada valor, o una etiqueta en caso de que el valor del índice no esté en la lista:

def get\_continente(pais):

país\_en\_continente = {‘Chile’: ‘América’, ‘Francia’:’Europa’, … etc.}

if país in país\_en\_continente:

result = país\_en\_continente[país]

else:

result = “desconocido”

return result

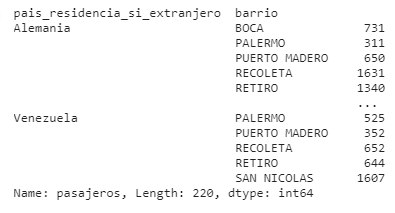
data\_grouped\_func = data.groupby(det\_continente, axis = 0)

#Si armé un DataFrameGroupby con 2 variables de agrupamiento, puedo generar un Dataframe a partir del mismo con el método unstack:

data\_grouped\_pais\_barrio = data.groupby([‘pais\_residencia\_si\_extranjero’,’barrio’])

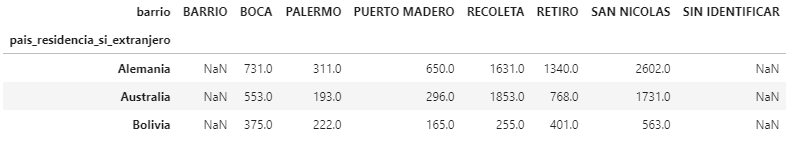
cant\_pasajeros\_pais\_barrio = data\_grouped\_pais\_barrio[‘pasajeros’].sum()

cant\_pasajeros\_pais\_barrio



cant\_pasajeros\_pais\_barrio\_df = cant\_pasajeros\_pais\_barrio.unstack()

cant\_pasajeros\_pais\_barrio\_df.head(3)

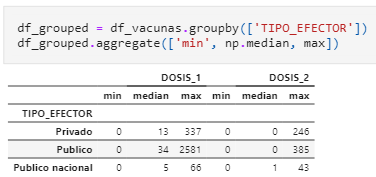


Una vez construidos los grupos, en la **etapa *apply***, se pueden realizar **operaciones de**:

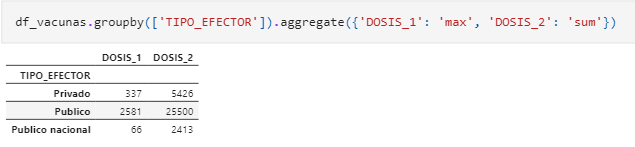
* **Agregación**: Cálculo de estadísticas de resumen para cada grupo; IE sum o mean.
* **Transformación:** Cálculos específicos por grupos devolviendo nuevos objetos indexados del mismo modo. IE: Llenar los NA dentro de un grupo con un valor calculado.
* **Filtro** (descartar algunos grupos de acuerdo a algún cálculo sobre el grupo que devuelva True o False; IE: Descartar los grupos con pocos miembros).

Método **.aggregate()** permite realizar varias agregaciones a la vez. Es **más flexible** que agregaciones como sum(), mean() o median().

Por ejemplo, dado un DataFrame agrupado por la columna “Tipo Efector”, se le puede aplicar el método aggregate para calcular min, mediana y máximo de cada uno de los valores posibles de la columna “Tipo Efector”.



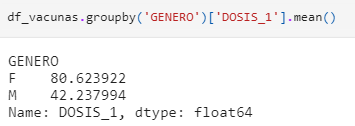
También es posible seleccionar las agregaciones por columna. Hay que generar un diccionario del tipo {columna:agregación}. En este ejemplo, calcula el valor máximo para los diferentes valores únicos de “TIPO\_EFECTOR” en la Dosis\_1; y la suma para los diferentes valores únicos de “TIPO\_EFECTOR” en la Dosis\_2:

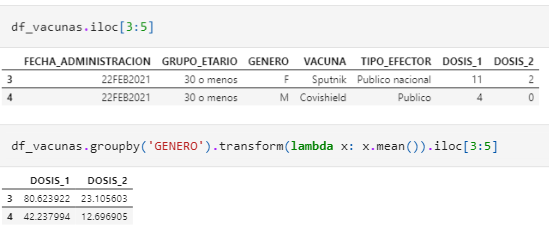


La **agrupación** devuelve un **resumen de los datos.**

El método **.transform** devuelve los **datos a los cuales se les aplicó una función a nivel de grupo**.

La **medida de salida** de la transformación tiene el **mismo shape que el objeto original.** Por ejemplo, calcula la media a la columna Dosis\_1 agrupada por género:





Al aplicar la función lambda con el método **.transform**, como la función **mean()** sólo puede aplicarse a valores numéricos, de todas las columnas sobre las que podría ejecutarse, termina dando sólo resultados para las columnas DOSIS\_1 y DOSIS\_2, que son las únicas columnas con valores numéricos.

**En Python:**

# Vamos a calcular la media condicionada de un DataFrame agrupando por país de procedencia y barrio, para aplicarla a los valores NaN de la columna “pernoctaciones”. Para aplicar un GroupBy, tenemos que tener en cuenta que el mismo no contempla valores NaN:

data\_key\_not\_null\_mask = np-logical\_and(data.pais\_residencia\_si\_extranjero.notnull(), data.barrio.notnull())

data\_key\_not\_null = data.loc[data\_key\_not\_null\_mask,:]

#Ahora hacemos el agrupamiento por país de residencia y barrio.

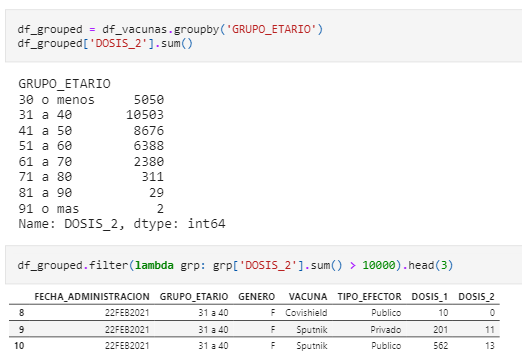
data\_key\_not\_null\_grouped\_pais\_barrio = data\_key\_not\_null.groupby[‘pais\_residencia\_si\_extranjero’,’barrio’]

#Por último hacemos la imputación usando el método transform y la función fillna en una lambda:

data\_filled = data\_key\_not\_null\_grouped\_pais\_barrio[‘pernoctaciones’].transform(lambda grp: grp.fillna(grp.mean()))

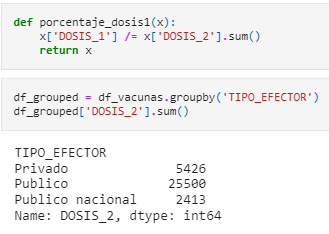
La operación de **Filtro** permite **eliminar datos** a partir de una **propiedad de los grupos**.

Por ejemplo, se toma un DataFrame y se lo agrupa por “Grupo Etario”. A continuación, se aplica sum() sobre la columna “DOSIS\_2” usando el DataFrameGroupBy recién creado. Luego, se usa el método **.filter** para usar una función lambda con una máscara booleana de suma de Dosis\_2 mayor a 10000. Por lo tanto, el análisis termina reduciéndose al grupo etáreo de 31 a 40, que es el único grupo del GroupBy que cumple con esta condición:

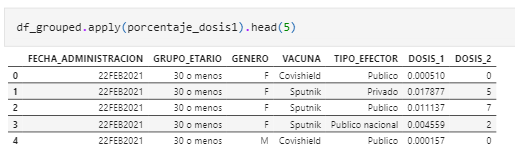


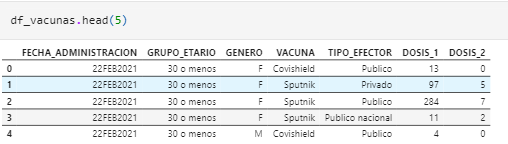
Con el método **.apply()** es posible aplicar una función definida por nosotros a los grupos resultantes de un GroupBy.

Por ejemplo, podemos generar una función que calcule el porcentaje de cada DOSIS\_1 en relación a la suma de todas las DOSIS\_2 para cada tipo de efector. Una vez calculado, podemos devolverlo en cada fila sobre la columna DOSIS1:



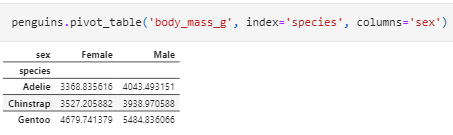
A continuación se muestra el código donde aplica el método .apply con la función recién definida porcentaje\_dosis1





**Tablas Pivot**: También llamadas **tablas dinámicas**. Herramienta generalmente disponible en programas de Hojas de Cálculo. Crea **medidas de resumen** por **una o más keys**, usándolas como **etiquetas de filas y/o columnas**. Pandas cuenta con el método **pivot\_table** sobre DataFrame y también como una función de Pandas.

Ejemplo: Vamos a trabajar con un Dataset de pingüinos llamado penguins. Podemos generar una tabla pivot que muestre la masa corporal entre especie y sexo. **Por default**, la función de agregación que se aplica es el promedio (**mean()**):



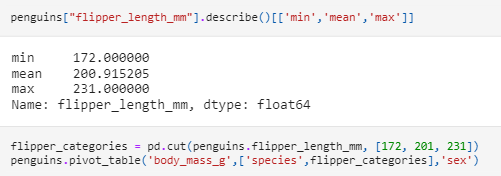
El **primer argumento** es la **columna** sobre la cual **se realiza la agrupación**.

**Index:** Columna cuyos valores sirven de índice en la tabla dinámica (títulos de filas).

**Columns:** Columna cuyos valores sirven de columna en la tabla dinámica (títulos de columnas).

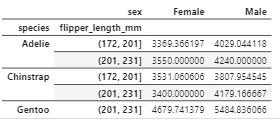
Es posible agregar una **tercera dimensión** además de las definidas en Index y Columns. Podemos generar categorías sobre una columna usando el método **cut**, dividiendo los valores de la columna en intervalos discretos de acuerdo a lo que definamos en el argumento **bins** (lista de valores).

Por ejemplo, a sexo y especie podemos agregarle la variable flipper\_length\_mm, la cual vamos a partir en dos intervalos en función de sus valores mínimos, máximos y media:



Con el método describe() conseguimos los valores mínimos, máximos y media de la columna flipper\_length\_mm. Usamos estos 3 valores para usar el método cut, definiendo los bins por los mismos. Entonces flipper\_length\_mm se va a dividir en una categoría, que serán aquellos valores que estén entre el valor mínimo y la media (no inclusive); y aquellos valores que estén entre la media y su valor máximo (inclusive).

Ahora, al generar la tabla dinámica, va a calcular la media de ‘body\_mass\_g’, para las especies calculando por separado los casos de aquellas especies con flipper por debajo o por encima de la media; y manteniendo en columnas el sexo. **El orden** **en que se ubicaron las categorías** en la lista de filas **afecta la forma en que se presenta** la información en la tabla dinámica.

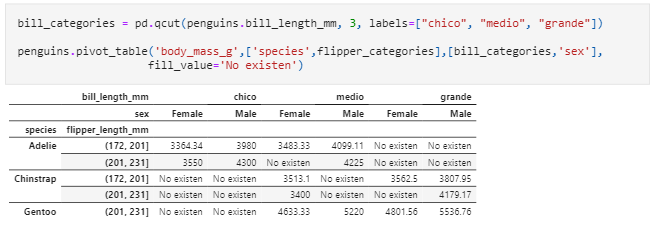


También es posible agregar una nueva columna a la tabla pivot y categorizar sus valores. Con la función **qcut** podemos **definir en cuántos cuantiles** queremos discretizar una columna y podemos generar una **etiqueta** para cada **cuantil** definiendo con una lista el parámetro **labels**.

La sentencia es: nombre\_categoría\_cuantilicada = pd.qcut(nombre\_DF.columna\_DF\_a\_partir,#\_de\_cuantiles\_a\_generar, labels =[lista\_de\_labels])

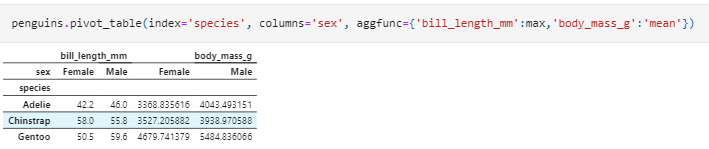
Por ejemplo, podemos definir 3 cuantiles para la variable bill\_length\_mm y etiquetar la longitud del pico como chico, medio o grande.

Con **fill\_value** podemos **etiquetar** los **valores nulos**.

