In [10]:
```python
import pandas as pd
import numpy as np
```

DATA TRANSFORMATION

Removing duplicates

Se puede dar el caso de que queramos eliminar aquellas columnas duplicadas, para ello disponemos del siguiente método

df.duplicated() --> Devuelve un Booleano en el que las filas repetidas toman el valor True

Partiendo del siguiente df:

In [11]:
```python
data=pd.DataFrame({'k1':['one']*3+['two']*4,'k2':[1,1,2,3,3,4,4]})
data
```

Out[11]:

	k1	k2
0	one	1
1	one	1
2	one	2
3	two	3
4	two	3
5	two	4
6	two	4

In [12]:
```python
data.duplicated()
```

Out[12]:
```
0    False
1     True
2    False
3    False
4     True
5    False
6     True
dtype: bool
```

df.drop_duplicates() --> Elimina aquellas filas duplicadas, manteniendo por defecto la primera de las duplicadas. Por defecto, solo elimina aquellas filas que son iguales EN TODAS SUS COLUMNAS, pero se puede especificar que columna/s queremos que mire.

Partiendo del dataframe anterior:

In [13]:
```python
data.drop_duplicates()
```

Out[13]:

	k1	k2
0	one	1
2	one	2
3	two	3
5	two	4

Si por ejemplo solo queremos que mire si hay datos duplicados en la columna 'k1', se lo especificamos:

In [14]:
```python
data.drop_duplicates(['k1'])
```

Out[14]:

	k1	k2
0	one	1
3	two	3

Por defecto, se queda con la primera fila de las repetidas, pero podemos indicarle que se quede con la última con **df.drop_duplicates(take_last=True)**:

In [15]:
```python
data.drop_duplicates(take_last=True)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-c00817e97961> in <module>
----> 1 data.drop_duplicates(take_last=True)

TypeError: drop_duplicates() got an unexpected keyword argument 'take_last'
```

Mapping transformation

df['columna_donde_tomar_dato].map(diccionario/funcion) --> Coge los valores de la columna que especificas a la izquierda del punto y busca en el diccionario el valor correspondiente

Partiendo del siguiente dataframe:

In [16]:
```python
data=pd.DataFrame({'food':['bacon','pulled pork','bacon','Pastrami','corned beef','Bacon','pastrami','honey ham','nova lox'],'ounces':[4,3,2,6,7.5,8.,3,5,6]})
data
```

Out[16]:

	food	ounces
0	bacon	4.0
1	pulled pork	3.0
2	bacon	2.0
3	Pastrami	6.0
4	corned beef	7.5
5	Bacon	8.0
6	pastrami	3.0
7	honey ham	5.0
8	nova lox	6.0

Queremos crear una nueva columna llamada **'animal'** que nos diga a qué animal pertenece cada carne, para ello creamos un diccionario con las equivalencias:

In [17]:
```python
meat_to_animal={'bacon':'pig','pulled pork':'pig','pastrami':'cow','corned beef':'cow','honey ham':'pig','nova lox':'salmon'}
meat_to_animal
```

Out[17]:
```
{'bacon': 'pig',
 'pulled pork': 'pig',
 'pastrami': 'cow',
 'corned beef': 'cow',
 'honey ham': 'pig',
 'nova lox': 'salmon'}
```

Por último, usamos df[].map() y ponemos la columna 'food' en minúscula:

In [18]:
```python
data['food']=data['food'].str.lower()  # .str.lower() permite ponerlo en minus

data['animal']=data['food'].map(meat_to_animal)
data
```

Out[18]:

	food	ounces	animal
0	bacon	4.0	pig
1	pulled pork	3.0	pig
2	bacon	2.0	pig
3	pastrami	6.0	cow
4	corned beef	7.5	cow
5	bacon	8.0	pig
6	pastrami	3.0	cow
7	honey ham	5.0	pig
8	nova lox	6.0	salmon

Port tanto, la sintaxis sería: **dataframe['columna_donde_coger_los_datos].map(diccionario_donde_estan_las equivalencias)**

También podemos usar una función con .map(lambda X: operaciones)

Replacing values

df.replace(['valor_que_se_sustituye_1','valor_que_se_sustituye_2',...],['valor_que_sustituye_a_1,valor_que_sustituye_a_2,...]) --> Cambia los valores de la primera lista por los de la segunda en toda la tabla (puedes por ejemplo poner df['columna'].replace() y así solo cambia los de una columna, etc..)

