**Pandas**

Creación, modificación y explotación de **tablas** (datos estructurados)

* **datos estructurados** (geográficamente tienen una coordenadas expresadas en tablas) → Ejemplo bbdd SQL
* datos semi-estructurados (jsons) → Ejemplo bbdd Nosql

**0.1º instalar pandas:**

**!conda install pandas**

**0.2º import library**

import numpy as pd

**import pandas as pd**

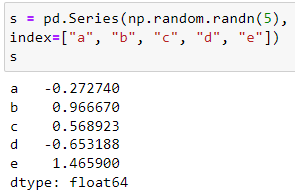
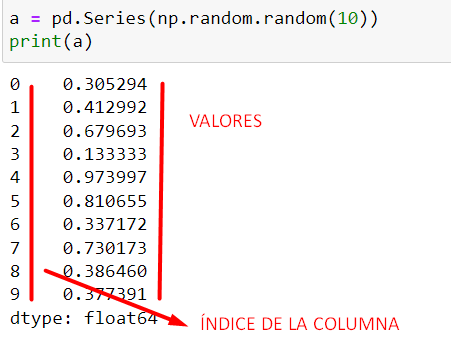
**axis = 0 → INDEX (filas)**

**axis = 1 → COLUMNS**

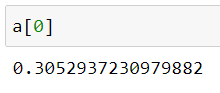
**TIPOS DE ESTRUCTURAS:**

1. **SERIES**: es una **matriz unidimensional** indexada donde los valores pueden ser de **cualquier tipo de datos**. Su “type” será el de sus valores

→ **pd.Series(***data*, index=*nombre filas***)**



* ***EXTRACCIÓN***:
  + en base a su índice → serie**[index-1]**

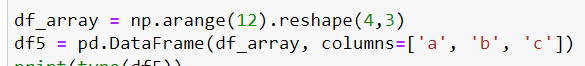
****

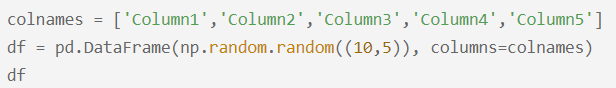
* + con el method get() → serie**.get(**nombre de la fila**) .**  EJ.: s.get(e)
* ***ATRIBUTOS***:
  + serie**.index** → índice de la columna (normalmente empieza por “0”)
  + serie**.size** → Nº de elementos
  + serie**.shape** → (FilasxColumnas) → ¡siempre es de una columna y unidimensional! **(Filas,)** Ej.: **(8,)**
  + serie**.ndim** → Dimensiones (1)
  + serie**.dtypes** → Tipo de sus valores (int, str, object, etc)
  + serie**.axes** → ejes
  + serie**.memory\_usage()** → bytes
  + serie**.describe()** → Te da INFORMACIÓN ÚTIL (count, mean, sd, mean, ...)
  + serie**.head()** → Muestra los PRIMEROS valores de la serie
  + serie**.tail()** → Muestra los ÚLTIMOS vores de la serie
  + **display(**serie**)** → Muestra TODOS valores de la serie

1. **DataFrame**: son estructuras indexadas **bidimensionales** donde cada columna puede ser de **cualquier tipo de datos**. Los DataFrames son muy similares a las hojas de cálculo y las tablas de bases de datos.

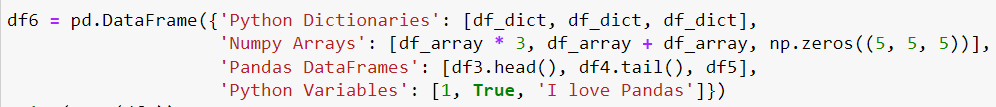
→ **pd.DataFrame(***data*, **index**=*nombre filas,* **columns**=*nombre columnas***)**

***\* Si no indicamos nombre a las columnas o filas tomará nombre de 0 en adelante***

****

****

****

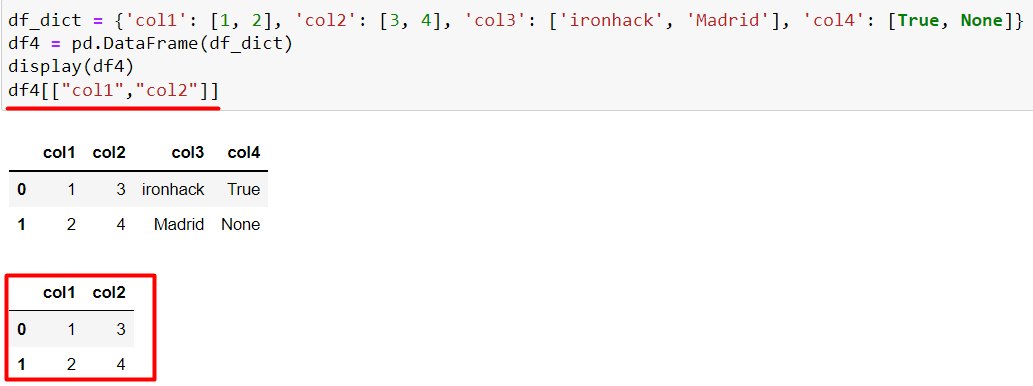
****

*\* También se les puede añadir el nombre de las columnas de la siguiente manera*





* ***EXTRACCIÓN, MODIFICACIÓN y CREACIÓN***:
  + EXTRACCIÓN: **nueva var = slicing** → **=** name\_dataFrame**[nom columna/s]**

****

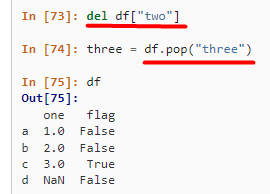
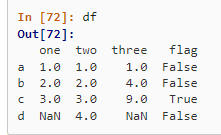
* + MODIFICACIÓN: **slicing = valor**

**→** name. dataFrame**[nom columna/s] = “new value”**

* + CRACCIÓN: **nuevo elemento del data frame = valor**

**→** name. dataFrame**[nom nueva columna/s] = “new value”**

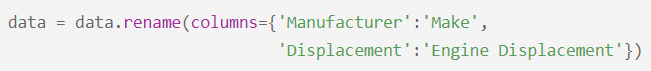
* ***ELIMINAR VALORES***:
  + **del** name\_dataFrame**[nom columna/s]**
  + name\_dataFrame.**pop(**name column**)**



* ***ATRIBUTOS***:
  + pd.**set\_option(**'display.max\_rows', 1000**)** → te permite cambiar opciones, en este caso hacer que para display() se muestren 1000 filas



* + dataFrame**.size** → Nº de elementos
  + dataFrame**.shape** → (FilasxColumnas)
  + dataFrame**.ndim** → Dimensiones (1)
  + dataFrame**.dtypes** → Tipo de sus valores (int, str, object, etc)
  + **.axes** → ejes
  + dataFrame**.memory\_usage()** → bytes
  + dataFrame**.describe()** → Te da INFORMACIÓN ÚTIL (count, mean, sd, mean, ...)
  + dataFrame**.info()** → Te da INFORMACIÓN como el describe()
  + dataFrame**.head()** → Muestra los PRIMEROS valores de la serie
  + dataFrame**.tail()** → Muestra los ÚLTIMOS vores de la serie
  + **display(**dataFrame**)** → Muestra TODOS valores de la serie
  + dataFrame**.columns** → crea o devuelve columnas
    - CREAR: dataFrame.columns = [“nombre1”, “nombre2”, “nombre3”]
    - EXTRAER: dataFrame.columns
  + dataFrame**.index** → crea o devuelve nombre de las filas
    - CREAR: dataFrame.index = [“nombre1”, “nombre2”, “nombre3”]
    - EXTRAER: dataFrame.index
  + dataFrame**.rename(**columns={old\_value:new\_value}**)** → cambia el nombre de las columnas o filas(index). Hay que equivaler el nombre antiguo al nuevo

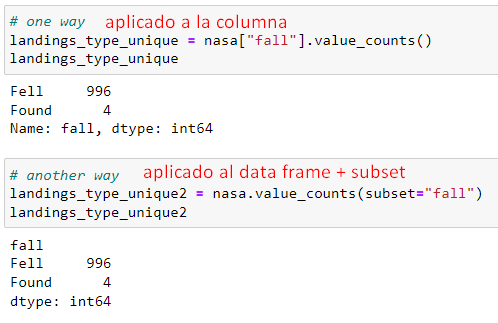


* + dataFrame**.T** → TRANSPONE la tabla

**METHODS(): *pd.*** *ò* ***name\_dataFrame.***

Los methods se pueden aplicar a:

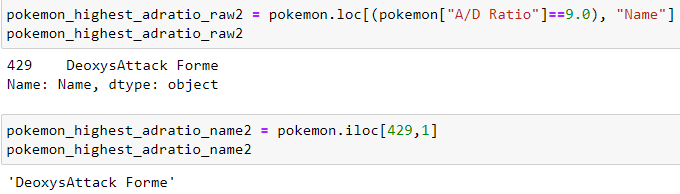
* **DF: al data frame entero** y dentro del method indicamos columna, fila o celda
* **SUBSET**: a la **columna, fila** o **celda** directamente

******

* dataFrame**.loc[**nombre fila, nombre columna**] →** es una forma de **extraer(slicing) valores** de la tabla.



TAMBIÉN, en vez de “nombre fila” se pueden poner valores específicos a través de CONDICIONES, ejemplo (valores columna >0)



**\* BUSCAR y EXTRAER**

**→ new var =** data\_frame**.loc[**nombre fila, nombre columna**]**

**\* BUSCAR y TRANSFORMAR columna existente**

**→** data\_frame**.loc[**nombre fila(posible condición), nombre columna**] = nuevo valor**

**\* BUSCAR y CREAR nueva columna**

**→** data\_frame**.loc[**nombre fila(posible condición), NUEVA columna**] = nuevo valor**

* dataFrame**.iloc[**index fila, index columna**] →** igual pero 100% posicional (a través de los index de fila y columna)



* dataFrame**.transpose() →** transpone
* **pd.read\_csv(**"../data/PRSA\_data\_2010.1.1-2014.12.31.csv"**) →**
* dataFrame**.value\_counts()** → la frecuencia con la que se repite un valor (nº veces). Te genera una SERIE
* dataFrame**.unique()** → te devuelve todos los valores únicos (valores deduplicados). Te genera una LISTA
* dataFrame**.sort\_values(by=**[“columnas”],**ascending=**False**) =** .sort\_values(**by**=[“nombre”, “apellidos”, “dinero”], **ascending**=[True, True, False])

→ **by:** te ordena la tabla en función de la columna que le indiques. Podemos usar un listado para ordenar por columnas en orden (primero en base a una columna, después en base a la siguiente,etc)

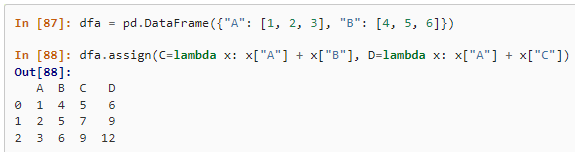
→ **ascending** = por defecto va en orden ascendente, si lo queremos cambiar tendríamos que poner “**ascending=false**” o valores booleanos.

* **pd.from\_dict(**data, **orient**= /index /columns**)**  **→**  si utilizamos orient = “index”, los nombres de las “columnas” en el dic pasarán a ser filas



* dataFrame**.sum() →** suma
* dataFrame**.mean() →** media
* dataFrame**.median() →** mediana
* dataFrame**.max() →** valor máximo
* dataFrame**.min() →** valor mínimo
* dataFrame**.assign(**name column = values**) → crea NUEVAS COLUMNAS**

**(**te permite **OPERAR**)

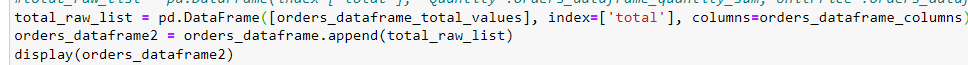
****

* dataFrame**.append: → crea NUEVAS FILAS (**te permite **OPERAR**)

1º creas un nuevo DataFrame de una sola línea (con los mismos nombres de columnas pero un nuevo nombre de linea

→ pd.DataFrame([orders\_dataframe\_total\_values], **index=**['total'], **columns=***orders\_dataframe\_columns*)

2º dataFrame**.append**



1. dataFrame**.append** (un nuevo “dic” con **ignore\_index=True**)

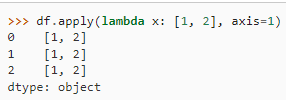


## **.apply() + lambda**

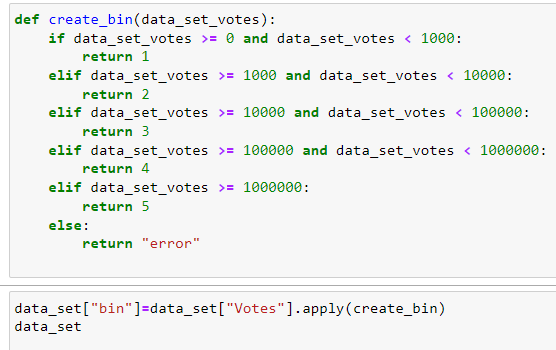
Aplica una función a lo largo de todo el axis=0(filas) o axis=1(columnas). Lo podemos aplicar sobre una columna, sobre una fila, sobre el data set entero, etc

1. **apply with lambda/function**

→ DataFrame**.apply(**func, axis=1, raw=false, (args)**)**



→ si vamos a aplicarlo solo sobre 1 columna o fila mejor NO utilizar lambda



1. **apply with lambda + function**

***def func****(****arg1****,* ***arg2****):*

***return*** *arg1 x arg2*

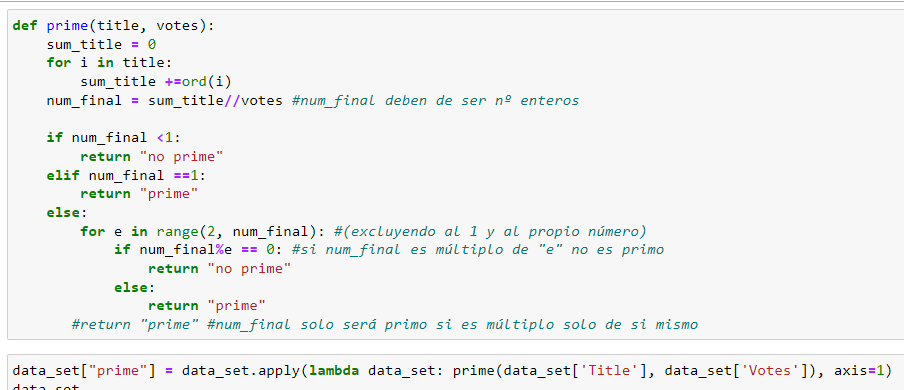
→DataFrame**.apply(lambda** DataFrame: **func**(***arg1***,***arg2***), axis=1**)**

→DataFrame**.apply(lambda** x: **func**(***arg1***,***arg2***), axis=1**)**

* ***arg1*** = x['col1'] y ***arg2*** = x['col2']

=DataFrame**.apply(lambda** DataFrame: **func**(DataFrame['col1'], DataFrame['col2']), axis=1**)**

=DataFrame**.apply(lambda** x: **func**(x['col1'], x['col2']), axis=1**)**

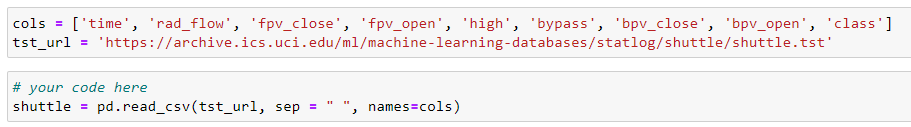


## **Importing and Exporting Data**

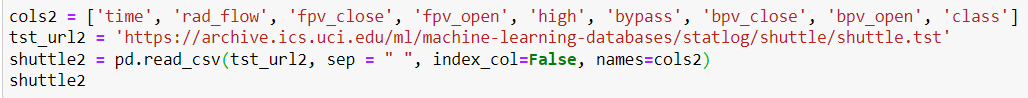
1. **CSV:** → **pd.read\_csv()** y **.to\_csv()** con param **“sep”, “names”** (si el csv no tiene nombre columnas se lo podemos dar con este parámetro)**,index\_col=False** (si el csv no tiene index, cogerá la primera columna como index, lo que estaría mal, si usamos “index\_col=False” te creará un index de “0” en adelante)
   1. **IMPORTAR**:

desde local: new\_data\_frame = **pd.read\_csv(**“file localization”, **sep=”,”, index\_col=False)**

desde una URL: new\_data\_frame = **pd.read\_csv(**“file URL”, **sep=”,”, index\_col=False)**

****

\* con **names=...**

****

* 1. **EXPORTAR**:

new\_data\_frame**.to\_csv(**“name\_file”, sep=”**,**”, **index=False)**

* + 1. *index=True* → exportamos el archivo con el index (nombre de las filas)
    2. *index=False* → exportamos el archivo sin el index (sin el nombre de las filas)

1. **EXCEL**: → **pd.read\_excel()** y **.to\_excel()**
   1. **instalar la librería** (pandas ya tendría que estar instalado)

!conda install **-y xlrd**

!conda install **-y openpyxl**

* 1. **IMPORTAR**: new\_data\_frame = **pd.read\_excel(**“file localization”**)**
  2. **EXPORTAR**: new\_data\_frame**.to\_excel(**“file localization”, index=False**)**
     1. *index=True* → exportamos el archivo con el index (nombre de las filas)
     2. *index=False* → exportamos el archivo sin el index (sin el nombre de las filas)

1. **JSON:** → **pd.read\_json()** y **.to\_csjson()** con param **“orient”**
   1. **IMPORTAR**: new\_data\_frame = **pd.read\_json(**“file localization”, **orient=”**records**”)**
   2. **EXPORTAR**: new\_data\_frame**.to\_json(**“file localization”, **orient=”**records**”)**
      1. records: our JSON file was structured as a list of dictionaries where each dictionary represented a complete record of data
      2. 'split': Flatten Dic que contiene indexes, columns, y data.
      3. 'index': Nested Dic anidados que contienen {índice: {columna: valor}}.
      4. 'columns': Nested Dic anidados que contienen {columna: {índice: valor}}
      5. 'values': Nested List anidada donde cada sublista contiene los valores de un registro.
      6. 'table': Nested Dic anidados que contienen esquemas y datos (registros).
2. **PARQUET:** → **pd.read\_parquet()**
   1. **instalar la librería** (pandas ya tendría que estar instalado)

!conda install **-c conda-forge pyarrow**

* 1. **IMPORTAR**: new\_data\_frame = **pd.read\_parquet(**“file localization”**)**

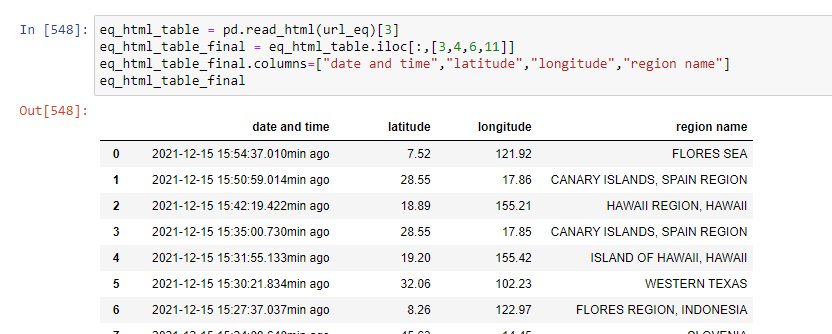
1. **DATA BASES:** → **pd.read\_sql\_query()** y **.to\_sql**
   1. **importar librerías**

* **import pymysql**
* **from** **sqlalchemy import create\_engine**
  1. **create\_engine function**: reemplazando username y password con el nombre de usuario y la contraseña para la base de datos MySQL en la máquina local.

→ engine = **create\_engine(**'mysql+pymysql://username:password@localhost/publications'**)**

* 1. **IMPORTAR**: **data = pd.read\_sql\_query('**SELECT \* FROM publications.employee', engine**)**
  2. **EXPORTAR**: data**.to\_sql**('employee2', engine, if\_exists='replace', index=False)

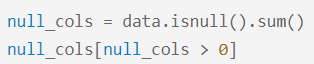
1. **read\_html → te lee las tablas (para el resto habra que utilizar el webscrapping**

****

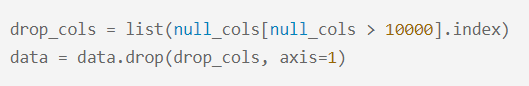
## **Data Cleaning**

**Examinar datos en busca de posibles problemas**

1. Valores faltantes:
   1. **DETECTAR**:
2. **.isnull()**
3. **.sum()**
4. ver las columnas en las que hay nulls más de 1 vez



* 1. **ELIMINAR (drop):** **eliminaremos** las **columnas** que tengan **muchos nulos** (EJEMPLO > 1000 NULOS) con **.drop(**datos,inplace=True, axis=1**)**

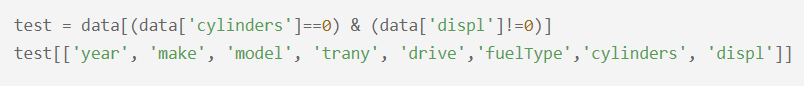


* + .index porque es una serie (y en una serie solo hay filas, sino, en un dataFrame index serían las filas)
  + Estamos diciendo que eliminen las columnas gracias a “axis=1”
  1. **RELLENAR (filling):** en vez de eliminar, lo rellenaremos **con valores a “0”**

**→ .fillna(**valor a rellenar**)**

****

1. Valores incorrectos: En este caso hacemos un proceso manual porque nos hemos fijado visualmente de que hay un error. En el ejemplo anterior, después de rellenar los valores nulos con “0”, vemos que sigue habiendo motores con cilindrada “0” pero con display no “0”.
2. vamos a encontrar los valores que queremos sustituir



1. sustituimos su valor de “0” cilindradas a “4” cilindradas

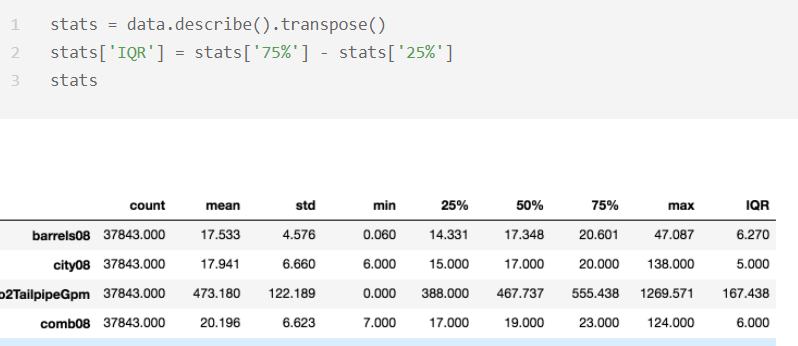


1. Valores Numéricos: Si queremos aplicar métodos estadísticos podemos extraer las columnas que solo tengan valores numéricos y aplicar la varianza, media, etc.

**→ .\_get\_numeric\_data(**valor a rellenar**)**

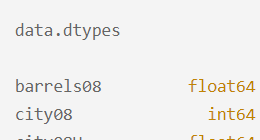
****

**→ .describe().transpose()** sobre una tabla numérica nos devolverá en columnas resultados estadísticos que podremos utilizar para analizar o limpiar



1. Corrección Data Types: **.dtypes()** y **astype()**

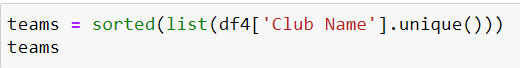
* **.dtypes() → muestra el tipo de datos**

****

* **.astype(**nuevo dataType**) →** cambia el tipo de datos

****

1. REEMPLAZAR strings y caracteres especiales:
2. ver los valores que hay (para eso veremos valores únicos y ordenados



1. **.str.replace(,,**regex=True**)** para cambiar lo que queramos



1. Eliminar DUPLICADOS: elimina duplicados.

* se lo podemos aplicar al dataset entero



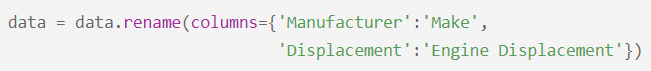
* o a un subset (si queremos analizar columnas o filas concretas)

**.drop\_duplicates(**subset=[“nom\_columns”], keep='first', ignore\_index=False**)**

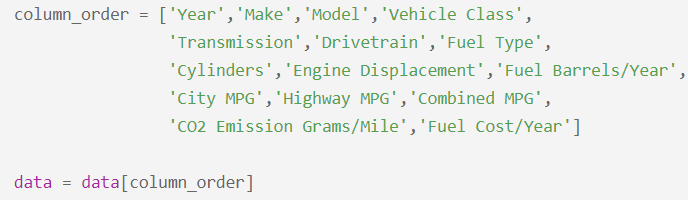
****

## **Data MANIPULATION**

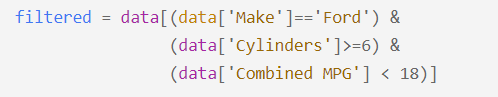
* dataFrame**.columns** → crea o devuelve columnas
  + CREAR: dataFrame.columns = [“nombre1”, “nombre2”, “nombre3”]
  + EXTRAER: dataFrame.columns
* dataFrame**.index** → crea o devuelve nombre de las filas
  + CREAR: dataFrame.index = [“nombre1”, “nombre2”, “nombre3”]
  + EXTRAER: dataFrame.index
* dataFrame**.rename(**columns={old\_value:new\_value}**)** → cambia el nombre de las columnas o filas(index). Hay que equivaler el nombre antiguo al nuevo



* **ORDENAR COLUMNAS**:
  + crear una lista que contenga los nombres de las columnas
  + Aplicarlo al dataFrame → **dataFrame[*column list*]**

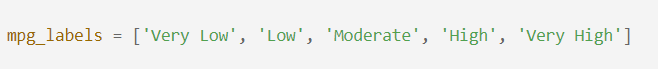


* **FILTRAR REGISTROS**: Al hacer slicing puedes poner MÚLTIPLES declaraciones condicionales
  + ( & ) → “AND” para múltiples condiciones
  + ( | ) → “OR” para múltiples condiciones



→ En este caso no queremos extraer unas columnas, como hicimos con los casos anteriores, sino crear un nuevo data frame (filtered) con todos los datos que cumplan las condiciones del slicing.

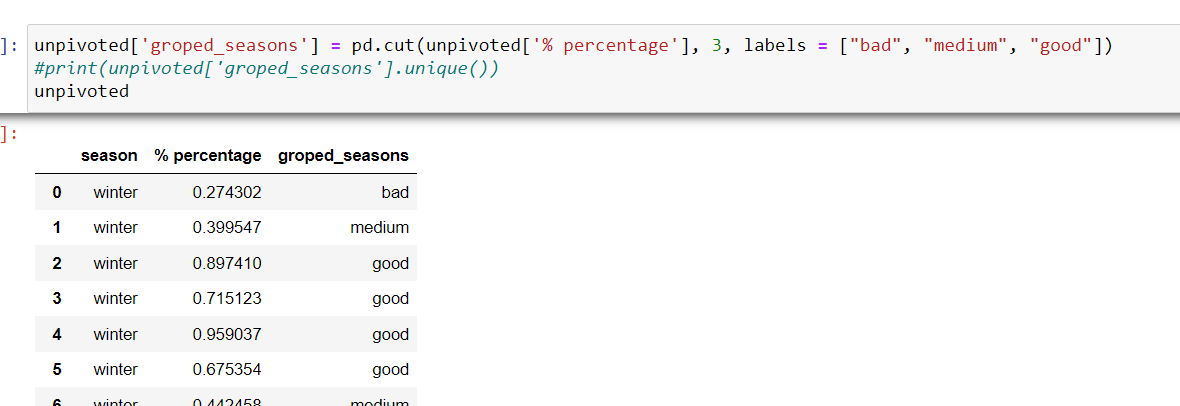
* **AGRUPAR variables NUMÉRICAS en CATEGÓRICAS**: Como crear rangos



* + POR RANGOS:
* *Equal width:* los rangos se dividen uniformemente no en base a que todos los rangos tengan el mismo número de valores, sino que **entre los rangos haya el mismo tamaño**.

→ bins = **pd.cut(**data['column name'],**Nºbins**, **labels=....)**

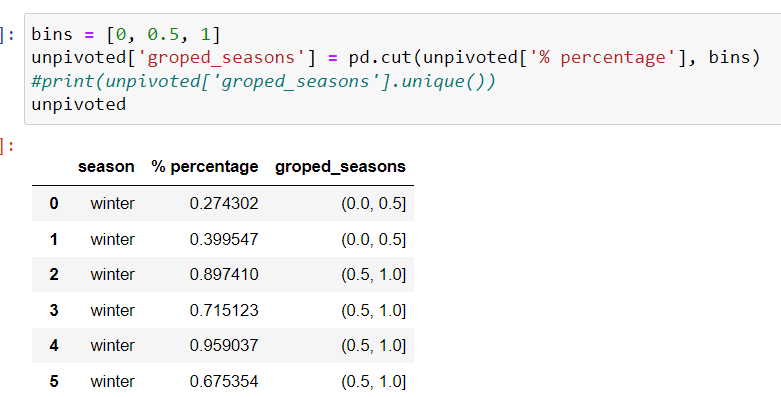
****

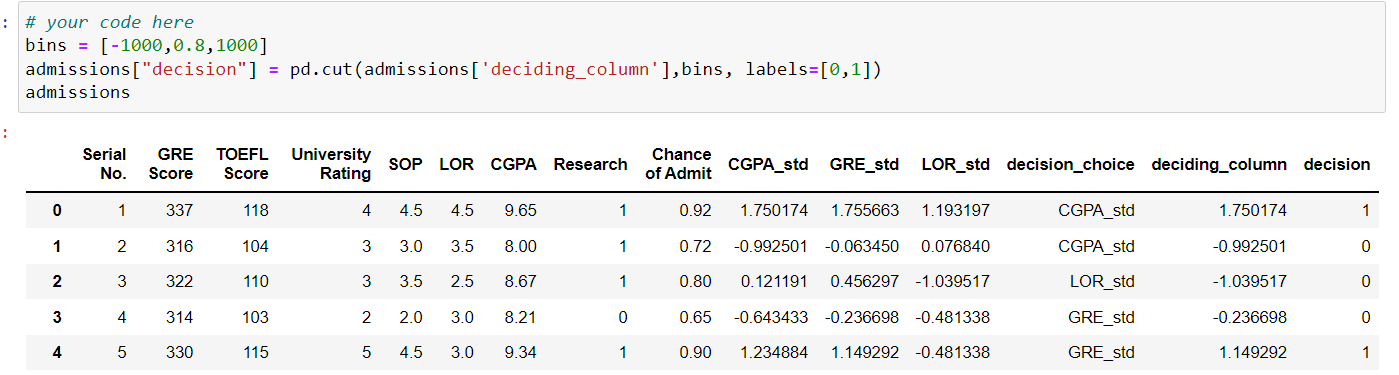
****

* *Custom-sized:* hay que **indicar los valores de corte de los rangos**.

→ bins = **pd.cut(**data['column name'],**Range values**, **labels=....)**

****

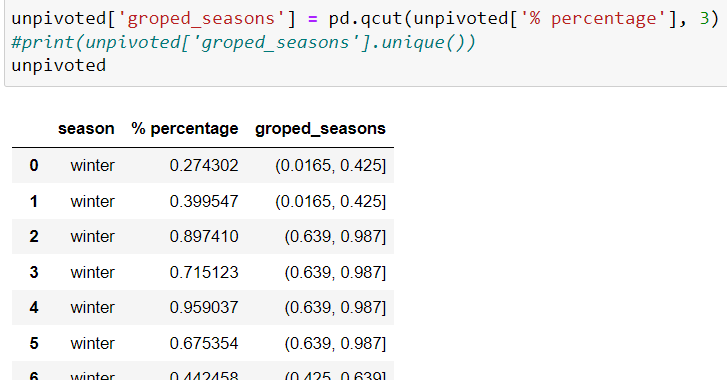
****

****

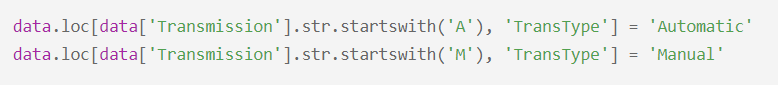
* + POR FREQUENCIA
* *Equal frequency:* **mismo número de valores** en todos los rangos.

→ bins = **pd.qcut(**data['column name'],**Nºcolums**, **labels=....)**

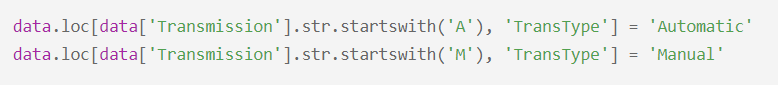
****

****

* **CATEGORÍAS CONDICIONALES**: Podemos crear columnas en base a condiciones gracias a diferentes methods. En el siguiente caso vamos a combinar **.loc()** y **.str.startswith(' ')** :
* MODIFICAR EL DF ENTERO



* CREAR NUEVAS COLUMNAS

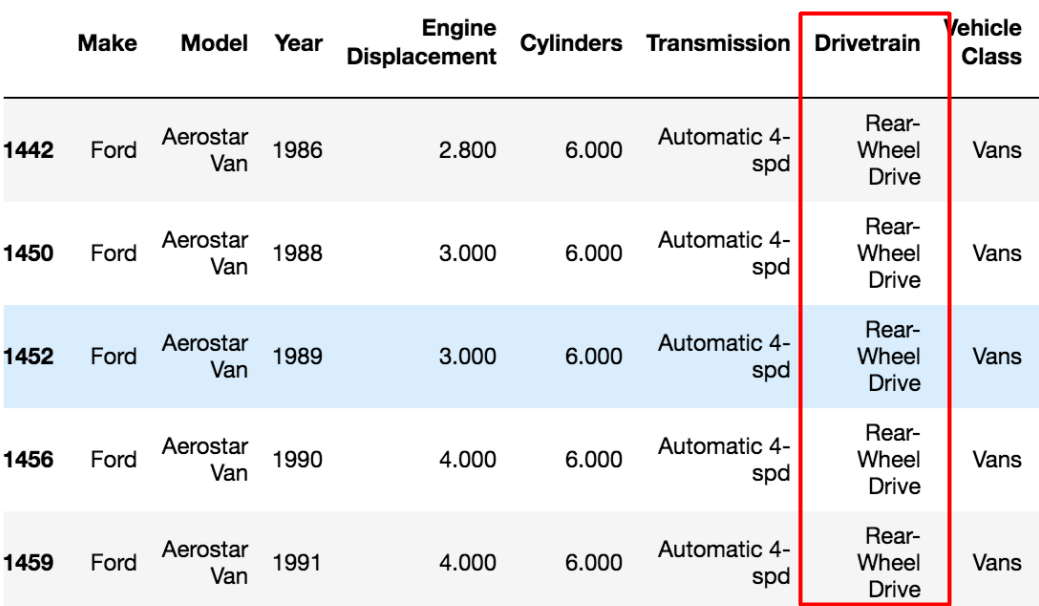


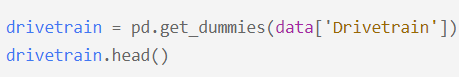
* **pd.get\_dummies(**data\_frame**, columns**=[...]**)**

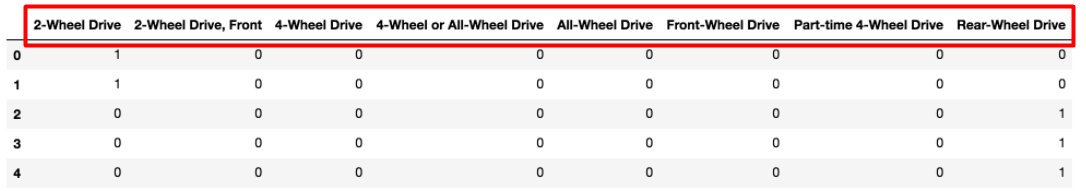
**ò**

**pd.get\_dummies(**data\_frame**[nombre columna])**

→ expande una sola columna categórica en tantas columnas booleanas como categorías haya

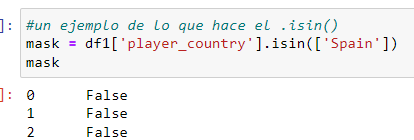






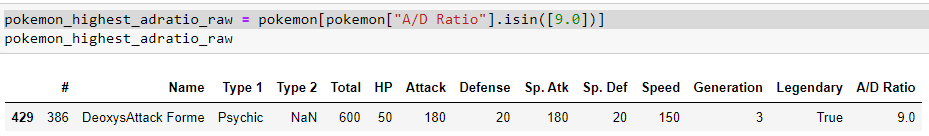
→ Los valores de las columnas son ahora nuevas columnas con valores de 1 y 0 para todos los registros del data frame original

* **.isin(**“valores”**)** → actúa como filtro o condicionante “is in”. Si se cumple la condición devuelve True, sino False

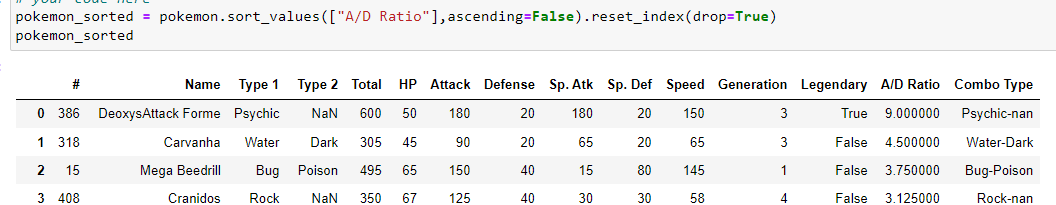


Aplicado al dataframe devolverá todos sus valores que cumplan con True:

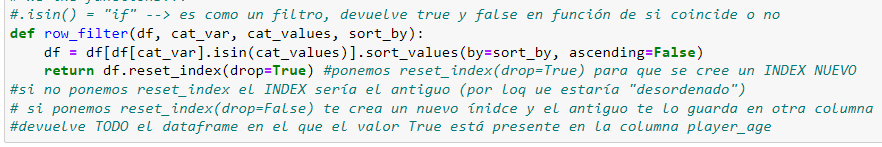




* **.reset\_index(drop=**True**)→ELIMINA el index antiguo** y pone el Default de nuevo
* drop=True → para que se cree un INDEX NUEVO.
* drop=False → DEFAULT, te crea INDEX NUEVO y el antiguo te lo guarda en otra columna



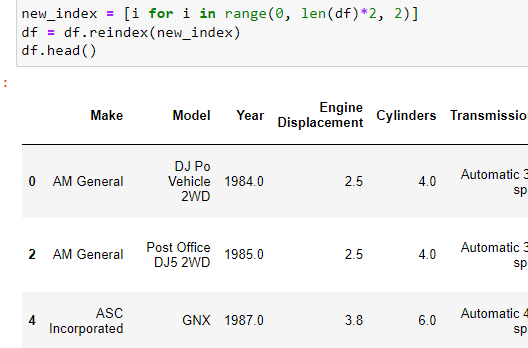
→ Para este caso, si no ponemos reset\_index el INDEX sería el antiguo pero desordenado.



* **.reindex(data)** → **CREA** un **index nuevo** a partir de NUEVOS VALORES.

Ponemos en “data” poner una lista o una Serie y hace que sus valores pasen a ser el index nuevo

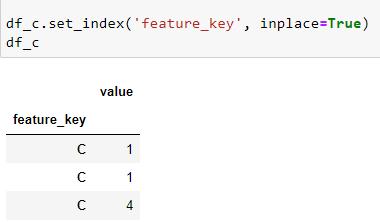
→ Con este method() EL ANTIGUO INDEX SE ELIMINA



* **.set\_index(**columna a elegir como index**, inplace=**True**)** → **CREA** un **index nuevo** a partir de UNA COLUMNA.

→ Con este method() podemos ELEGIR QUÉ HACER CON EL ANTIGUO INDEX

* inplace=True → te lo sustituye por el antiguo
* inplace=False → DEFAULT, crea uno nuevo y el antiguo lo guarda en una columna



* **.groupby(**valor a agrupar, **as\_index=**False**)** → te agrupa un DF según una columna, una fila, etc

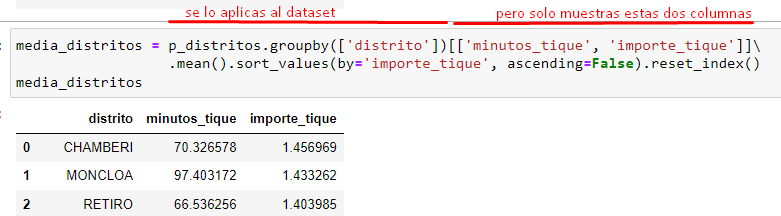
¡IMPORTANTE! necesita de un method a aplicar después, ya que groupby solo crea un objeto

* **as\_index=**True → el valor a agrupar pasa a ser el index
* **as\_index=**False →crea index

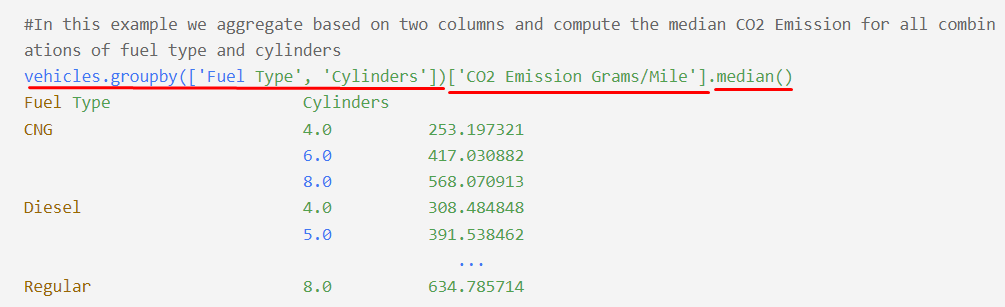




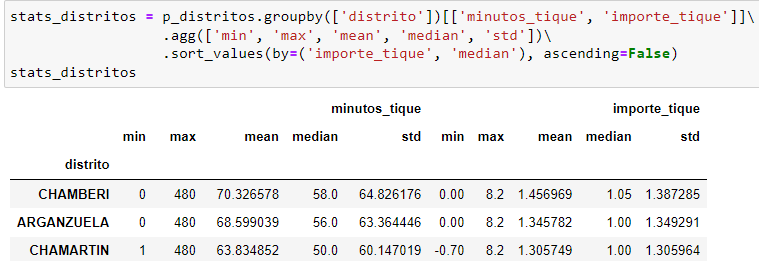
→ podemos **aplicarlo a unas columnas específicas** del data set



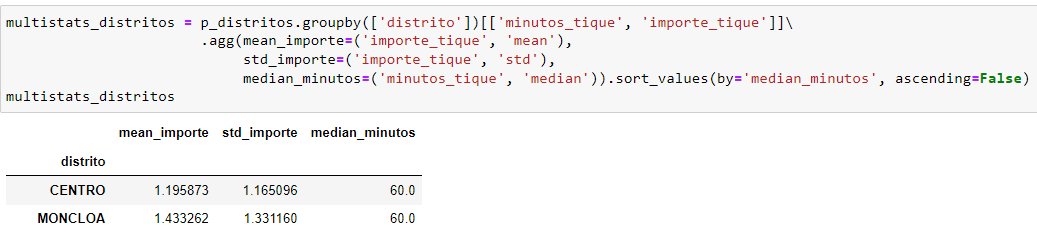
→ se puede **agrupar** en base a **más de una columna**



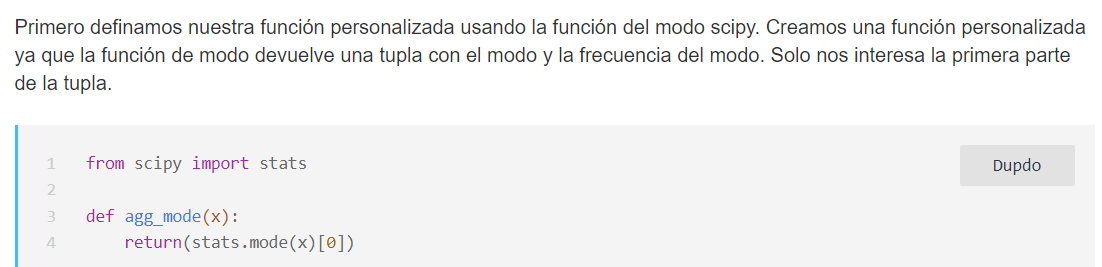
* **.agg( [**conjunto de methods**]** **)** → te permite aplicar methods a un dataset
* puedes aplicar **múltiples methods**

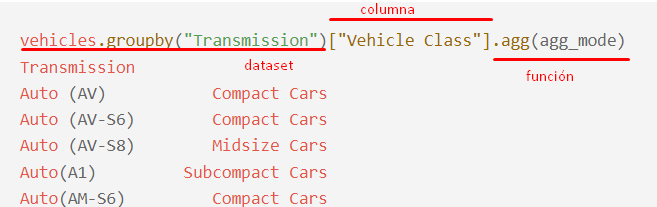


* puedes elegir **a qué columna aplicarlo** concretamente



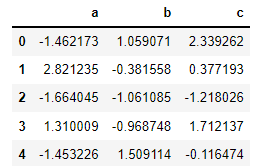
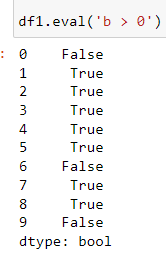
* puedes aplicar **funciones**





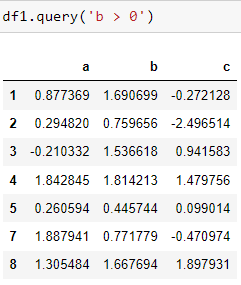
* **.eval( “expresion”)** → evalúa la condición y devuelve TRUE o FALSE

Se admiten **+, -, \*, /, \*\*, %, //, |(o), &(y)** y **~(no).**

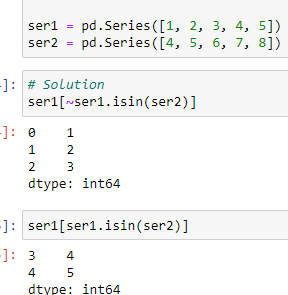
** **

* **.query( “expresion”)** → evalúa la condición y devuelve LOS VALORES que la cumplan

Se admiten **+, -, \*, /, \*\*, %, //, |(o), &(y)** y **~(no).**

****

* **“~”** → significa lo contrario (como “!” en javascript)



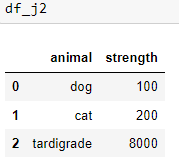
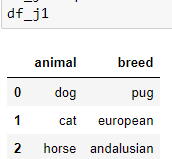
→ En el siguiente ejemplo si “isin” es que coincida, ahora será **que no coincida**

### **pd.merge()** → FUSIONA 2 columnas, a partir de un PRIMARY KEY ([link](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.merge.html))

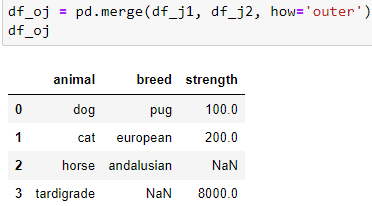
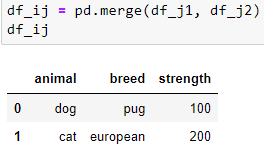
[LINK LAB](https://colab.research.google.com/drive/10uSl3s7RFXwok_Be04YJNmQX69KTbzTc)

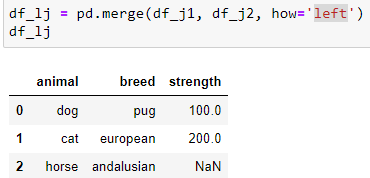
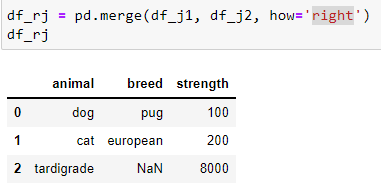
**pd.merge(** col 1, col 2, **how='**xxxxx**', on='**xxxxx**')**

→ Si hay varios Primary Keys con **“on” eliges con que unir la tabla**



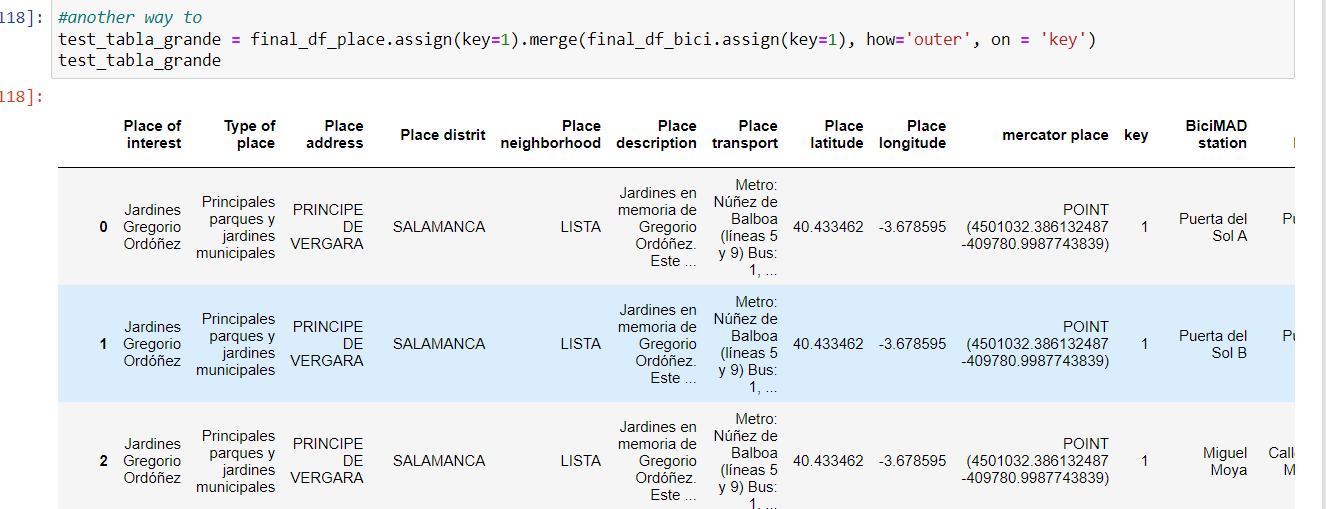
* Inner JOIN: **pd.merge(** col 1, col 2 **)** → FUSIONA donde hay valores
* Outer JOIN: **pd.merge(** col 1, col 2, **how=**'outer'**)** → FUSIONA todo
* Left JOIN: **pd.merge(** col 1, col 2, **how=**'left'**)** → Col 1 como PRIMARY KEY
* Right JOIN: **pd.merge(** col 1, col 2, **how=**'right'**)** → Col 2 como PRIMARY KEY

****

** **

* forma con **“outer”** y **.assign(key=x)** que hace un merge entre dos tablas y une todas las posibilidades, al ambas tener la misma key.

*(también podríamos hacer pd.merge(nombre tabla1, nombre tabla2, on = “key” how=”outer”), pero esto solo si las tablas ya tienen esa columna con el mismo key)*

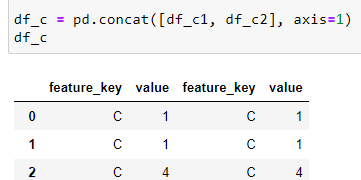
****

### **pd.concat(objs, axis**=0**, join=**'outer**', ignore\_index=**False**, keys=**None**)** [link](https://pandas.pydata.org/docs/reference/api/pandas.concat.html)

### → añade un dataframe debajo de otro (es decir sus filas) (**axis=0**)

### **→** añade unas columnas a la derecha o izquierda (**axis=1**)

**pd.concat( [** col 1, col 2 **]**, **axis=0)**

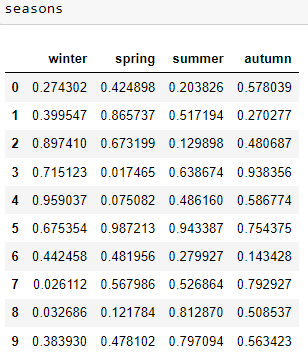


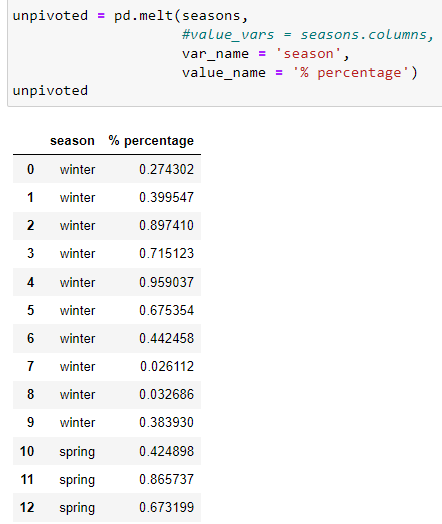
### **pd.melt()** → ***melt = unpivoted →*** *pasar las columnas a filas*

*→ útil cuando queremos crear gráficos de barras, líneas, etc*

*→ TE LO PASA A DOS EJES: X, Y*

***pandas.melt(frame, id\_vars=****None****, value\_vars=****None****, var\_name=****None****, value\_name=****'value'****, col\_level=****None****, ignore\_index=****True****)***

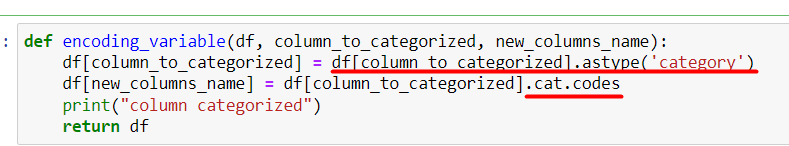
**

**

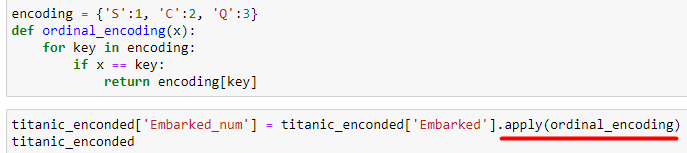
**PASAR DE VAR, CATEGÓRICA A NUMÉRICA**: a cada valor un número (problema, le estaríamos asignando un peso diferente a cada valor (a mayor número más peso → Si quieres **PONDERAR usa esto**)

→column**.astype(“category”)**

→column**.cat.codes**

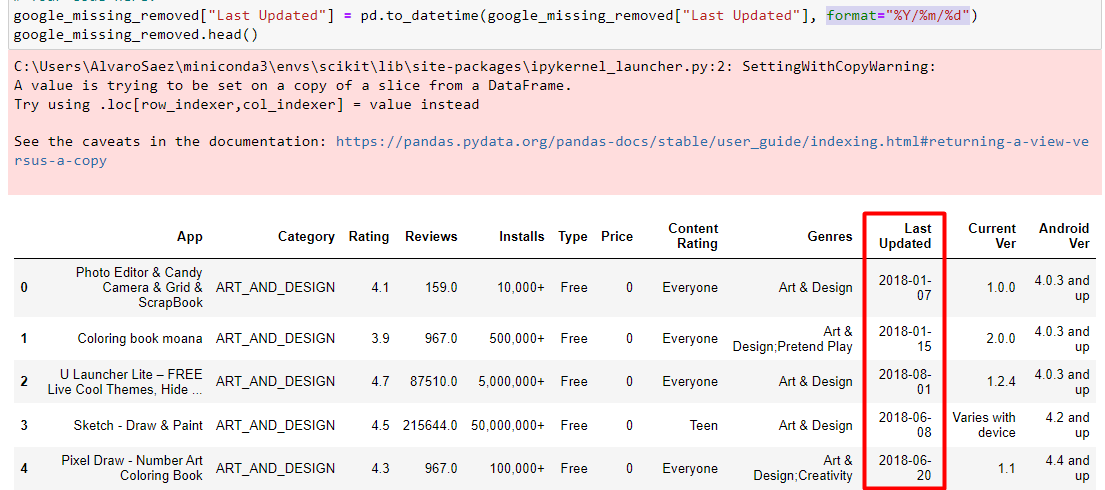
****

Ò crear un **dic**, **iterar** y **aplicarlo** al df

****

**to\_datetime** [**link**](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.to_datetime.html)

**pd.to\_datetime(**arg**,format="**%Y/%m/%d**")**

****