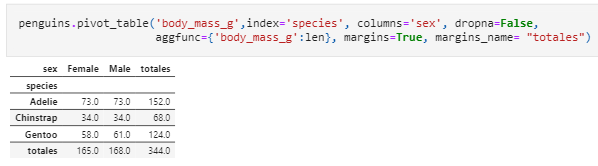


Viendo el código de generación de la tabla dinámica, simplemente se agregó la categoría bill\_categories en la lista de columnas. Como se puso primero, la tabla pivot muestra primero la segregación por tamaño de pico y luego la segregación por sexo.

Es posible **especificar distintas funciones de agregación** para **distintas columnas del DataFrame original**. Simplemente hay que definirlo en el parámetro **aggfunc**. En el próximo ejemplo muestra cómo usa dos variables de agregación, mostrando el máx para una y la media para la otra.

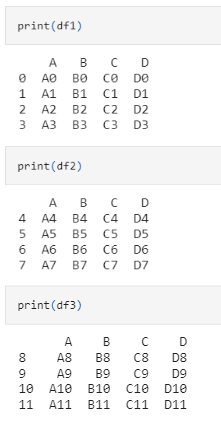


Es posible agregar totales tanto de filas como de columnas, usando los parámetros **margins** y darles un nombre con el parámetro **margins\_name**. Con el parámetro **dropna=False** podemos pedirle **que incluya en la cuenta de totales** (*en filas; no en columnas porque para poder clasificarlo dentro de alguna columna necesariamente tiene que tener un valor definido*) a los valores **nulos**:



**Combinando objetos DataFrame. Joins.** Es posible **combinar Series y DataFrames** con métodos que aplican la **lógica de conjuntos** y **álgebra relacional**. Pensando en las estructuras de datos como **conjuntos**, es posible aplicar funciones tales como **unión, joins, merge e intersect**.

Por Ejemplo: Tenemos 3 DataFrames.



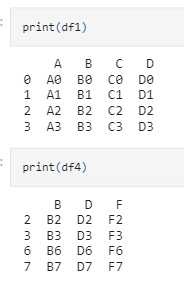
Podemos hacer una **unión** de los mismos por **filas** usando **concat** (tiene un parámetro **axis** que **por default** es **cero**):



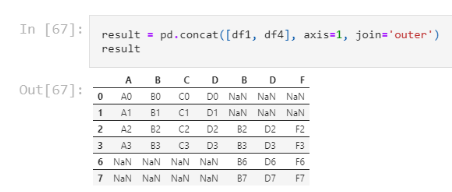
Ahora vamos a hacer una **unión** de los mismos DataFrames, pero esta vez por **columnas** (parámetro **axis = 1**). Hay que agregar el parámetro **join**, que **relaciona las filas con el índice** y **puede tomar 2 valores**:

1. **outer**: Realiza la unión de las columnas sin pérdida de información. Es el default.
2. **inner**: Genera la **intersección** de las columnas. Se **puede perder información**.

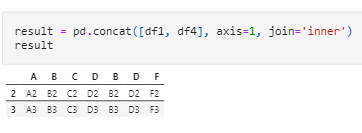
Ejemplo: Tenemos los DataFrame df1 y df4:



Usando el parámetro Join = ‘outer’: Agrega la información del segundo df. En caso **de haber combinaciones de filas y columnas en un DataFrame pero no en el otro**, los **completa con una NaN**.

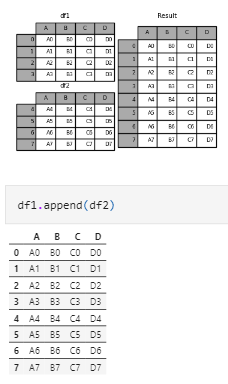


Usando el parámetro Join = ‘inner’:

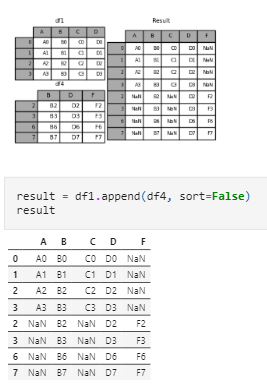


Deja sólo aquellas combinaciones de filas y columnas donde hay información definida en ambos DataFrames, **perdiéndose la información** de aquellas **filas y columnas que estén en uno, pero no en el otro**.

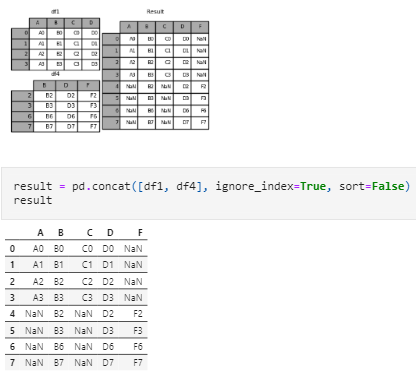
**Append:** Es similar a concat, pero trabaja sólo en filas. Su uso más simple es equivalente a concat con axis = 0.



Append junta todas las filas, incluso si tienen el mismo índice; y junta también todas las columnas. En caso de no encontrar valores, los rellena con una NaN. Realiza la operación entre conjuntos **UNION ALL**:

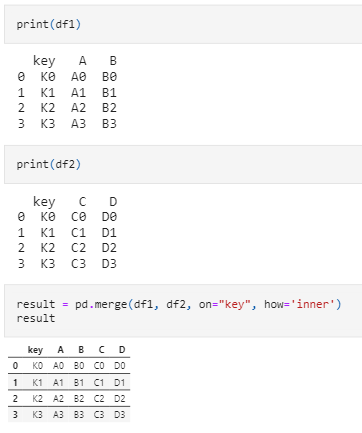


Tanto **concat** como **append** permiten unir DataFrames sobreescribiendo los índices originales por unos nuevos. Para hacer esto hay que usar el parámetro **ignore\_index = True**:



**Merge:** Con Pandas, es posible realizar operaciones de **Joins** entre DataFrames en forma similar a como se trabaja con las bases relacionales de **SQL**. La idea es **vincular** (**join**) las **filas de dos DataFrames** y generar uno nuevo. La **función merge()** sirve como punto de entrada para realizar los **distintos** **tipos de vínculos**. El **método join()** usa internamente al método **merge**. Puede resultar conveniente para vincular DataFrames por sus índices. Los **DataFrames** se **vinculan** a través de las **claves (keys)**, que son **columnas o índices del DataFrame**. El vínculo puede realizarse mediante **una o más columna o índice** con el **mismo nombre en ambos DataFrames.**

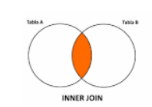
Ejemplo: Tenemos dos DataFrames df1 y df2. Los vinculamos a través de la columna key que está en los dos. El resultado es un nuevo DataFrame que tiene la columna key, todas las columnas de df1 y también todas las columnas de df2:

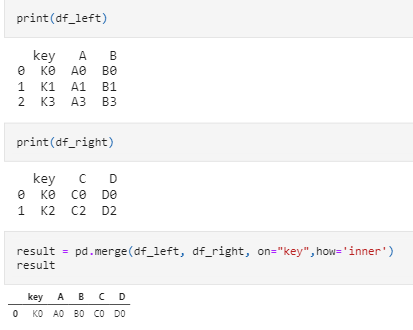


Los **primeros 2 parámetros** son los **DataFrames a vincular**. Al **primero** se lo denomina **izquiedo** y al **segundo**, **derecho**. El parámetro **on** indica la(s) **columna(s) que se usa(n) como clave(s) para vincular** los dataframes. Con el parámetro **how** se indica el **tipo de vínculo**.

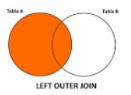
El **Tipo de Vínculo** es **muy importante** porque **define el DataFrame Resultado**. Dependiendo de lo que ocurra con la información en uno y otro DataFrame, tenemos que tomar definiciones:

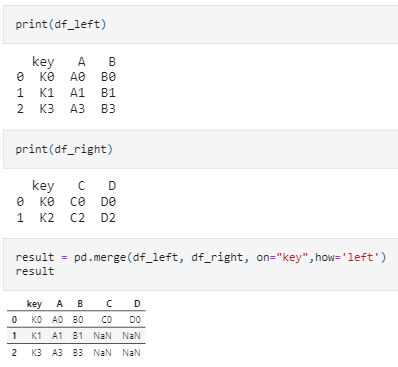
* Caso 1: La fila existe en los 2 DataFrames. Por lo que no tenemos problema.



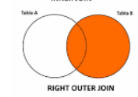


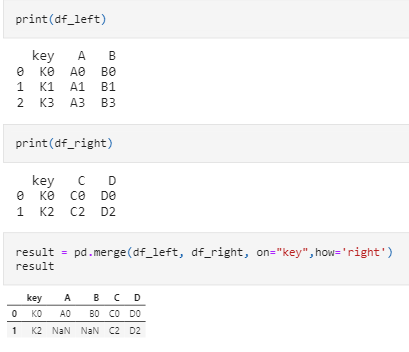
* Caso 2: La fila existe en el DataFrame izquierdo, pero no en el derecho. Acá tenemos 2 caminos posibles: No generar la fila, o bien, generar la fila en el DataFrame Resultado completando con NaN la información que hubiera venido del DataFrame derecho.

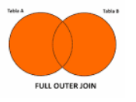




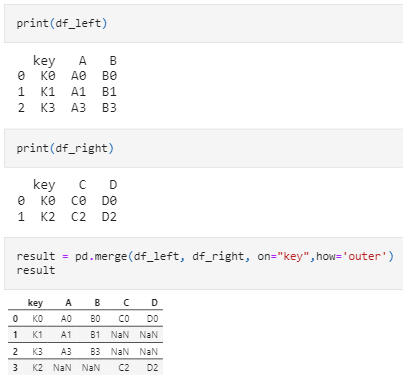
* Caso 3: La fila no existe en el DataFrame izquierdo, pero existe en el derecho. Acá tenemos 2 caminos posibles: No generar la fila, o bien, generar la fila en el DataFrame Resultado completando con NaN la información que hubiera venido del DataFrame izquierdo.

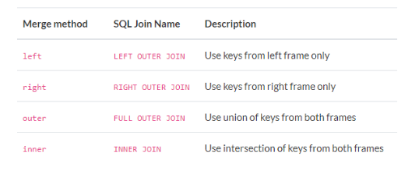




 Es la combinación de **LEFT OUTER JOIN** y **RIGHT OUTER JOIN**.

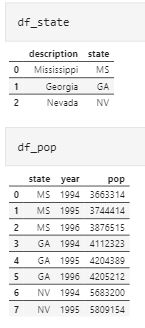
En caso de que el valor exista en ambos DataFrames, no duplica la fila, sino que la fusiona en una sola.



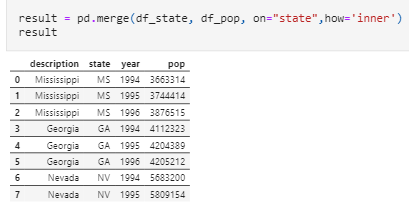


**Merge – Uno a Muchos**: Los **Joins** **uno a uno** son aquellos donde un valor existe en un DataFrame y puede existir o no en el otro, pero una sola vez.

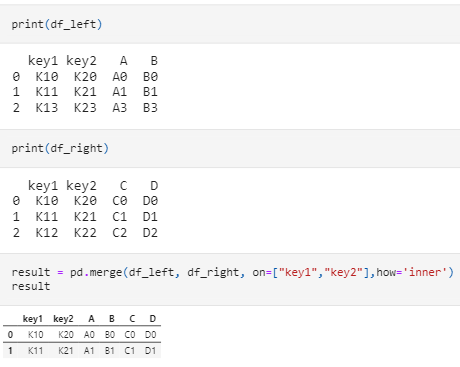
Puede haber casos donde para un **único valor** en el **DataFrame Izquierdo** hay **muchos valores** en el **DataFrame Derecho.** A esto se lo denomina **Joins Uno a Muchos**.



La sintaxis del join es prácticamente igual. La utilidad práctica de estos casos es que podemos complementar el DataFrame que tiene muchos valores con los valores únicos del DataFrame que tiene uno solo.



También es posible trabajar con más de una clave. Simplemente en lugar de definir una columna en **on**, definimos una lista de columnas.



**Función Join**: La función join vincula DataFrames, usando los índices como claves. Ayuda en el sentido de que podemos escribir en forma muy reducida la misma funcionalidad.

Ejemplo: Tenemos 2 dataframes generados a partir de un dicicionario donde se definen las columnas con sus respectivos valores. En index usamos de K0 a K2.



La sintaxis de Join nos ahorra tener que dar el parámetro **on**, porque ya sabe que tiene que matchear los index de ambos DataFrames.