Se aplica tanto a series como a dataframe, por ejemplo si tenemos las siguiente serie y dataframe:

In [19]:
```python
serie1=pd.Series([1,-999,2,-999,-1000,3])
df=pd.DataFrame({'food':['bacon','pulled pork','bacon','Pastrami','corned beef','Bacon','pastrami','honey ham','nova lox'],'ounces':[4,3,2,6,7.5,8.,3,5,6]})
```

In [21]:
```python
serie1.replace(-999,np.nan)
```

Out[21]:
```
0       1.0
1       NaN
2       2.0
3       NaN
4   -1000.0
5       3.0
dtype: float64
```

In [27]:
```python
df.replace(['bacon','pulled pork'],['CERDO','CERDO MECHADO'])
```

Out[27]:

	food	ounces
0	CERDO	4.0
1	CERDO MECHADO	3.0
2	CERDO	2.0
3	Pastrami	6.0
4	corned beef	7.5
5	Bacon	8.0
6	pastrami	3.0
7	honey ham	5.0
8	nova lox	6.0

Discretization and binning

Discretizar datos consiste en incluir un juego de datos en distintos intervalos, por ejemplo un rango de edades. Para ello, tenemos la herramienta **cut**, que nos permite crear un elemento denominado **objeto categórico** que permite esta discretización.

Para ello necesitamos 2 o 3 elementos, como mínimo 2:

- Lista/serie de datos --> Constituyen los datos que queremos discretizar. Si partimos de una serie, se creará un objeto categórico de una serie por lo que es recomendable partir de una lista o hacer serie.values() antes.
- Lista de intervalos --> Ponemos en una lista la frontera de los intervalos, y pandas solito creará los intervalos, por defecto (-], con **right=False** se invierte.
- Serie de labels (opcional) --> Se especifica como **labels=[.......]** y permite atribuirle un nombre a los intervalos, en vez de enseñar el intervalo.

In [39]:
```python
datos=[20,22,25,27,21,23,37,31,61,45,41,78] #Datos a discretizar

intervalos=[18,25,35,60,100] #Intervalos deseados

obj_cat=pd.cut(datos,intervalos)
obj_cat
```

Out[39]:
```
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100], (35, 60], (35, 60], (60, 100]]
Length: 12
Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]
```

Como podemos ver, se ha incluido cada dato en su intervalo correspondiente, y podemos ver que se ha creado una nueva entidad denominada **objeto categórico**, en el cual podemos consultar sus **codes** y sus **categories**

objeto_categorico.codes --> Muestra los codes del onjeto categórico, es decir, el valor de 0 a X del intervalo al que pertenece el valor. Por ejemplo si el primer elemento entra en el primer intervalo tendrá valor 0, si el segundo valor entra en el quitn ointervalo tendrá valor 4, etc...

objeto_categórico.categories --> Muestra los intervalos en los que has dividido el objeto categórico

pd.value_counts(objeto_categórico) --> Te muestra los intervalos y el número de elementos de la lista que entra en cada uno de estos intervalos

In [43]:
```python
obj_cat.codes
```

Out[43]:
```
array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 3], dtype=int8)
```

In [44]:
```python
obj_cat.categories
```

Out[44]:
```
IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]],
              closed='right',
              dtype='interval[int64]')
```

In [46]:
```python
pd.value_counts(obj_cat)
```

Out[46]:
```
(18, 25]     5
(35, 60]     3
(60, 100]    2
(25, 35]     2
dtype: int64
```

Podemos hacer que los intervalos se apliquen al revés:

In [47]:
```python
obj_cat_2=pd.cut(datos,intervalos, right=False)
obj_cat_2
```

Out[47]:
```
[[18, 25), [18, 25), [25, 35), [25, 35), [18, 25), ..., [25, 35), [60, 100), [35, 60), [35, 60), [60, 100)]
Length: 12
Categories (4, interval[int64]): [[18, 25) < [25, 35) < [35, 60) < [60, 100)]
```

pd.cut(valores, intervalos, labels=['nombre_intervalo_1',nombre_intervalo_2'...])

labels=[ ] permite generar un **mote** a cada intervalo, que se mostrará en el output. Por ejemplo, podemos crear un objeto categórico con los mismo elementos que el anterior, pero que en vez de ver los intervalos veamos un mensaje como 'Young', 'old', 'middle-age'...

In [49]:
```python
datos=[20,22,25,27,21,23,37,31,61,45,41,78] #Datos a discretizar

intervalos=[18,25,35,60,100] #Intervalos deseados

labels=['Youth','YoungAdult','MiddleAged','Senior'] #Etiquetas de los intervalos

obj_cat_2=pd.cut(datos,intervalos, labels=labels)
obj_cat_2
```

Out[49]:
```
['Youth', 'Youth', 'Youth', 'YoungAdult', 'Youth', ..., 'YoungAdult', 'Senior', 'MiddleAged', 'MiddleAged', 'Senior']
Length: 12
Categories (4, object): ['Youth' < 'YoungAdult' < 'MiddleAged' < 'Senior']
```

También podemos indicar el número de intervalos en el que queremos dividir la lista de datos y la precisión de dicha división, en vez de meter nosotros los intervalos

In [52]:
```python
datos=[20,22,25,27,21,23,37,31,61,45,41,78] #Datos a discretizar

obj_cat_3=pd.cut(datos,4, precision=2)
obj_cat_3
```

Out[52]:
```
[(19.94, 34.5], (19.94, 34.5], (19.94, 34.5], (19.94, 34.5], (19.94, 34.5], ..., (19.94, 34.5], (49.0, 63.5], (34.5, 49.0], (34.5, 49.0], (63.5, 78.0]]
Length: 12
Categories (4, interval[float64]): [(19.94, 34.5] < (34.5, 49.0] < (49.0, 63.5] < (63.5, 78.0]]
```

Podemos ver que se han creado 4 intervalos de igual longitud en función del dato de mayor y menor valor

In [59]:
```python
pd.value_counts(obj_cat_3)
```

Out[59]:
```
(19.94, 34.5]    7
(34.5, 49.0]     3
(63.5, 78.0]     1
(49.0, 63.5]     1
dtype: int64
```

Como cut hace la division de los intervalos en función del mayor y menor valor de la lista que se aporta como input, puede ser que esa separación no sea útil, por lo que tenemos el comando **qcut**, que hace los mismo que cut pero en vez de hacer la division por los valores lo hace por **el valor de los cuantiles del set de datos**, permitiendo una discretización más fiel:

pd.qcut(datos, X) --> Divide el set de datos en X intervalos según los cuantiles

In [61]:
```python
datos=[20,22,25,27,21,23,37,31,61,45,41,78] #Datos a discretizar


obj_cat_4=pd.qcut(datos,4)
obj_cat_4
```

Out[61]:
```
[(19.999, 22.75], (19.999, 22.75], (22.75, 29.0], (22.75, 29.0], (19.999, 22.75], ..., (29.0, 42.0], (42.0, 78.0], (42.0, 78.0], (29.0, 42.0], (42.0, 78.0]]
Length: 12
Categories (4, interval[float64]): [(19.999, 22.75] < (22.75, 29.0] < (29.0, 42.0] < (42.0, 78.0]]
```

In [63]:
```python
pd.value_counts(obj_cat_4)
```

Out[63]:
```
(42.0, 78.0]       3
(29.0, 42.0]       3
(22.75, 29.0]      3
(19.999, 22.75]    3
dtype: int64
```

df.describe() --> Te hace un dataframe con el máx, min, media, conteo... de cada columna. MUY UTIL

In [5]:
```python
data = pd.DataFrame(np.random.randn(1000, 4))
data.describe()
```

Out[5]:

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.033525	0.019540	-0.010918	0.018824
std	0.963619	1.012333	1.039636	1.022737
min	-3.196769	-3.336273	-3.610035	-3.236457
25%	-0.635529	-0.703857	-0.695737	-0.669419
50%	0.047496	0.052643	-0.000858	0.033643
75%	0.676527	0.674735	0.703200	0.753825
max	3.170322	3.213130	3.322277	3.081167

MUY UTIL: Mostrar las filas/columnas que dan true con .any()

Con **.any() posterior a una expresión que nos da un booleano**, se imprimarán aquellas (por defecto) columnas que den true. Por ejemplo:

In [10]:
```python
data.isnull().any()
```

Out[10]:
```
0    False
1    False
2    False
3    False
dtype: bool
```

Vemos que como no hay ningún valor nulo en ninguna columna, se muestran las columnas y false porque .isnull() ha generado un booleano lleno de false.

Con .any(axis=1) lo que devuelve son las filas que correspondan al True del booleano que lo precede. Por ejemplo:

In [12]:
```python
data.isnull().any(axis=1)
```

Out[12]:
```
0      False
1      False
2      False
3      False
4      False
       ...
995    False
996    False
997    False
998    False
999    False
Length: 1000, dtype: bool
```

Vemos que como no hay ningún valor nulo, any devuelve false en todas las filas.

INTERESANTE: COMO MOSTRAR LAS FILAS QUE TIENEN UN NAN EN ALGUNA COLUMNA

In [18]:
```python
exam_data = {'name': ['Anastasia', 'Dima', 'Katherine', 'James', 'Emily', 'Michael', 'Matthew', 'Laura', 'Kevin', 'Jonas'],
'score': [12.5, 9, 16.5, np.nan, 9, 20, 14.5, np.nan, 8, 19],
'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
'qualify': ['yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no', 'no', 'yes']}
labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

tabla=pd.DataFrame(exam_data,index=labels)

tabla.loc[tabla.isnull().any(axis=1)] #Solo aparecen las líneas que tienen un NaN.
```

Out[18]:

	name	score	attempts	qualify
d	James	NaN	3	no
h	Laura	NaN	1	no

Detecting and filtering Outliers

Podemos usar **.any()** para filtrar outliers, pero también podemos optar por las selecciones que se hacen con las operaciones de selección de datos como las del tema 3 o la que muestro aquí abajo:

IMPOTANTE Y UTIL: FILTRAR VALORES EN TODO EL DATAFRAME A LA VEZ (EL SIGUIENTE COMANDO MUESTRA TODAS LAS FILAS QUE TIENEN AL MENOS UN MAYOR DE 3 O MENOR DE -3)

In [7]:
```python
data[(data.values > 3) | (data.values < -3)]
```

Out[7]:

	0	1	2	3
128	0.942153	-3.152529	-1.645313	-1.238682
146	-0.975323	0.155506	3.183247	1.146679
197	-1.336702	1.084280	3.039936	0.753445
549	0.000535	-3.336273	2.275924	-0.603542
630	0.361533	1.453583	0.276974	-3.236457
631	3.170322	-0.358504	0.903972	0.079699
682	0.855815	-0.603597	-3.320775	-1.082348
711	1.718556	0.423623	0.262085	3.081167
769	0.117802	-0.507364	-3.610035	-1.590043
770	-0.037498	-0.520408	3.322277	-0.044292
778	1.041005	3.213130	-0.092797	1.460916
821	-3.196769	1.970020	0.379262	0.931079

(condicion).any() --> .any() muestra las filas que cumplen la condicion previa, debe de tener un BOOLEAN DATAFRAME como input. Podemos usar .any(axis=1) para que filtre por columnas y no por filas

DAR LA VUELTA A LAS FILAS O COLUMNAS

In [1]:
```python
df.loc[:,::-1] #Dar la vuelta a las columnas

df.loc[::-1]   #Dar la vuelta a las filas
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-7f9f3005be8a> in <module>
----> 1 df.loc[:,::-1] #Dar la vuelta a las columnas
      2 
      3 df.loc[::-1]   #Dar la vuelta a las filas

NameError: name 'df' is not defined
```

In [ ]: