# Librería Pandas

Antonio Espín Herranz

# Pandas

Estructuras de datos pandas

Series y DataFrame

# Pandas

- Pandas proporciona estructuras de datos ricas y funciones diseñadas para **trabajar con datos estructurados** rápidos, fáciles y expresivos.

- El objeto principal en pandas es el **DataFrame**, un marco bidimensional, estructura de datos tabular y orientada a columnas con etiquetas de fila y columna (**tipo hoja Excel**).

```
>>> frame
    total_bill  tip   sex     smoker  day  time    size
1   16.99       1.01  Female  No      Sun  Dinner  2
2   10.34       1.66  Male    No      Sun  Dinner  3
3   21.01       3.5   Male    No      Sun  Dinner  3
4   23.68       3.31  Male    No      Sun  Dinner  2
5   24.59       3.61  Female  No      Sun  Dinner  4
6   25.29       4.71  Male    No      Sun  Dinner  4
7   8.77        2     Male    No      Sun  Dinner  2
8   26.88       3.12  Male    No      Sun  Dinner  4
9   15.04       1.96  Male    No      Sun  Dinner  2
10  14.78       3.23  Male    No      Sun  Dinner  2
```

# Pandas

- Se relaciona con otras biblioteca dedicadas a la ciencia y el análisis de datos como:
  - Librerías de computación numérica como: **NumPy** y **SciPy**
  - Librerías de análisis de datos como **statsmodels** y **Scikit-learn**
  - Librerías de visualización de datos como **matplotlib**

# Pandas

- Combina las características de computación de alto rendimiento con NumPy con características flexibles de manipulación de datos de **hojas de cálculo** y BD relacionales.

- **Lectura y escritura de los principales formatos de archivos de datos como**:
  - CSV, texto, Excel, json, bases de datos SQL …

- Filtrar, agregar o eliminar columnas. Reindexar datos.

- Realizar cálculos de agregados o transformaciones de datos con el motor de **group by**.

# Pandas

- Está construida sobre la librería NumPy
- Desde 2008.
- Sobre todo enfocada a **trabajar con conjuntos grandes de registros** y operar con estos datos.
- Alrededor de 800 colaboradores.
- Mantiene operaciones tipo **Merge** como en las BD.
- Documentación de pandas:
  - https://pandas.pydata.org/pandas-docs/stable/api.html

# Estructuras de datos
# Series y DataFrame

- Importar la librería:
- import pandas as pd
  - De esta forma para utilizar la librería hay que preceder a cada función con el prefijo pd.

- from pandas import Series, DataFrame
  - Nos evitamos poner el prefijo.

# Series

- Una Serie en pandas es un array de una dimensión que contiene cualquier tipo de datos NumPy) y una matriz asociada de datos (los índices).

- La serie se puede crear a partir de una lista, los índices serían numéricos:
  - obj = Series([4,7,-5,3])
  - Se puede acceder a todo, o por separado: los índices y los valores.
    - print("valores:", **obj.values**)
    - print("Indices:", **obj.index**) *# Devuelve un iterador* ***RangeIndex***
    - print("Indices:", **list(obj.index))**

# Series

- Los índices pueden ser de otro tipo, como ocurre en un **dict**.
- obj2 = **Series**([4,7,-5,3], **index**=['d','b','a','c’])

- Las series también permiten indexación: para recuperar o modificar un valor:
  - print(obj2['d'])
  - obj2['d’]=6

  - Se pueden utilizar varios índices para seleccionar un conjunto:
  - obj2[[‘c’,’a’,’d’]]

# Series

- Se pueden **filtrar los datos** o aplicar operaciones a todos los datos de la serie:
  - print("mayores que 0\n",obj2[obj2 > 0])
  - print(obj2*2)

- El **operador in** también funciona:
  - print('b' in obj2)

- Sobre un objeto **Series** se pueden aplicar cálculos de **numpy**:
  - import numpy as np
  - np.exp(obj2)

# Series

- Una **serie también se puede crear a partir de un dict**:
  - sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
  - obj3 = Series(sdata)

- Se pueden **pasar las claves**:
  - estados = ['California','Ohio','Oregon','Texas']
  - obj4 = Series(sdata, index=estados)

  - En este caso como California **no existe** se le asociará el valor **NaN** (not a number).
    - Y Utah no estará, porque pasamos 4 claves…

# Series

- Se puede preguntar por valores nulos o no nulos
  - **Funciones** de pandas
    - print(pd.isnull(obj4))
    - print(pd.notnull(obj4))
    - Con los datos anteriores California será null.
    - Se indica con True / False.

- También vienen como **métodos** sobre un objeto Series:
  - print(obj4.isnull())
  - print(obj4.notnull())

# Series

- # sumando series:
  - print(obj3+obj4)
  - Se alinea automáticamente con las claves

- # establecer **nombres** a la **serie** y **índice**:
  - obj4.name = "poblacion"
  - obj4.index.name = "estado"
  - print(obj4)

```
In [37]: obj3
Out[37]:
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64

In [38]: obj4
Out[38]:
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64

In [39]: obj3 + obj4
Out[39]:
California         NaN
Ohio           70000.0
Oregon         32000.0
Texas         142000.0
Utah               NaN
dtype: float64
```

# DataFrame

- Un DataFrame representa una **estructura de datos tabular**, tipo **hoja de cálculo** que contiene una colección de columnas, cada una de las cuales puede ser un tipo de valor diferente (numérico, cadena, booleano, etc.)
  - Se pueden representar más dimensiones utilizando la indexación jerárquica.

- El **DataFrame** tiene un **índice de fila y columna**; puede ser pensado como **un dict de Series** (uno para todos compartiendo el mismo índice)

# DataFrame

- Se puede construir con un diccionario y listas de la misma longitud.

```
data = {
    'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
    'year': [2000, 2001, 2002, 2001, 2002],
    'pop': [1.5, 1.7, 3.6, 2.4, 2.9]
}
frame = DataFrame(data)
```

| | pop | state | year |
|---|---|---|---|
| 0 | 1.5 | Ohio | 2000 |
| 1 | 1.7 | Ohio | 2001 |
| 2 | 3.6 | Ohio | 2002 |
| 3 | 2.4 | Nevada | 2001 |
| 4 | 2.9 | Nevada | 2002 |

Si utilizamos el libro de notas
De **Jupyter** mostrará los DataFrame
Como tablas de HTML

| | state | year | pop |
|---|---|---|---|
| 0 | Ohio | 2000 | 1.5 |
| 1 | Ohio | 2001 | 1.7 |
| 2 | Ohio | 2002 | 3.6 |
| 3 | Nevada | 2001 | 2.4 |
| 4 | Nevada | 2002 | 2.9 |
| 5 | Nevada | 2003 | 3.2 |

Para Dataframe grandes:
Utilizar **frame.head()** muestra las 5 primeras filas

# DataFrame

- Se pueden pasar las columnas:
  - frame = DataFrame(data, columns=['year', 'state', 'pop’])
    - En este caso recoloca las columnas.

- frame2 = DataFrame(**data**,
  **columns**=['year', 'state', 'pop', 'debt’],
  **index**=['one', 'two', 'three', 'four', 'five’])
  - Se puede añadir una columna que no tenga datos, se crea con valor **NaN**
  - **Se pueden indicar los índices con nombre a las filas**.

# DataFrame

- Se pueden **obtener las columnas**:

  frame2.**columns**

  - Devuelve una lista con las columnas.
  - También podemos modificar los nombres de las columnas con: **frame2.columns = lista_cols**
    - frame2.columns = [‘col1’, ‘col2’, … ‘colN’]

- Se puede pedir **una columna en concreto (Serie)**:

  frame2[‘state’] 🡨🡪 frame2.state

  - Devuelve los datos de esta columna, con sus índices en fila.
  - Podemos fijar también la fila para obtener la **celda**:
    - **frame2[‘state’][0]**

# DataFrame

- Dimensiones:
  - Para saber las dimensiones del DataFrame tenemos la propiedad:
  - **df.shape**: Devuelve una tupla (filas, cols).
    - No cuenta la fila de las cabeceras.

- Para obtener listas de las filas de un DataFrame:
  - A partir de un df ya cargado:

```
# Será una lista de listas.
L = df.values.tolist()
```

# DataFrame: Filas

- Dispone de la propiedad **loc** que se puede indexar por clave o por posición (al acceder a una **fila**)

  frame2.**loc**[0] 🡐 🡒 frame2.**loc**[‘one’]

- A nivel de fila se pueden realizar las siguientes operaciones:
  - A partir de un DataFrame (df) cargado:
    - **Slice**: df = df[3:5] # muestra las filas 3 y 4
    - También es válido: **df.loc[2:4]**
    - Una **fila**: print(df.loc[3]) # la fila 3

# DataFrame: Columnas

- Se puede modificar una columna asignando un valor:

  frame2[‘debt’] = 16.2

- También se pueden crear campos calculados en función de otras columnas del DataFrame:
  - frame2[‘nuevaCol’] = frame2[‘col1’] + frame2[‘col3’]

- Con valores distintos utilizando arange de numpy:

  frame2[debt] = np.arange(5.)

# DataFrame

- **Se pueden utilizar Series con valor y con los índices para asignarlos a una columna**.

- Si nos saltamos alguna clave se rellenará con **NaN**.

- Ejemplo:
  - En este caso no rellena las claves: one y three.
    - val = Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
    - frame2['debt']=val
    - Se puede añadir una columna al DataFrame a partir de un objeto **Series**

# DataFrame: nuevas columnas

- Si se asigna un v**alor a una columna, se crea automáticamente** y se puede rellenar:
  - frame2['eastern']=frame2.state == 'Ohio’
  - En este caso itera por todas las filas, realizando la comparación.
  - OJO, no se pueden crear nuevas columnas utilizando la sintaxis: **frame2.nuevaCol =** …

| | year | state | pop | debt | eastern |
|---|---|---|---|---|---|
| one | 2000 | Ohio | 1.5 | NaN | True |
| two | 2001 | Ohio | 1.7 | -1.2 | True |
| three | 2002 | Ohio | 3.6 | NaN | True |
| four | 2001 | Nevada | 2.4 | -1.5 | False |
| five | 2002 | Nevada | 2.9 | -1.7 | False |

# DataFrame

- Se pueden borrar columnas de la misma forma que elementos en un dict:

  del frame2[‘eastern’]

- Se pueden **anidar dict**:

```
data2 = {
        'Nevada':{2001: 2.4, 2002: 2.9},
        'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}
}
```

- Se puede **Trasponer** la tabla:

  print(frame2.T)

| | Nevada | Ohio |
|---|---|---|
| 2000 | NaN | 1.5 |
| 2001 | 2.4 | 1.7 |
| 2002 | 2.9 | 3.6 |

| | 2000 | 2001 | 2002 |
|---|---|---|---|
| Nevada | NaN | 2.4 | 2.9 |
| Ohio | 1.5 | 1.7 | 3.6 |

# Constructor de DataFrame

- Al constructor de DataFrame se le pueden pasar los siguientes datos:
  - 2D ndarray:
    - Una matriz de datos, pasando etiquetas opcionales de fila y columna
  - Dict de arrays, list, tuplas:
    - Cada secuencia se convierte en una columna en el DataFrame; todas las secuencias deben tener la misma longitud
  - Arrays estructurados de Numpy:
    - Tratado como el caso de "dict de arrays“
  - Dict de Series:
    - Cada valor se convierte en una columna; índices de cada serie se unen para formar el índice de fila del resultado si no se pasa ningún índice explícito

# Constructor de DataFrame

- Dict de dicts:
  - Cada diccionario interno se convierte en una columna; las claves se unen para formar el índice de filas como en el caso "dict de Series“.
- Lista de dict o Series:
  - Cada elemento se convierte en una fila en el DataFrame; unión de claves dict o índices de serie se convierten en las etiquetas de columna del DataFrame.
- Lista de listas o tuplas:
  - Tratado como 2D array
- Otro DataFrame
- Arrays de máscaras de Numpy:
  - Igual que en el caso “nd array”

# Objectos Index de Pandas

- Los objetos **Index** de Pandas se encargan de mantener las etiquetas del eje y otros metadatos (como los nombres de los ejes)
- Los índices son objetos inmutables y no se pueden modificar.
- Los índices se pueden recuperar de un objeto Series.
- Se puede aplicar slicing
- Error si hacemos: índices[1] = ‘d’
- Un índice puede tener etiquetas duplicadas:
  - Labels = pd.Index([‘uno’,’dos’,’uno’,’tres’])

```
In [20]: obj = pd.Series(range(3), index=['a', 'b', 'c'])

In [21]: obj

Out[21]: a    0
         b    1
         c    2
         dtype: int64

In [22]: type(obj)

Out[22]: pandas.core.series.Series

In [23]: indices = obj.index

In [24]: indices

Out[24]: Index(['a', 'b', 'c'], dtype='object')
```

# Métodos de Index

| Method | Description |
|---|---|
| append | Concatenate with additional Index objects, producing a new Index |
| diff | Compute set difference as an Index |
| intersection | Compute set intersection |
| union | Compute set union |
| isin | Compute boolean array indicating whether each value is contained in the passed collection |
| delete | Compute new Index with element at index i deleted |
| drop | Compute new index by deleting passed values |
| insert | Compute new Index by inserting element at index i |
| is_monotonic | Returns True if each element is greater than or equal to the previous element |
| is_unique | Returns True if the Index has no duplicate values |
| unique | Compute the array of unique values in the Index |

# Ejemplos: Métodos Index

```
In [26]: type(indices)
Out[26]: pandas.core.indexes.base.Index
```

---

```
In [29]: nuevo = pd.core.indexes.base.Index(['d'])

In [30]: nuevo
Out[30]: Index(['d'], dtype='object')

In [31]: indices.append(nuevo)
Out[31]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

# Pandas

**Funcionalidad Esencial**

# Reindexar

- Método: **reindex**
- Crea un nuevo objeto Index, se crea una reasignación de índices.
- Se aplica a un objeto **Series**.

```
In [32]: indices.reindex(['x','y','z','w'])

Out[32]: (Index(['x', 'y', 'z', 'w'], dtype='object'),
          array([-1, -1, -1, -1], dtype=int64))

In [33]: indices

Out[33]: Index(['a', 'b', 'c'], dtype='object')
```

# Reindexar

- In [93]: obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c’])
- In [94]: obj
- Out[94]:
  - d  4.5
  - b  7.2
  - a  -5.3
  - c  3.6
  - dtype: float64

- Al reindexar los valores que faltan se rellenan con NaN
- In [95]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e’])
- In [96]: obj2
- Out[96]:
  - a  -5.3
  - b  7.2
  - c  3.6
  - d  4.5
  - **e  NaN**
  - dtype: float64

# Reindexar

- Para los datos ordenados como series temporales, puede ser conveniente hacer alguna interpolación o relleno de valores al volver a indexar.
- La opción nos permite hacer esto, utilizando un método como , que rellena hacia adelante los valores:método: **ffill**

- obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
- obj3.reindex(range(6), method='**ffill**')
- Rellena los huecos:
  - 0 blue
  - 1 blue
  - 2 purple
  - 3 purple
  - 4 yellow
  - 5 yellow

También se puede hacer un reindex a un frame
frame2 = frame.reindex(['a','b','c','d'])

# Ejemplo con DataFrame

```
In [100]: frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
    .....:                     index=['a', 'c', 'd'],
    .....:                     columns=['Ohio', 'Texas', 'California'])

In [101]: frame
Out[101]:
   Ohio  Texas  California
a     0      1           2
c     3      4           5
d     6      7           8

In [102]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])

In [103]: frame2
Out[103]:
   Ohio  Texas  California
a   0.0    1.0         2.0
b   NaN    NaN         NaN
c   3.0    4.0         5.0
d   6.0    7.0         8.0
```

En el DataFrame:
El índice que no tiene valores
Se rellena automáticamente con NaN

Los índices se pueden eliminar
Con **reset_index()** aplicado al DataFrame
Pasan otra vez a ser columnas (si teníamos indexada alguna Columna)

# Argumentos de la función reindex

| Argumento | Descripción |
|---|---|
| `index` | Nueva secuencia para usar como índice. Puede ser una instancia de index o cualquier otra estructura de datos de Python similar a una secuencia. Un índice se utilizará exactamente como es sin ninguna copia. |
| `method` | Método de interpolación (relleno); se llena hacia adelante, mientras se llena hacia atrás.`'ffill''bfill'` |
| `fill_value` | Sustituya el valor que se utilizará al introducir los datos que faltan volviendo a indizar. |
| `limit` | Cuando se rellena hacia delante o hacia atrás, el espacio de tamaño máximo (en número de elementos) que se va a rellenar. |
| `tolerance` | Cuando se rellena hacia delante o hacia atrás, el espacio de tamaño máximo (en distancia numérica absoluta) se debe rellenar para las coincidencias inexactas. |
| `level` | Coincidir con el índice simple en el nivel de MultiIndex; de lo contrario, seleccione subconjunto de. |
| `copy` | Si , siempre copie los datos subyacentes incluso si el nuevo índice es equivalente al índice anterior; si , no copie los datos cuando los índices sean equivalentes.`TrueFalse` |

# Eliminar índices: drop

- Al eliminar un índice de una Serie se obtiene un nuevo objeto Index
  - obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e’])
  - new_obj = obj.drop(‘c’)

- También permite borrar varios:
  - obj.drop(['d', 'c’])

- La función drop admite como parámetro un valor o una lista de valores.

# DataFrame: filas

- **Añadir una fila:**
  - Suponiendo que tenemos 3 columnas: id, nombre y cargo: se indica el nuevo índice de fila.

    df.loc[len(df)] = [len(df)+1,'Jorge','administrador’]

  - Mas eficiente es utilizar el método **append**, permite añadir una fila a un DF o directamente un DF entero a otro.
    - df1 = df1.append(df2, **ignore_index=True**) → **Crea nuevos índices**.

- **Borrar una fila:**
  - Borrar 4 filas por el index:
    - df = df.drop([0,1,2,3])

  - También se pueden indicar las **etiquetas** de los **índices** si las hemos asignado en la propiedad index al crear el DataFrame.

# DataFrame: columnas

- **Borrar columnas:**

  df = df.**drop**(columns=[‘col1’,’col2’,…])

- **Reordenar columnas:**
  - Si queremos **cambiar el orden** en el que se encuentran las **columnas** en el DataFrame:
  - Ojo, serán columnas existentes (indicamos los nombres de las columnas y **el orden se marca con la posición que indicamos en la lista**:

    df = df[[‘col1’,’col2’,…]]

- **Renombrar una columna:**

  df = df.**rename**(columns = {‘nombre_old’:’nombre_new'})

# Ejemplo: Borrar index en DataFrame

```
In [112]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
   .....:                     index=['Ohio', 'Colorado', 'Utah', 'New York'],
   .....:                     columns=['one', 'two', 'three', 'four'])

In [113]: data
Out[113]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

```
In [114]: data.drop(['Colorado', 'Ohio'])
Out[114]:
          one  two  three  four
Utah        8    9     10    11
New York   12   13     14    15
```

# Ejemplo: Borrar index en DataFrame

- Para eliminar valores de las columnas se puede pasar axis=1 o axis='columns':

```
In [115]: data.drop('two', axis=1)
Out[115]:
          one  three  four
Ohio        0      2     3
Colorado    4      6     7
Utah        8     10    11
New York   12     14    15

In [116]: data.drop(['two', 'four'], axis='columns')
Out[116]:
          one  three
Ohio        0      2
Colorado    4      6
Utah        8     10
New York   12     14
```

# Borrar sin devolver un nuevo objeto

- El método drop puede modificar el tamaño o la forma de una Serie o DataFrame. Con el parámetro inplace=True

- In [117]: obj.drop('c', inplace=True)

# Ejemplo

```
>>> from pandas import DataFrame
>>> import numpy as np
>>> data = np.arange(20).reshape(5,4)
>>> data
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
>>> df = DataFrame(data, columns=['col0','col1','col2','col3'])
>>> df
   col0  col1  col2  col3
0     0     1     2     3
1     4     5     6     7
2     8     9    10    11
3    12    13    14    15
4    16    17    18    19
>>> df.drop('col1',axis='columns',inplace=True)
>>> df
   col0  col2  col3
0     0     2     3
1     4     6     7
2     8    10    11
3    12    14    15
4    16    18    19
```

# Series indexación, filtrado

- Funciona como en los arrays de numpy, pero en este caso podemos:

- Indexar por la etiqueta o por la posición del índice.
  - obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
  - obj['b']
  - obj[1]
  - obj[2:4]
  - obj[['b','a','d']]

  - Establecer filtros de forma similar a numpy:
    - obj[obj < 2]

  - Se puede realizar slicing: obj['b':'c']
  - También para modificar: obj['b':'c'] = 5
  - ***OJO en pandas slicing es inclusivo. En el caso anterior la 'c' entra.***

# Ejemplo: indexación en un DataFrame

```
In [130]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
    .....:                    index=['Ohio', 'Colorado', 'Utah', 'New York'],
    .....:                    columns=['one', 'two', 'three', 'four'])

In [131]: data
Out[131]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15

In [132]: data['two']
Out[132]:
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int64

In [133]: data[['three', 'one']]
Out[133]:
          three  one
Ohio          2    0
Colorado      6    4
Utah         10    8
New York     14   12
```

```
In [134]: data[:2]
Out[134]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7

In [135]: data[data['three'] > 5]
Out[135]:
          one  two  three  four
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

# Indexación DataFrame con resultados bool

```
In [136]: data < 5
Out[136]:
             one    two  three   four
Ohio        True   True   True   True
Colorado    True  False  False  False
Utah       False  False  False  False
New York   False  False  False  False

In [137]: data[data < 5] = 0

In [138]: data
Out[138]:
          one  two  three  four
Ohio        0    0      0     0
Colorado    0    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

# Resumen para indexar en un DataFrame

- df[val]
  - Selecciona una sola columnas o secuencia de columnas.
  - df[‘col1’]
  - df[[‘col1’,’col2’]]

- df.loc[val]
  - Selecciona una sola fila o subconjunto de filas por etiqueta.
  - df.loc['a']
  - df.loc[['a','b']]

- df.loc[:, val]
  - Selecciona una sola columna o subconjunto por etiquetas.
  - df.loc[:,'col1’]
  - df.loc['a':'c','col1']

- df.loc[val1,val2]
  - Seleccionar filas y columnas por etiqueta
  - df.loc['a','col1']

| | col1 | col2 | col3 | col4 | col5 |
|---|---|---|---|---|---|
| a | 0 | 1 | 2 | 3 | 4 |
| b | 5 | 6 | 7 | 8 | 9 |
| c | 10 | 11 | 12 | 13 | 14 |
| d | 15 | 16 | 17 | 18 | 19 |

# Resumen para indexar en un DataFrame

- df.iloc[where]
  - Selecciona una sola fila o subconjunto de filas por posición de un entero
  - df.iloc[2]

- df.iloc[where_i, where_j]
  - Selecciona filas y columnas por posición de entero
  - df.iloc[2,3]

- df.at[label_i, label_j]
  - Seleccione un único valor escalar por fila y etiqueta de columna
  - df.at[‘a’,’col1’]

- df.iat[i, j]
  - Seleccione un único valor escalar por posición de fila y columnas (enteros).
  - df.iat[2,3]

| | col1 | col2 | col3 | col4 | col5 |
|---|---|---|---|---|---|
| a | 0 | 1 | 2 | 3 | 4 |
| b | 5 | 6 | 7 | 8 | 9 |
| c | 10 | 11 | 12 | 13 | 14 |
| d | 15 | 16 | 17 | 18 | 19 |

# DataFrame: Filtrar filas

- Se pueden **filtrar** filas, añadiendo **condiciones**:
  - Imprimir las filas donde la columna id se encuentre entre 4 y 8 (sin incluir)
    - print(df[(df.id > 4) **&** (df.id < 8)])
    - **Utilizar & para el and**
    - **Utilizar | para el or**

- Las 10 **primeras** filas:
  - print(df.**head**(10))

- Las **últimas** 10 filas:
  - print(df.**tail**(10))

# DataFrame: Filtrar y modificar

- Para modificar utilizar loc si no, genera una copia y no nos dejará modificar:
  - dtHu.loc[(dtHu.NUDOS >= n_min) & (dtHu.NUDOS <= n_max),['CATEGORIA','COLOR']]=cat,color

  - cat y color: Son dos variables que vienen de una extracción anterior (por ejemplo otro dataframe).

# DataFrame: obtener información

- Una vez tenemos el DataFrame cargado en memoria podemos utilizar los métodos **head** y **tail** como indicamos anteriormente.

- También disponemos del método **info** que nos indica el tipo de cada columna y cuantos datos no nulos tenemos en cada columna.

- El número de filas y columnas lo podemos obtener:
  - **FILAS**: len(mi_data_frame)
  - **COLS**: len(mi_data_frame.columns)

- El método **describe** también ofrece información pero sería más temas estadísticos. Para el conjunto de campos / columnas de tipo numérico calcula: suma, cuenta, media, mínimo, máximo y percentiles cada 25%.

- También podemos seleccionar varias columnas según el tipo de dato que tienen:
  - datos.**select_dtypes**(include='number) → seleccionar columnas con números
    - Otros tipos pueden ser object (para texto), categorical (datos categóricos)

# Aritmética y alineación de datos

- Se puede operar con dos DataFrame

```
In [157]: df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
   .....:                     index=['Ohio', 'Texas', 'Colorado'])

In [158]: df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
   .....:                     index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [159]: df1
Out[159]:
             b    c    d
Ohio       0.0  1.0  2.0
Texas      3.0  4.0  5.0
Colorado   6.0  7.0  8.0

In [160]: df2
Out[160]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0
```

```
In [161]: df1 + df2
Out[161]:
             b   c     d   e
Colorado   NaN NaN   NaN NaN
Ohio       3.0 NaN   6.0 NaN
Oregon     NaN NaN   NaN NaN
Texas      9.0 NaN  12.0 NaN
Utah       NaN NaN   NaN NaN
```

# Aritmética y operaciones

- Cuando se suman dos DataFrame con columnas o índices diferentes. En los valores no coincidentes se rellenan con NaN, se pueden utilizar valores de relleno para esas celdas (pero en los DataFrame de origen):
  - Se ponen a cero las celdas que no existen para poder sumar con el otra celda.

- El resultado es diferente de df1 + df2
- El método es: **df1.add(df2, fill_value=0)**

# Resumen Métodos Aritméticos

- add, radd — Adición +
- sub, rsub — Restar -
- div, rdiv — División /
- floordiv, rfloordiv — División entera //
- mul, rmul — Multiplicación *
- pow, rpow — exponenciación **

- Con los métodos que empieza con r (rsum, rsub, etc.) opera con los DF al revés, por ejemplo:
  - df1.sub(df2) → df1 - df2
  - df1.rsub(df2) → df2- df1

# Operaciones entre Series y DataFrame

- Al igual que con las matrices NumPy de diferentes dimensiones, también se define la **aritmética entre DataFrame y Series**.

- Podemos tener un DataFrame de 4x3 y una Serie de 3 elementos.

- Cuando se calcula la resta: frame - series, la resta se aplica a nivel de fila. Esto recibe el nombre de **Broadcasting**.

- Cuando hacemos una operación y no coinciden las etiquetas de los índices entre el frame y la serie se rellena con un valor **NaN**.

- Por defecto, se aplica a nivel de filas, pero puede aplicarse a nivel de columnas con el parámetro: axis='index' en el método:
  - **frame.sub(series3, axis='index')**

# Ejemplo

```
In [188]: series3 = frame['d']

In [189]: frame
Out[189]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0

In [190]: series3
Out[190]:
Utah       1.0
Ohio       4.0
Texas      7.0
Oregon    10.0
Name: d, dtype: float64

In [191]: frame.sub(series3, axis='index')
Out[191]:
          b    d    e
Utah   -1.0  0.0  1.0
Ohio   -1.0  0.0  1.0
Texas  -1.0  0.0  1.0
Oregon -1.0  0.0  1.0
```

**La resta la realiza a nivel de columnas!**

# Aplicación y asignación de Funciones

- Las **funciones** de la librería numpy que se aplican a los arrays de numpy, se pueden **aplicar a los objetos DataFrame**.

- A partir de un DataFrame ya creado podemos calcular el valor absoluto de cada una de las celdas del frame.

- **np.abs(frame)**
  - Como resultado obtenemos otro frame donde se ha aplicado a todas las celdas la función abs.

```
In [192]: frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),
   .....:                      index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [193]: frame
Out[193]:
                b         d         e
Utah    -0.204708  0.478943 -0.519439
Ohio    -0.555730  1.965781  1.393406
Texas    0.092908  0.281746  0.769023
Oregon   1.246435  1.007189 -1.296221

In [194]: np.abs(frame)
Out[194]:
                b         d         e
Utah     0.204708  0.478943  0.519439
Ohio     0.555730  1.965781  1.393406
Texas    0.092908  0.281746  0.769023
Oregon   1.246435  1.007189  1.296221
```

# Aplicación y asignación de Funciones

- También se pueden aplicar funciones o funciones lambda con el método **apply**()
  - f = lambda x : x.max() - x.min()

- frame.**apply**(f)
  - En este caso se aplica a cada columna.
  - Calcula el mínimo y el máximo de cada columna y aplica la función

```
In [195]: f = lambda x: x.max() - x.min()

In [196]: frame.apply(f)
Out[196]:
b    1.802165
d    1.684034
e    2.689627
dtype: float64
```

# Aplicación y asignación de Funciones

- A la función apply se le puede pasar el parámetro axis='columns' y el cálculo se realiza para cada fila:

```
In [197]: frame.apply(f, axis='columns')
Out[197]:
Utah      0.998382
Ohio      2.521511
Texas     0.676115
Oregon    2.542656
dtype: float64
```

# Aplicando funciones

- Las funciones pueden devolver una Serie:

```
In [198]: def f(x):
    .....:     return pd.Series([x.min(), x.max()], index=['min', 'max'])

In [199]: frame.apply(f)
Out[199]:
             b         d         e
min -0.555730  0.281746 -1.296221
max  1.246435  1.965781  1.393406
```

# Ejemplo aplicando una lambda

- Formatear a dos decimales:

```
In [200]: format = lambda x: '%.2f' % x

In [201]: frame.applymap(format)
Out[201]:
             b     d      e
Utah     -0.20  0.48  -0.52
Ohio     -0.56  1.97   1.39
Texas     0.09  0.28   0.77
Oregon    1.25  1.01  -1.30
```

# Aplicar funciones a Series

- El objeto Series de pandas también proporciona el método **map** para poder aplicar una función a los elementos de la Serie.

```
In [202]: frame['e'].map(format)
Out[202]:
Utah       -0.52
Ohio        1.39
Texas       0.77
Oregon     -1.30
Name: e, dtype: object
```

# Ordenación

- Podemos ordenar las Series con sort_index
  - obj = pd.Series(range(4), index=['d', 'a', 'b', 'c’])
  - obj.**sort_index**()

- En el caso de dataframe se puede ordenar por filas o por columnas:
- frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
  index=['three', 'one'], columns=['d', 'a', 'b', 'c’])
  - Ordenar por índices: frame.sort_index() → one, three
  - Ordenador por columnas: frame.sort_index(axis=1) → a,b,c,d

- La ordenación es ascending de forma predeterminada:
  - frame.sort_index(axis=1, ascending=False)

# Ordenación

- Para ordenar por los valores: **sort_values**
  - obj = pd.Series([4, 7, -3, 2])
  - obj.sort_values()

- Los valores NaN se ordenan al final de forma predeterminada:
  - obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])
  - obj.sort_values()

# Ordenación

- En la ordenación de DataFrame se puede indicar la columna(s) por la(s) que ordenamos:
  - frame = pd.DataFrame({**'b**': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})
  - frame.sort_values(by=**'b**’)

- Por varias columnas:
  - frame.sort_values(by=['a', 'b’])

- Devuelve otro frame ordenado.

# Método rank en Series / DataFrame

- El método **rank** se aplica a objetos Series y DataFrame.

- La clasificación asigna rangos desde uno hasta el número de puntos de datos válidos en una matriz.

- Si lo valores de la Serie están ordenados el rango es ascendente.

```
>>> from pandas import Series
>>> obj = Series(np.arange(10))
>>> obj
0    0
1    1
2    2
3    3
4    4
5    5
6    6
7    7
8    8
9    9
dtype: int32
>>> obj.rank()
0     1.0
1     2.0
2     3.0
3     4.0
4     5.0
5     6.0
6     7.0
7     8.0
8     9.0
9    10.0
dtype: float64
```

# Otro ejemplo

```
In [217]: obj = pd.Series([7, -5, 7, 4, 2, 0, 4])

In [218]: obj.rank()
Out[218]:
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

Los métodos del objeto rank
ascending=True (por defecto)
method='…' Ver tabla siguiente

El método es la forma de desempate

# Ejemplo en DataFrame

- Se pueden calcular rangos en filas o columnas:

```
In [221]: frame = pd.DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],
   .....:                       'c': [-2, 5, 8, -2.5]})

In [222]: frame
Out[222]:
     b  a    c
0  4.3  0 -2.0
1  7.0  1  5.0
2 -3.0  0  8.0
3  2.0  1 -2.5

In [223]: frame.rank(axis='columns')
Out[223]:
     b    a    c
0  3.0  2.0  1.0
1  3.0  1.0  2.0
2  1.0  2.0  3.0
3  3.0  2.0  1.0
```

# Métodos de aplicación a rank

| Método | Descripción |
|---|---|
| `'average'` | Defecto: asignar el rango promedio a cada entrada en el grupo igual |
| `'min'` | Utilizar el rango mínimo para todo el grupo |
| `'max'` | Utilizar el rango máximo para todo el grupo |
| `'first'` | Asignar clasifica en el orden en que aparecen los valores en los datos |
| `'dense'` | Al igual que `method='min'`,pero los rangos siempre aumentan en 1 entre los grupos en lugar del número de elementos iguales en un grupo |

# Indices con etiquetas duplicadas

- Tanto en las Series como en los DataFrame podemos tener etiquetas en los índices duplicadas.
  - obj = pd.Series(range(5), index=['a', 'a', 'b', 'b', 'c’])
- La **propiedad** del índice nos puede indicar si es único o no.
  - obj.index.**is_unique**
- La selección se comporta de forma diferente, al devolver todas las filas que están indexadas por dicho índice.
  - obj['a’]
  - a 0
  - a 1d
  - type: int64

# Indices con etiquetas duplicadas (DataFrame)

```
In [229]: df = pd.DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])

In [230]: df
Out[230]:
          0         1         2
a  0.274992  0.228913  1.352917
a  0.886429 -2.001637 -0.371843
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228

In [231]: df.loc['b']
Out[231]:
          0         1         2
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228
```

Se comporta de la misma forma
Que las Series

# Pandas

Resumen de datos y estadística

# Resumen de datos y estadística

- Los objetos pandas están equipados con un conjunto de métodos matemáticos y estadísticos comunes.

- La mayoría de ellos pertenecen a la categoría de reducciones o estadísticas de resumen, métodos que extraen un único valor (como la suma o la media) de una serie o una serie de valores de las filas o columnas de un DataFrame.

# Operaciones

```
     one  two
a   1.40  NaN
b   7.10 -4.5
c    NaN  NaN
d   0.75 -1.3
```

- Podemos sumar a nivel de columnas o de filas:
- A partir de un DataFrame ya creado:
  - **df.sum()** suma a nivel de columnas o **df.sum(axis=0)**

```
one     9.25
two    -5.80
dtype: float64
```

  - **df.sum(axis='columns') o df.sum(axis=1)**

```
a     1.40
b     2.60
c     0.00
d    -0.55
dtype: float64
```

Por defecto los valores NaN
Se excluyen, se pueden tener
En cuenta con el parámetro: skipna=False

# Opciones de métodos de reducción

| Método | Descripción |
|---|---|
| `axis` | Eje a reducir más; 0 para filas de DataFrame y 1 para columnas |
| `skipna` | Excluir valores faltantes; `True`por defecto |
| `level` | Reducir agrupados por nivel si el eje está indexado jerárquicamente (MultiIndex) |

Algunos métodos, como `idxmin`y `idxmax`, devuelven estadísticas indirectas como el valor del índice donde se alcanzan los valores mínimo o máximo:

# Operaciones

- De la misma forma que **sum**() disponemos de **mean**() para la media.

- Los métodos: **idxmin**() y **idxmax**(), devuelven el índice donde se encuentra el mínimo y el máximo valor.

- La suma acumulada se puede calcular con **cumsum**()
- Para obtener **diversas estadísticas** utilizar: df.**describe**()
  - Devuelve count, mean, std, percentiles, etc.

# Salida del método describe

DataFrame

```
In [239]: df.describe()
Out[239]:
              one       two
count    3.000000  2.000000
mean     3.083333 -2.900000
std      3.493685  2.262742
min      0.750000 -4.500000
25%      1.075000 -3.700000
50%      1.400000 -2.900000
75%      4.250000 -2.100000
max      7.100000 -1.300000
```

El método **describe** también
Se puede aplicar a un objeto **Series**

```
In [240]: obj = pd.Series(['a', 'a', 'b', 'c'] * 4)

In [241]: obj.describe()
Out[241]:
count     16
unique     3
top        a
freq       8
dtype: object
```

# Resumen de operaciones

- count
  - Número de valores no NaN
- describe
  - Calcular conjunto de estadísticas resumidas para Series o cada columna DataFrame
- min, max
  - Calcular valores mínimos y máximos
- argmin, argmax
  - Calcular ubicaciones de índice (números enteros) en las que se obtiene el valor mínimo o máximo, respectivamente
- idxmin, idxmax
  - Calcular etiquetas de índice en las que se obtiene el valor mínimo o máximo, respectivamente.
- quantile
  - Calcular cuantil de muestra que va de 0 a 1

# Resumen de operaciones II

- sum
  - Suma de valores
- mean
  - La Media de valores
- median
  - Aritmética mediana (50% cuantil) de valores
- mode
  - La moda
- mad
  - desviación absoluta del valor medio
- prod
  - Producto de todos los valores
- var
  - Muestra varianza de valores

# Resumen de operaciones III

- std
  - Muestra desviación estándar de valores
- skew
  - Muestra asimetría (tercer momento) de los valores
- kurt
  - Muestra curtosis de valores.
- cumsum
  - Acumulativo suma de valores
- cummin, cummax
  - Acumulativo mínimo o máximo de valores, respectivamente
- cumprod
  - Acumulativo producto de valores
- diff
  - Calcular primera diferencia aritmética (útil para series de tiempo)
- pct_change
  - Calcular por ciento cambios

# DataFrame: Operaciones en una columna

- Las operaciones antes detalladas se pueden aplicar a nivel de **columna**:
- Cada columna se representa por una **Serie**
  - print('Suma:',df['importe'].**sum**())
  - print('Media:',df['importe'].**mean**())
  - print('Max:',df['importe'].**max**())
  - print('Min:',df['importe'].**min**())
  - print('cuenta:',df['importe'].**count**())
  - print('Describe:',df['importe'].describe())
    - Muestra el tipo de la columna.

# Correlación y Covarianza

- Instalar el paquete: pandas-datareader
  - **pip install pandas-datareader** en Python
  - **conda install pandas-datareader** en Anaconda

- Algunas estadísticas resumidas, como la correlación y la covarianza, se calculan a partir de pares de argumentos.
- Se pueden utilizar como datos precios de acciones y volúmenes obtenidos de Yahoo.
  - Se puede obtener con el módulo pandas-datareader.

# Correlación y Covarianza

- La **correlación** se puede aplicar a una columna de un DataFrame. El objeto Series (una columna del DataFrame) dispone de método corr().

- La covarianza también se puede aplicar a un objeto Series con el método cov().

- Ambas funciones se aplican sobre dos columnas del DataFrame.
  - df[‘col1’].**corr**(df[‘col2’]) ←→ df.col1.corr(df.col2)
  - df[‘col1’].**cov**(df[‘col2’]) ← → df.col1.cov(df.col2)

# Correlación y Covarianza

- Ambos métodos si se aplican a nivel de DataFrame hacen el cruce de todas las columnas tomadas de dos en dos.
- df.corr()
- df.cov()

```
In [249]: returns.corr()
Out[249]:
          AAPL      GOOG       IBM      MSFT
AAPL  1.000000  0.407919  0.386817  0.389695
GOOG  0.407919  1.000000  0.405099  0.465919
IBM   0.386817  0.405099  1.000000  0.499764
MSFT  0.389695  0.465919  0.499764  1.000000

In [250]: returns.cov()
Out[250]:
          AAPL      GOOG       IBM      MSFT
AAPL  0.000277  0.000107  0.000078  0.000095
GOOG  0.000107  0.000251  0.000078  0.000108
IBM   0.000078  0.000078  0.000146  0.000089
MSFT  0.000095  0.000108  0.000089  0.000215
```

# Correlación y Covarianza

- Utilizando DataFrames también se pueden calcular la correlación de una columna con las demás con el método **corrwith**.
- La sintaxis se puede aplicar dos formas:
  - df.corrwith(df.col)
  - df.corrwith(col)

Se puede utilizar el parámetro: **axis=columns** para realizar los cálculos Por filas.

```
In [251]: returns.corrwith(returns.IBM)
Out[251]:
AAPL    0.386817
GOOG    0.405099
IBM     1.000000
MSFT    0.499764
dtype: float64
```

```
In [252]: returns.corrwith(volume)
Out[252]:
AAPL   -0.075565
GOOG   -0.007067
IBM    -0.204849
MSFT   -0.092950
dtype: float64
```

# Gestión duplicados

- Dentro de un DataFrame podemos tener datos repetidos:
  - Podemos preguntar cuales son los repetidos.
  - Y además se pueden borrar.

- Los métodos **dt.duplicated**() muestran con True y False las filas repetidas.

- Con **dt.drop_duplicates**() elimina las filas duplicadas.

- **Valores únicos** de una Serie:
  - obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c’])
  - uniques = obj.**unique**()
  - Se pueden ordenar posteriormente con **uniques.sort()**

# Gestión de duplicados

- Recuento de valores duplicados: **value_count()**
  - obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c’])

  - obj.value_counts()

    In [256]: obj.value_counts()
    Out[256]:
    **a 3**
    **c 3**
    **b 2**
    **d 1**
    dtype: int64

- El método value_counts también se encuentra a nivel de pandas como una función:
  - **pd.value_counts(obj.values, sort=False)**

# Gestión de duplicados (ejemplo con DF) I

- Se puede calcular un histograma en varias columnas relacionadas con un DataFrame:
- Para ello podemos aplicar la función **value_counts** con el método **apply** del objeto DataFrame. A partir de estos datos:

```
In [265]: data = pd.DataFrame({'Qu1': [1, 3, 4, 3, 4],
   .....:                      'Qu2': [2, 3, 1, 2, 3],
   .....:                      'Qu3': [1, 5, 2, 4, 4]})

In [266]: data
Out[266]:
   Qu1  Qu2  Qu3
0    1    2    1
1    3    3    5
2    4    1    2
3    3    2    4
4    4    3    4
```

# Gestión de duplicados (ejemplo con DF) II

- Coloca como índice los valores distintos del DF, y hace un recuento de cada valor por columna.

```
In [267]: result = data.apply(pd.value_counts).fillna(0)

In [268]: result
Out[268]:
   Qu1  Qu2  Qu3
1  1.0  1.0  1.0
2  0.0  2.0  1.0
3  2.0  2.0  0.0
4  2.0  0.0  2.0
5  0.0  0.0  1.0
```

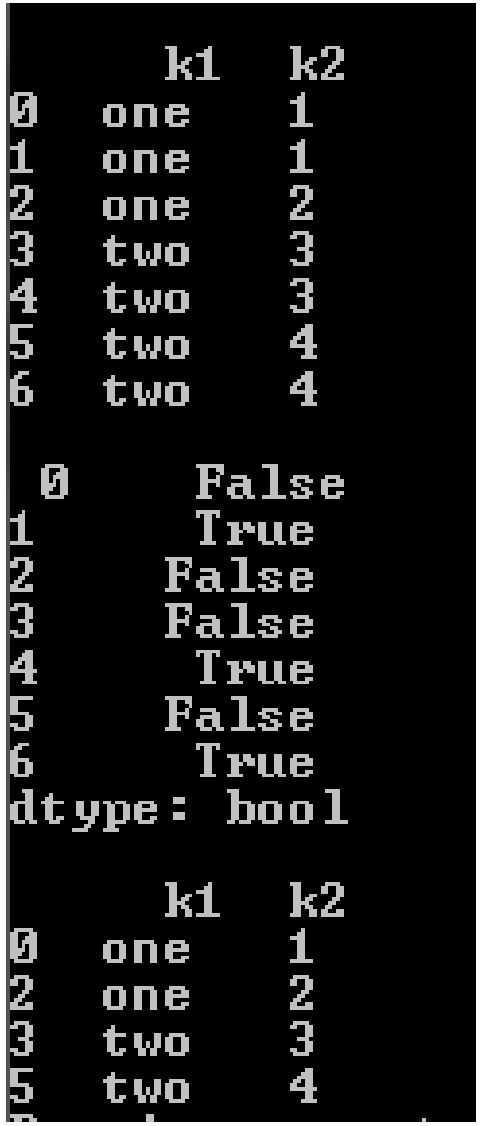
# Ejemplo

```
import pandas as pd
from pandas import Series, DataFrame

datos = DataFrame({'k1':['one']*3 + ['two'] * 4,
                   'k2':[1,1,2,3,3,4,4]})
print('\n',datos)

#mostrar los duplicados:
print('\n',datos.duplicated())

datos2 = datos.drop_duplicates()
print('\n',datos2)
```



# Pertenencia a un conjunto

- **isin** realiza un verificación de pertenencia a un conjunto y puede ser útil para filtrar un conjunto de datos a un subconjunto de valores en una serie o columna en un DataFrame.

```
In [258]: obj
Out[258]:
0    c
1    a
2    d
3    a
4    a
5    b
6    b
7    c
8    c
dtype: object

In [259]: mask = obj.isin(['b', 'c'])
```

```
In [260]: mask
Out[260]:
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool
```

# Pertenencia a un conjunto

- Relacionado con el método **isin** el método **Index.get_indexer** que devuelve una matriz con las posiciones que ocupan unos valores de array en otro.

- Ejemplo:

```
In [262]: to_match = pd.Series(['c', 'a', 'b', 'b', 'c', 'a'])

In [263]: unique_vals = pd.Series(['c', 'b', 'a'])

In [264]: pd.Index(unique_vals).get_indexer(to_match)
Out[264]: array([0, 2, 1, 1, 0, 2])
```

# Resumen de métodos

| Método | Descripción |
|---|---|
| `isin` | Calcule una matriz booleana que indique si cada valor de Serie está contenido en la secuencia de valores pasada |
| `get_indexer` | Calcularíndices enteros para cada valor en una matriz en otra matriz de valores distintos; útil para la alineación de datos y las operaciones de tipo unión |
| `unique` | Calcular matriz de valores únicos en una serie, devueltos en el orden observado |
| `value_counts` | Devuelve una serie que contiene valores únicos como índice y frecuencias como valores, ordenados recuento en orden descendente |

# pandas

Datos categóricos

# Datos categóricos

- Dentro de un DataFrame o una Serie nos podemos encontrar columnas que no son numéricas, normalmente columnas de texto con valores que tienden a repetirse.

- Estas columnas se pueden convertir en datos categóricos para mejorar el rendimiento y espacio ocupado por estas.

- Además los algoritmos de aprendizaje automático no funcionan correctamente con columnas de texto y necesitamos convertir previamente estas columnas en datos categóricos.

# Datos categóricos

- Una columna puede contener instancias repetidas de un conjunto de mas pequeño de valores distintos.

- Las funciones **unique** y **value_counts** permiten extraer los distintos valores y recuentos para calcular sus frecuencias.

- Muchos sistemas de datos (para almacenamiento de datos, computación estadística u otros usos) han desarrollado enfoques especializados para representar datos con valores repetidos para un almacenamiento y computación más eficientes.
  - En el almacenamiento de datos, una mejor práctica es utilizar las llamadas **tablas de dimensiones** que contiene los valores distintos y almacena las observaciones primarias como claves enteras que hacen referencia a la tabla de dimensiones.

# Datos categóricos

- Podemos representar los valores de una Serie por medio de valores numéricos.

- Asignando un número a cada valor distinto.

- Podemos utilizar el método **take** para recuperar la serie original:
  - dim.**take**(values)

```
In [17]: values = pd.Series([0, 1, 0, 0] * 2)

In [18]: dim = pd.Series(['apple', 'orange'])

In [19]: values
Out[19]:
0    0
1    1
2    0
3    0
4    0
5    1
6    0
7    0
dtype: int64

In [20]: dim
Out[20]:
0     apple
1    orange
dtype: object
```

# Datos categóricos

- dim.**take**(values)
  - Esta representación como números enteros se denomina  categórica o representación codificada por diccionario.
  - La matriz de valores distintos puede denominarse categorías , diccionario o niveles de los datos.
  - Los valores enteros que hacen referencia a las categorías se denominan códigos de categoría o simplemente códigos .

```
In [21]: dim.take(values)
Out[21]:
0     apple
1    orange
0     apple
0     apple
0     apple
1    orange
0     apple
0     apple
dtype: object
```

# Datos categóricos

- La representación categórica puede producir mejoras de rendimiento significativas cuando realiza análisis. También puede realizar transformaciones en las categorías sin modificar los códigos.

- Algunos ejemplos de transformaciones que se pueden realizar a un costo relativamente bajo son:
  - Cambio de nombre de categorías.
  - Agregar una nueva categoría sin cambiar el orden o la posición de las categorías existentes.

# Datos categóricos

- Pandas tiene un tipo especial llamo **categorical** que utiliza una representación basada en números.

- Un DataFrame con una columna de tipo texto se puede convertir a un tipo categórico.

| | basket_id | fruit | count | weight |
|---|---|---|---|---|
| 0 | 0 | apple | 5 | 3.858058 |
| 1 | 1 | orange | 8 | 2.612708 |
| 2 | 2 | apple | 4 | 2.995627 |
| 3 | 3 | apple | 7 | 2.614279 |
| 4 | 4 | apple | 12 | 2.990859 |
| 5 | 5 | orange | 8 | 3.845227 |
| 6 | 6 | apple | 5 | 0.033553 |
| 7 | 7 | apple | 4 | 0.425778 |

# Datos categóricos

- La columna fruit se puede convertir a dato categórico llamando:

```
In [26]: fruit_cat = df['fruit'].astype('category')

In [27]: fruit_cat
Out[27]:
0     apple
1    orange
2     apple
3     apple
4     apple
5    orange
6     apple
7     apple
Name: fruit, dtype: category
Categories (2, object): [apple, orange]
```

Fruit_cat no es una matriz de **numpy** sino una instancia de **pandas.categorical**

# Datos categóricos

- Un objeto categorical tiene como atributos **categories** y **codes**.

```
In [28]: c = fruit_cat.values

In [29]: type(c)
Out[29]: pandas.core.arrays.categorical.Categorical

In [30]: c.categories
Out[30]: Index(['apple', 'orange'], dtype='object')

In [31]: c.codes
Out[31]: array([0, 1, 0, 0, 0, 1, 0, 0], dtype=int8)
```

El cambio permanente dentro de un DataFrame se puede realizar asignando el valor a la columna:
**df['fruit'] = df['fruit'].astype('category')**

Pandas proporciona la función pd.**Categorical**(['hola','hola','adios','adios','adios'])

# Datos categóricos

Los datos categóricos **NO tienen** porque ser texto
Puede ser cualquier tipo **inmutable**.

- Disponemos de un constructor alternativo:
  - pd.Categorical.from_codes( codes, categories, [ordered=True])
  - Es posible indicar un orden en las categorías, aunque se encuentren desordenadas.
  - La instancia devuelve por la función se puede ordenar mediante:
    - my_cats_2.**as_ordered**()

```
In [36]: categories = ['foo', 'bar', 'baz']

In [37]: codes = [0, 1, 2, 0, 0, 1]

In [38]: my_cats_2 = pd.Categorical.from_codes(codes, categories)

In [39]: my_cats_2
Out[39]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo, bar, baz]
```

# Cálculos con categóricos

- El uso de Categorical en pandas en comparación con la versión no codificada (como una versión de cadenas) generalmente te comporta de la misma manera.

- La función **groupby**: funciona mejor cuando se trabaja con categóricos.

```
In [43]: np.random.seed(12345)

In [44]: draws = np.random.randn(1000)

In [45]: draws[:5]
Out[45]: array([-0.2047,  0.4789, -0.5194, -0.5557,  1.9658])
```

# Cálculos con categóricos

- La función **qcut** permite el agrupamiento de cuartiles.

```
In [46]: bins = pd.qcut(draws, 4)

In [47]: bins
Out[47]:
[(-0.684, -0.0101], (-0.0101, 0.63], (-0.684, -0.0101], (-0.684, -0.0101], (0.63,
 3.928], ..., (-0.0101, 0.63], (-0.684, -0.0101], (-2.949999999999997, -0.684],
(-0.0101, 0.63], (0.63, 3.928]]
Length: 1000
Categories (4, interval[float64]): [(-2.949999999999997, -0.684] < (-0.684, -0.0
101] < (-0.0101, 0.63] <
                                    (0.63, 3.928]]
```

# Cálculos con categóricos

- Se pueden indicar etiquetas con el parámetro **labels**.

```
In [48]: bins = pd.qcut(draws, 4, labels=['Q1', 'Q2', 'Q3', 'Q4'])

In [49]: bins
Out[49]:
[Q2, Q3, Q2, Q2, Q4, ..., Q3, Q2, Q1, Q3, Q4]
Length: 1000
Categories (4, object): [Q1 < Q2 < Q3 < Q4]

In [50]: bins.codes[:10]
Out[50]: array([1, 2, 1, 1, 3, 3, 2, 2, 3, 3], dtype=int8)
```

# Cálculos con categóricos

- El bins categórico etiquetado no contiene información sobre los bordes del contenedor en los datos, por lo que podemos usar groupby para extraer algunas estadísticas resumidas:

```
In [51]: bins = pd.Series(bins, name='quartile')

In [52]: results = (pd.Series(draws)
   ....:            .groupby(bins)
   ....:            .agg(['count', 'min', 'max'])
   ....:            .reset_index())

In [53]: results
Out[53]:
  quartile  count       min       max
0       Q1    250 -2.949343 -0.685484
1       Q2    250 -0.683066 -0.010115
2       Q3    250 -0.010032  0.628894
3       Q4    250  0.634238  3.927528
```

La columna **quartile** conserva la información categórica original, incluido el orden

```
In [54]: results['quartile']
Out[54]:
0    Q1
1    Q2
2    Q3
3    Q4
Name: quartile, dtype: category
Categories (4, object): [Q1 < Q2 < Q3 < Q4]
```

# Mejor rendimiento con categóricos

- Si realizamos muchos análisis en un conjunto de datos en particular, la conversión a categórica puede generar ganancias sustanciales en el rendimiento general.

- Una **versión categórica** de una columna DataFrame a menudo también **utilizará significativamente menos memoria**.
  - El método **memory_usage**() del objeto Series devuelve la ocupación en memoria de un objeto.

- Las **operaciones** de **GroupBy** pueden ser significativamente **más rápidas** con categóricos porque los algoritmos subyacentes utilizan la matriz de códigos basados en números enteros en lugar de una matriz de cadenas.

# Ejemplo: Comprobar espacio

```
In [55]: N = 10000000

In [56]: draws = pd.Series(np.random.randn(N))

In [57]: labels = pd.Series(['foo', 'bar', 'baz', 'qux'] * (N // 4))
```

```
In [58]: categories = labels.astype('category')

In [59]: labels.memory_usage()
Out[59]: 80000080

In [60]: categories.memory_usage()
Out[60]: 10000272
```

# Métodos categóricos

- El objeto Series que contiene datos categóricos tiene varios métodos especiales similares a **Series.str** (*métodos de cadena especializados*).

- Dada una Serie el atributo especial **cat** proporciona acceso a métodos categóricos, proporciona propiedades como **codes** y **categories**

```
In [62]: s = pd.Series(['a', 'b', 'c', 'd'] * 2)

In [63]: cat_s = s.astype('category')

In [64]: cat_s
Out[64]:
0    a
1    b
2    c
3    d
4    a
5    b
6    c
7    d
dtype: category
Categories (4, object): [a, b, c, d]
```

```
In [65]: cat_s.cat.codes
Out[65]:
0    0
1    1
2    2
3    3
4    0
5    1
6    2
7    3
dtype: int8

In [66]: cat_s.cat.categories
Out[66]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

# Métodos categóricos

- Podemos cambiar las categorías con el método: **set_categories**.
  - actual_categories = ['a', 'b', 'c', 'd', 'e']
  - cat_s2 = cat_s.cat.**set_categories**(actual_categories)

- Cambios en las categorías se plasman en la función **values_counts** aunque no tenga valores la nueva categoría (si estará presente).
  - Mostrará el recuento de cada categoría, con la nueva categoría a cero.

# Métodos categóricos

```
In [70]: cat_s.value_counts()
Out[70]:
d    2
c    2
b    2
a    2
dtype: int64

In [71]: cat_s2.value_counts()
Out[71]:
d    2
c    2
b    2
a    2
e    0
dtype: int64
```

- En grandes conjuntos de datos, los categóricos se utilizan a menudo como una herramienta conveniente para ahorrar memoria y mejorar el rendimiento.

- Después de filtrar un DataFrame o una serie grandes, es posible que muchas de las categorías no aparezcan en los datos.
  - Para ayudar con esto, podemos usar el **remove_unused_categories** método para recortar las categorías no observada

# Ejemplo

```
In [72]: cat_s3 = cat_s[cat_s.isin(['a', 'b'])]

In [73]: cat_s3
Out[73]:
0    a
1    b
4    a
5    b
dtype: category
Categories (4, object): [a, b, c, d]

In [74]: cat_s3.cat.remove_unused_categories()
Out[74]:
0    a
1    b
4    a
5    b
dtype: category
Categories (2, object): [a, b]
```

# Resumen de métodos categóricos

| Método | Descripción |
|---|---|
| `add_categories` | Adjuntar categorías nuevas (no utilizadas) al final de las categorías existentes |
| `as_ordered` | Hacer categorías ordenadas |
| `as_unordered` | Hacer categorías desordenadas |
| `remove_categories` | Eliminar categorías, estableciendo cualquier valor eliminado en nulo |
| `remove_unused_categories` | Eliminar cualquier valor de categoría que no aparezca en los datos |
| `rename_categories` | Reemplazar categorías con el conjunto indicado de nuevos nombres de categorías; no se puede cambiar el número de categorías |
| `reorder_categories` | Se comporta como `rename_categories`, pero también puede cambiar el resultado para tener categorías ordenadas |
| `set_categories` | Reemplazar las categorías con el conjunto indicado de nuevas categorías; puede agregar o quitarcategorias |

# Crear variables ficticias para modelar

- Cuando utilizamos estadísticas o herramientas de aprendizaje automático, a menudo transformaremos datos categóricos en variables ficticias, conocidas como la codificación **one-hot**.

- Esto implica la creación de un DataFrame con una columna para cada categoría distinta; estas columnas contienen **1 para las ocurrencias de una categoría determinada** y **0 en caso contrario**.

- Para ello disponemos de la función **get_dummies** de pandas (realiza la codificación one-hot.

# Ejemplo

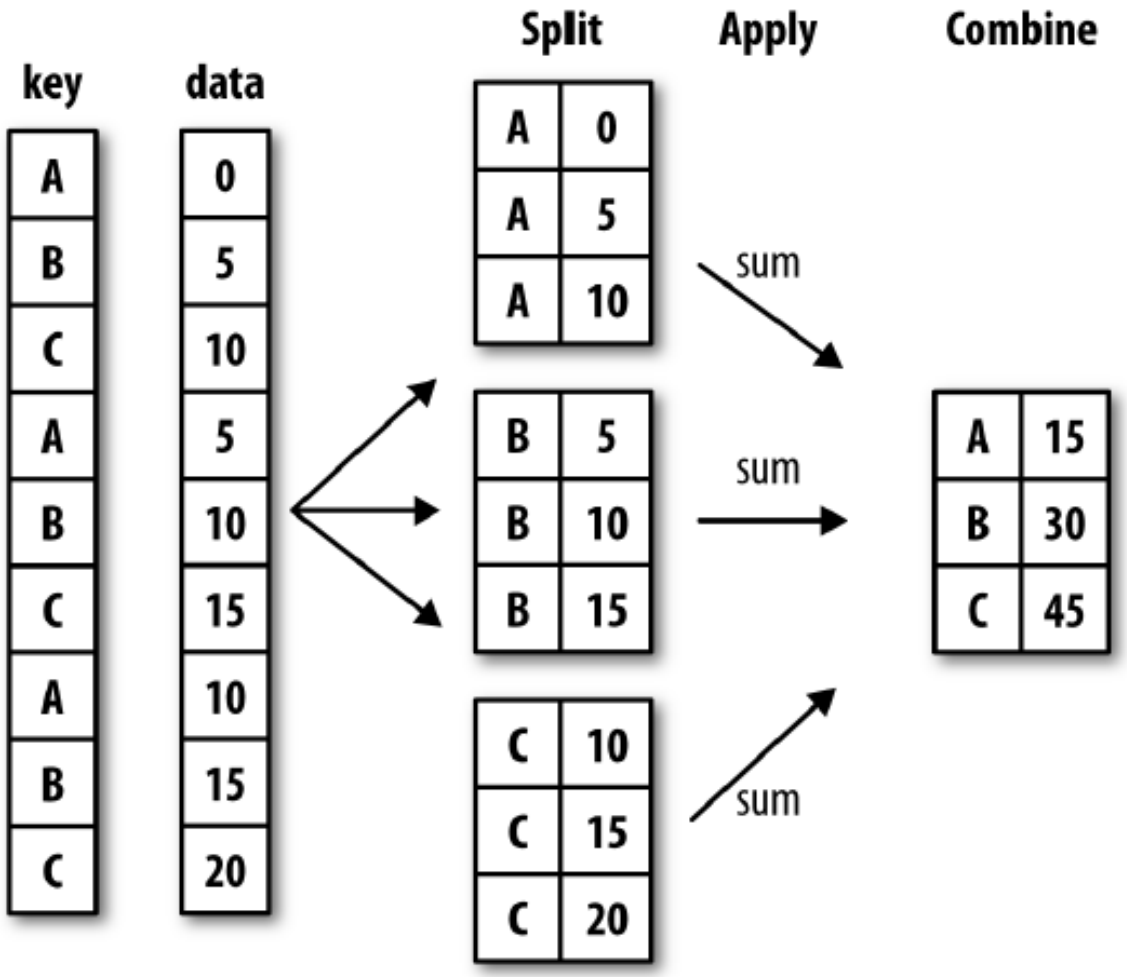
```
In [75]: cat_s = pd.Series(['a', 'b', 'c', 'd'] * 2, dtype='category')

In [76]: pd.get_dummies(cat_s)
Out[76]:
   a  b  c  d
0  1  0  0  0
1  0  1  0  0
2  0  0  1  0
3  0  0  0  1
4  1  0  0  0
5  0  1  0  0
6  0  0  1  0
7  0  0  0  1
```

# pandas

GroupBy / Merge

# DataFrame: Group by



# DataFrame: Group by

- Podemos agrupar por una columna y utilizar otra como clave:

  g1 = df['importe'].groupby(df['pais’])

  *Agrupa el campo importe por país.*

- Una vez se ha creado el grupo podemos aplicar operaciones:

  print('Media: ',g1.mean())

  print('Suma: ', g1.sum())

  print('Min: ', g1.min())

  print('Max: ', g1.max())

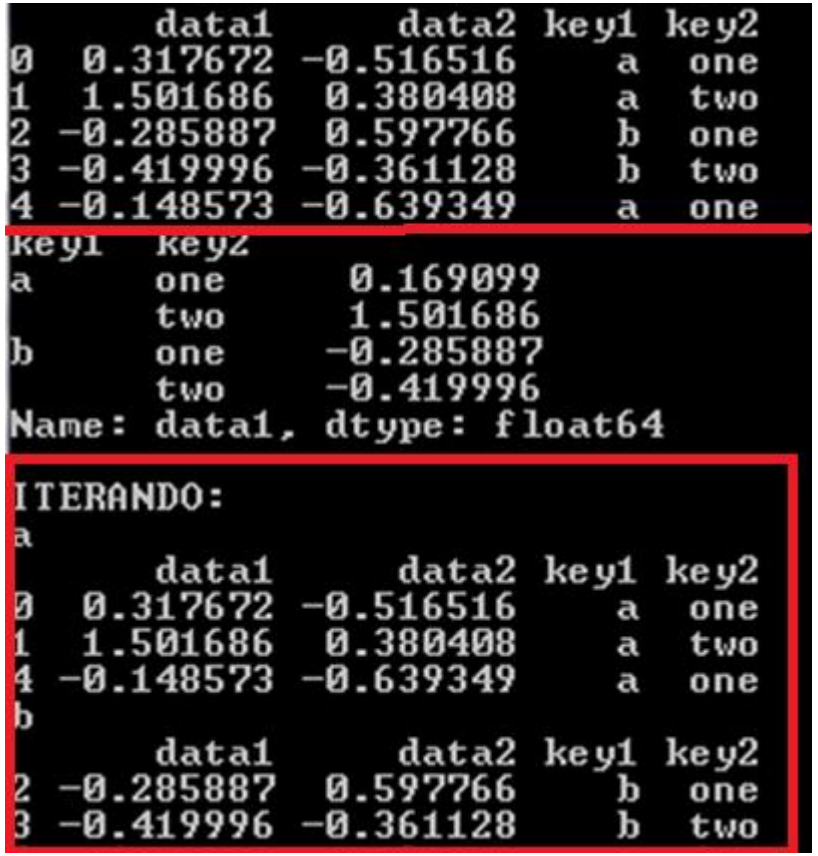
- Se puede agrupar por varias claves:

  means = df['data1'].groupby([df['key1'], df['key2']]).mean()

# DataFrame: Group by

- Iterar por los resultados:

```
for nombre, grupo in df.groupby('key1'):
    print(nombre)
    print(grupo)
```



# Agrupar por Año / Mes

- Podemos convertir las fechas a datetime con la función:
  - pd.to_datetime(col_de_fecha)

- Ejemplo:
  - frame['FECHA'] = pd.to_datetime(frame['FECHA’])

- Para agrupar por año:
  - gNumerosAño = dfNumeros['NUMERO'].groupby(dfNumeros.FECHA.dt.year)
  - gNumerosAño.mean() … sum() , … count(), etc.

- Para agrupar por mes:
  - gNumerosMes = dfNumeros['NUMERO'].groupby(dfNumeros.FECHA.dt.month)

- Para agrupar por mes dentro de cada año:
  - gNumerosAño = dfNumeros['NUMERO'].groupby(dfNumeros.FECHA.dt.year, dfNumeros.FECHA.dt.month )

# Merge

- **pandas.merge** conecta filas de DataFrames basados en una o mas claves.

- Esta operación es similar a los join que se realizan con Bases de datos en MySQL.

- Une los dataframe mediante la coincidencia de las claves. Y devuelve un nuevo DataFrame

  pandas.merge(df1, df2)

# Ejemplo

```
import pandas as pd
from pandas import Series, DataFrame
df1 = DataFrame({'key':list('hola'), 'data1': range(4)})
df2 = DataFrame({'key':list('adios'), 'data2': range(5)})
print('\nMerge:')
dfM = pd.merge(df1, df2)
print(dfM)
```

```
df1
   data1 key
0      0   h
1      1   o
2      2   l
3      3   a

df2
   data2 key
0      0   a
1      1   d
2      2   i
3      3   o
4      4   s

Merge:
   data1 key  data2
0      1   o      3
1      3   a      0
```

# Merge

- Si no indicamos el nombre de la clave, pandas superpone los nombres:
  - Pero se pueden indicar los nombres de las claves de cada DataFrame.

- A partir de 2 dataframe creados:

  pd.merge(df3, df4, left_on='**lkey**', right_on='**rkey**')

# Pandas

Importación, exportación y carga de datos

# Carga, Almacenaje de datos
# Formato de los ficheros

- Dentro de la librería **pandas** tenemos utilidades para cargar ficheros en **DataFrame** de una forma sencilla.

- Las funciones más habituales son:
  - **read_csv**: Carga datos delimitados de un fichero, o URL. Por defecto, utiliza la coma como separador.
  - **read_table**: Idem del anterior pero por defecto utiliza **\t** como separador.

  - ***Ambas funciones hacen lo mismo, la diferencia es el parámetro por defecto***.
    - ***"El separador de Columnas"***

# Cargar datos

- import pandas as pd
- df = pd.read_csv('datos.txt’)
- Por defecto utiliza la ,
- Toma la primera fila como los nombres de las columnas.
- Si indicamos el parámetro **header=None**, genera nombres para las cabeceras de forma automática.
  - Las numera desde 0 … n-1 como hace con las filas.

# Cargar datos

- Los nombres de las columnas se puede indicar y utilizar uno de estos nombres como clave de las filas.
- Por ejemplo:
  - names = ['a', 'b', 'c', 'd', 'message']
  - df2 = pd.read_csv('datos.txt', names=names, index_col='message')

| | a | b | c | d |
|---|---|---|---|---|
| message | | | | |
| message | a | b | c | d |
| hello | 1 | 2 | 3 | 4 |
| world | 5 | 6 | 7 | 8 |
| foo | 9 | 10 | 11 | 12 |

# Cargar datos

- Índices jerárquicos, si partimos de estos datos:
  - key1,key2,value1,value2
  - one,a,1,2
  - one,b,3,4
  - one,c,5,6
  - one,d,7,8
  - two,a,9,10
  - two,b,11,12
  - two,c,13,14
  - two,d,15,16

```
df3 = pd.read_csv('datos2.txt', index_col=['key1','key2'])
```

| | | value1 | value2 |
|---|---|---|---|
| key1 | key2 | | |
| one | a | 1 | 2 |
| | b | 3 | 4 |
| | c | 5 | 6 |
| | d | 7 | 8 |
| two | a | 9 | 10 |
| | b | 11 | 12 |
| | c | 13 | 14 |
| | d | 15 | 16 |

# Cargar datos

- Los valores nulos:
  - Valores que no existen dentro de un fichero, puede ser cadena vacía o un valor especial. Pandas suele utilizar NA, #IND y NULL

```
something,a,b,c,d,message
one,1,2,3,4,NA
two,5,6,,8,world
three,9,10,11,12,foo
```

| | something | a | b | c | d | message |
|---|---|---|---|---|---|---|
| 0 | one | 1 | 2 | 3.0 | 4 | NaN |
| 1 | two | 5 | 6 | NaN | 8 | world |
| 2 | three | 9 | 10 | 11.0 | 12 | foo |

```
result = pd.read_csv('datos3.txt')
print(pd.isnull(result))
```

| | something | a | b | c | d | message |
|---|---|---|---|---|---|---|
| 0 | False | False | False | False | False | True |
| 1 | False | False | False | True | False | False |
| 2 | False | False | False | False | False | False |

# Procesar ficheros grandes

- Se puede limitar el número de filas a leer.
  - Parámetro **nrows=número**
    - pd.read_csv('ch06/ex6.csv', nrows=5)

- Procesar el fichero por partes:
  - **chunker** = pd.read_csv('ex6.csv', chunksize=1000)
  - En trozos de 1000 líneas, devuelve un iterador.

- Ejemplo, acumular una columna y contar el número de claves encontradas.

```
tot = Series([])
for piece in chunker:
    tot = tot.add(piece['key'].value_counts() ,fill_value=0)
```

# Parámetros: read_csv / read_table

- **path**
  - Ruta al fichero o URL a cargar.
- **sep o delimiter**
  - Separador, puede ser un char. o una exp.reg
- **header**
  - Número de fila para las columnas, por defecto es 0. Puede ser None para indicar que no hay columnas.
- **index_col**
  - Índice para las columnas, nombre o número de las lista de nombres de columnas.
  - Puede ser una lista para formar índices jerárquicos.
- **names**
  - La lista de nombres de las columnas.
- **skiprows**
  - Número de filas a ignorar al principio del fichero.
  - Una lista con los números.
  - La primera es la 0.
  - pd.read_csv('ch06/ex4.csv', skiprows=[0, 2, 3])

# Parámetros: read_csv / read_table

- **nrows**
  - Número de filas a leer del fichero (desde el principio).
- **skip_footer**
  - Número de filas a saltar por el final del archivo.
- **encoding**
  - Codificación del fichero. Por ejemplo: ‘utf-8’
- **thousands**
  - Separador de miles, coma o punto. ‘,’ o ‘.’
- **chunksize**
  - Se indica un número que indica un tamaño de las partes en que se dividirá un fichero. Devuelve un iterador para pasar por cada pieza.
- **iterator**
  - Devuelve un objeto TextParser para leer el archivo por partes.

# Parámetros: read_csv / read_table

- **na_values**
  - Secuencia de valores para reemplazar por NA.

- **parse_dates**
  - Intente analizar datos en datetime; False por defecto. Si True, intentará analizar todas las columnas.
    - De lo contrario, puede especificar una lista de números de columna o un nombre para analizar. Si el elemento de la lista es tupla o lista, combinará varias columnas juntas y analizará hasta la fecha (por ejemplo, si la fecha / hora se divide en dos columnas).

- **keep_date_col**
  - Si une columnas para analizar la fecha, mantenga las columnas unidas; False por defecto.

- **dayfirst**
  - Al analizar fechas potencialmente ambiguas, trátelas como formato internacional (por ejemplo, 6/7/2012 -> 7 de junio de 2012); False por defecto.

- **date_parser**
  - Función a utilizar para analizar fechas.

# Lectura de otros formatos

- **read_clipboard**
  - Versión de read_csv que lee datos del portapapeles.
- **read_excel**
  - Leer archivos: Excel XLS, XLSX
- **read_html**
  - Leer tablas que se encuentran en un documento HTML.
- **read_json**
  - Lee una cadena de Json
- **read_sql**
  - Leer los resultados de una consulta SQL (usando SQLAlchemy)

# Lectura de otros formatos

- **read_fwf**
  - Leer datos en formato de columna de ancho fijo (es decir, sin delimitadores).
- **read_hdf**
  - Leer HDF5 archivos escritos por pandas.
- **read_msgpack**
  - Leer pandas datos codificados con el formato binario MessagePack.
- **read_pickle**
  - Leer un objeto arbitrario almacenado en formato Python pickle.
- **read_sas**
  - Leer un Conjunto de datos SAS almacenado en uno de los formatos de almacenamiento personalizados del sistema SAS.

# Lectura de otros formatos

- **read_stata**
  - Leer un conjunto de datos de Formato de archivo Stata.

- **read_feather**
  - Leer el Formato de archivo binario Feather.

# Escribir ficheros de texto

- **DataFrame** dispone del método **to_csv** para volcar a un formato de texto.
- Para mostrarlo por consola: sys.stdout o la ruta a un fichero para grabarlo.
- Se le puede indicar un carácter de separación, **sep = '..'**
- Se pueden reemplazar los valores NaN por otro valor, **na_rep = 'NULL'**

```
import sys
import pandas as pd
from pandas import Series, DataFrame

data = pd.read_csv('ex5.csv')
print(data.to_csv(sys.stdout, sep='|'))
data.to_csv('nuevo_ex5.csv', sep='|')
data.to_csv(sys.stdout, na_rep='NULL')
```

# Escribir ficheros de texto

- Se pueden indicar que columnas queremos grabar y si van o con índices. Parámetros: index = True / False y columns = [‘col1’, …]
- data.to_csv(sys.stdout, **index=False**, **columns=['a','b','c’])**

- **Se puede grabar en un fichero csv:**

  unDataFrame.**to_csv**(‘nombreFichero.csv’, sep=‘;’)

# Trabajar con formatos delimitados

- Para cualquier archivo con **un delimitador de un solo carácter**, podemos usar el módulo **csv** de Python incorporado.

- Para usarlo, pasamos cualquier archivo abierto u objeto similar de un archivo a **csv.reader**:

```
import csv
f = open('examples/ex7.csv')
reader = csv.reader(f)

for line in reader:
    print(line)

['a', 'b', 'c']
['1', '2', '3']
['1', '2', '3']
```

Los archivos CSV pueden venir con un **formato diferente**,
Se puede definir una subclase de la clase: **csv.Dialect** indicando El formato del fichero.

```
class my_dialect(csv.Dialect):
    lineterminator = '\n'
    delimiter = ';'
    quotechar = '"'
    quoting = csv.QUOTE_MINIMAL

reader = csv.reader(f, dialect=my_dialect)
```

# Opciones del dialecto CSV

| Argumento | Descripción |
|---|---|
| `delimiter` | Cadena de un carácter para separar campos; predeterminado a `','`. |
| `lineterminator` | Terminador de línea para escritura; predeterminado a `'\r\n'`. Reader ignora esto y reconoce terminadores de línea multiplataforma. |
| `quotechar` | Carácter de comillas para campos con caracteres especiales (como un delimitador); predeterminado es `'"'`. |
| `quoting` | Convención de citas. Las opciones incluyen `csv.QUOTE_ALL`(citar todos los campos), `csv.QUOTE_MINIMAL`(solo campos con caracteres especiales como el delimitador) `csv.QUOTE_NONNUMERIC`y `csv.QUOTE_NONE`(sin comillas). Consulte la documentación de Python para obtener detalles completos. Por defecto es `QUOTE_MINIMAL`. |
| `skipinitialspace` | Ignore los espacios en blanco después de cada delimitador; predeterminado es `False`. |
| `doublequote` | Cómo manejar el carácter entre comillas dentro de un campo; si `True`se duplica (consulte la documentación en línea para conocer todos los detalles y el comportamiento). |
| `escapechar` | Cadena para escapar del delimitador si `quoting`se establece en `csv.QUOTE_NONE`; desactivado por defecto. |

# Escritura CSV manual

- Para escribir archivos delimitados manualmente, podemos usar **csv.writer**.
- Acepta un objeto de archivo abierto que se puede escribir y las mismas opciones de formato y dialecto como csv.reader:

```
with open('mydata.csv', 'w') as f:
    writer = csv.writer(f, dialect=my_dialect)
    writer.writerow(('one', 'two', 'three'))
    writer.writerow(('1', '2', '3'))
    writer.writerow(('4', '5', '6'))
    writer.writerow(('7', '8', '9'))
```

# Procesar ficheros Excel: Leer

- Con la librería pandas se puede leer ficheros Excel (xlsx / xls).

```
import pandas as pd
excelFile = pd.ExcelFile(fichero)
tabla = excelFile.parse("nombre_hoja")
```

- pandas proporciona la función **pd.read_excel**:
  - frame = pd.**read_Excel**('fichero.xlsx', 'Hoja1')

- Si obtenemos el error:
  - ImportError: Install xlrd >= 0.9.0 for Excel support
  - Instalar el modulo xlrd
    - **pip install xlrd** (en una consola de administrador)

# Procesar Ficheros Excel: Grabar

- Para grabar un DataFrame en un fichero Excel, necesitamos, el módulo: **openpyxl**

- En una consola de administración:
  **pip install openpyxl**

- Ejemplo:
  from pandas import ExcelWriter
  writer = ExcelWriter('fichero.xslx')
  unDataFrame.**to_excel**(writer, 'hojaExcel')
  writer.save()

- También puede pasar una ruta de archivo to_excely evitar ExcelWriter:
  - frame.**to_excel**('examples/ex2.xlsx')

# Bases de datos

- Con La librería pandas también podemos cargar una tabla de la BD en un DataFrame.

- De forma manual:
  - Desde la parte de SQL, podemos utilizar el método fecthall que devuelve una tupla de tuplas (se convertirá a una lista de tuplas).

  - Y para las columnas tener en cuenta la propiedad: cursor.description. En la posición 0, viene el nombre del campo.

  - Una vez tenemos las dos listas podemos construir un DataFrame.

# Bases de datos con pandas.io.sql

- Pandas dispone de este módulo que permite conectar con una BD y recuperar un DataFrame con el resultado de una consulta sql.

- Necesitamos una conexión con la BD.
  - import pandas.io.sql as sql
  - dt = sql.**read_sql**(sql, conexionBD)
  - Nos devuelve un DataFrame

# Formato Json

- JSON (abreviatura de JavaScript Object Notation) tiene convertirse en uno de los formatos estándar para enviar datos mediante solicitud HTTP entre navegadores web y otras aplicaciones.

- Es un formato de datos de forma mucho más libre que un formulario de texto tabular como CSV.

```
obj = """
{"name": "Wes",
 "places_lived": ["United States", "Spain", "Germany"],
 "pet": null,
 "siblings": [{"name": "Scott", "age": 30, "pets": ["Zeus", "Zuko"]},
              {"name": "Katie", "age": 38,
               "pets": ["Sixes", "Stache", "Cisco"]}]
}
"""
```

# Procesar ficheros json

- pandas.read_json(path_or_buf=None, orient=None, typ='frame', dtype=None, convert_axes=None, convert_dates=True, keep_default_dates=True, numpy=False, precise_float=False, date_unit=None, encoding=None, lines=False, chunksize=None, compression='infer’)

- Indicar el **path** del fichero que queremos cargar en el DataFrame.
- https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_json.html
- Podemos convertir un DataFrame a una cadena JSON: **DataFrame.to_json** o una Serie a JSON con: **Series.to_json**

# Procesar ficheros en json

- Disponemos del módulo **json** con los métodos **dumps** (para parsear una estructura **json** y convertirla a **string**).

- **Formatear la impresión:**

```
print(json.dumps({'4':  5, '6': 7}, sort_keys=True,  indent=4))
{
        "4": 5,
        "6": 7
}
```

- Y el método **loads** le pasamos una cadena **json** y nos devuelve una estructura Python.

- Para **grabar** en un fichero disponemos del módulo **dump** serializa un objeto en formato json a un fichero. **json.dump(obj, fichero)**

- Para leer de un fichero y cargar en **json.load(fichero)** lo devuelve como un objeto Python.

# Procesar ficheros json

- pandas dispone de la función **read_json**()

```
In [70]: data = pd.read_json('examples/example.json')

In [71]: data
Out[71]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

```
In [69]: !cat examples/example.json
[{"a": 1, "b": 2, "c": 3},
 {"a": 4, "b": 5, "c": 6},
 {"a": 7, "b": 8, "c": 9}]
```

# Formato HDF5

- HDF5 es un formato de archivo destinado a almacenar grandes cantidades de datos de matrices científicas.
- Está disponible como una biblioteca C y tiene interfaces disponibles en muchos otros lenguajes, incluidos Java, Julia, MATLAB y Python.
- El "HDF" en HDF5 significa formato de datos jerárquico.
- HDF5 admite la compresión sobre la marcha con una variedad de modos de compresión, lo que permite que los datos con patrones repetidos se almacenen de manera más eficiente.

# Ejemplo

```
In [93]: frame = pd.DataFrame({'a': np.random.randn(100)})

In [94]: store = pd.HDFStore('mydata.h5')

In [95]: store['obj1'] = frame

In [96]: store['obj1_col'] = frame['a']

In [97]: store
Out[97]:
<class 'pandas.io.pytables.HDFStore'>
File path: mydata.h5
```

Los objetos contenidos en el archivo **HDF5** se pueden recuperar con la misma API tipo dict: **store['obj1']**

# Formato HDF5

- **HDFStore** admite dos esquemas de almacenamiento **'fixed'** y **'table'**.

- Este último es generalmente más lento, pero admite **operaciones de consulta** utilizando una sintaxis especial:

```
In [99]: store.put('obj2', frame, format='table')

In [100]: store.select('obj2', where=['index >= 10 and index <= 15'])
Out[100]:
           a
10  1.007189
11 -1.296221
12  0.274992
13  0.228913
14  1.352917
15  0.886429

In [101]: store.close()
```

# Formato HDF5

- pandas proporciona las funciones **to_hdf()** y **read_hdf()**

```
In [102]: frame.to_hdf('mydata.h5', 'obj3', format='table')

In [103]: pd.read_hdf('mydata.h5', 'obj3', where=['index < 5'])
Out[103]:
          a
0 -0.204708
1  0.478943
2 -0.519439
3 -0.555730
4  1.965781
```

**HDF5** no es una base de datos. Es más adecuado para conjuntos de datos de escritura única y lectura múltiple.
Si bien se pueden agregar datos a un archivo en cualquier momento, si varios escritores lo hacen simultáneamente, el archivo puede dañarse.

# Formatos de datos binarios

- Los objetos pandas permiten la serialización haciendo uso del módulo de pickle.
- Disponen del método **to_pickle()**

```
In [88]: frame = pd.read_csv('examples/ex1.csv')

In [89]: frame
Out[89]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo

In [90]: frame.to_pickle('examples/frame_pickle')
```

# Formatos de datos binarios

- Podemos leer cualquier objeto almacenado en un archivo utilizando el módulo de pickle directamente, o incluso más convenientemente usando la función **pandas.read_pickle.**

```
In [91]: pd.read_pickle('examples/frame_pickle')
Out[91]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

El formato pickle sólo se recomienda a **Corto plazo**, ya con el tiempo puede cambiar El formato de la biblioteca en Python.

# XML y HTML

- Python tiene muchas bibliotecas para leer y escribir datos en los formatos HTML y XML.

- Los ejemplos incluyen **lxml** , **Beautiful Soup** y **html5lib**.
  - Mientras **lxml** es comparativamente mucho **más rápido** en general, **las otras bibliotecas** pueden manejar mejor **archivos HTML o XML mal formados**.

- pandas tiene una función incorporada **read_html**, que usa bibliotecas como **lxml** y **Beautiful Soup** para analizar automáticamente tablas de archivos HTML como objetos DataFrame.

- Librerías necesarias:
  - **pip install beautifulsoup4 lxml html5lib**
  - **conda install beautifulsoup4 lxml html5lib**

# XML y HTML

- La función **pandas.read_html** tiene varias opciones, pero de forma predeterminada busca e intenta analizar todos los datos tabulares contenidos en las etiqueta <table>.

- El resultado es una lista de objetos DataFrame:
  - tables = pd.**read_html**('examples/fdic_failed_bank_list.html')

# Analizar XML con LXML.OBJECTIFY

- XML (lenguaje de marcado extensible) es otro formato de datos estructurado común que admite datos jerárquicos y anidados con metadatos.
- Usando **lxml.objectify**, analizamos el archivo y obtenemos una referencia al nodo raíz del archivo XML con **getroot**:

```
from lxml import objectify

path = 'datasets/mta_perf/Performance_MNR.xml'
parsed = objectify.parse(open(path))
root = parsed.getroot()
```

# Analizar XML con LXML.OBJECTIFY

- root.INDICATOR devuelve un generador que produce cada elemento XML <INDICATOR> (*sería una etiqueta del archivo*).

- Para cada registro, podemos completar un dict de nombres de etiquetas (como YTD_ACTUAL) a valores de datos (excluyendo algunas etiquetas):

```
data = []

skip_fields = ['PARENT_SEQ', 'INDICATOR_SEQ',
               'DESIRED_CHANGE', 'DECIMAL_PLACES']

for elt in root.INDICATOR:
    el_data = {}
    for child in elt.getchildren():
        if child.tag in skip_fields:
            continue
        el_data[child.tag] = child.pyval
    data.append(el_data)
```

```
In [82]: perf = pd.DataFrame(data)

In [83]: perf.head()
Out[83]:
              AGENCY_NAME             CATEGORY  \
 0  Metro-North Railroad  Service Indicators
1  Metro-North Railroad  Service Indicators
2  Metro-North Railroad  Service Indicators
3  Metro-North Railroad  Service Indicators
4  Metro-North Railroad  Service Indicators
```

# Pandas

**Tablas Pivot**

# Tablas Pivot

| | Product_Code | Warehouse | Product_Category | Date | Order_Demand | Dayofweek | Month | Year | Period | User |
|---|---|---|---|---|---|---|---|---|---|---|
| **35429** | Product_1286 | Whse_J | Category_019 | 2012-03-30 | 2000.000 | 4 | 3 | 2012 | 2012-03 | user 1 |
| **53043** | Product_1359 | Whse_J | Category_019 | 2012-04-16 | 50000.000 | 0 | 4 | 2012 | 2012-04 | user 2 |
| **577455** | Product_1453 | Whse_J | Category_019 | 2014-05-13 | 1200.000 | 1 | 5 | 2014 | 2014-05 | user 1 |
| **159301** | Product_1891 | Whse_J | Category_021 | 2012-03-19 | 12.000 | 0 | 3 | 2012 | 2012-03 | user 2 |
| **767518** | Product_1891 | Whse_A | Category_021 | 2015-03-04 | 19.000 | 2 | 3 | 2015 | 2015-03 | user 2 |
| **491114** | Product_1382 | Whse_J | Category_019 | 2014-07-23 | 1000.000 | 2 | 7 | 2014 | 2014-07 | user 2 |
| **327890** | Product_2055 | Whse_A | Category_021 | 2013-10-02 | 10.000 | 2 | 10 | 2013 | 2013-10 | user 1 |

- Partimos del **DataFrame** y podemos realizar operaciones de agrupado por **Year**:
  - **df.groupby(['Year']).size()**

| Year | size |
|---|---|
| **2011** | 640 |
| **2012** | 203635 |
| **2013** | 218298 |
| **2014** | 216404 |
| **2015** | 209661 |
| **2016** | 188645 |
| **2017** | 53 |

  - Podemos utilizar pivot_table para realizar operaciones de este tipo:
  - **pd.pivot_table(df, index=['Year'], aggfunc=['size'])**

# Tablas Pivot

- la ***tabla pivot*** puede asemejarse a realizar un ***grupo***, lo que necesitamos:
  - Un objeto **DataFrame**,
  - un **índice** sobre el cual realizaremos la agrupación de los valores
  - Y una **función** a aplicar sobre ellos.

- En cambio la utilidad y el mayor poder de las tablas **pivot** reside en utilizarlas, justamente, como tablas, es de decir con parámetros de **índice** (filas) y **columnas**.

# Tablas Pivot

- **pivot_table** tiene una mejor respuesta para agrupaciones por dos columnas que la función **groupby**:
  - df.groupby([‘Year’, ‘Month’]).size().to_frame()[:15]

| | | 0 |
|---|---|---|
| Year | Month | |
| 2011 | 1 | 1 |
| | 5 | 1 |
| | 6 | 2 |
| | 9 | 4 |
| | 10 | 3 |
| | 11 | 31 |
| | 12 | 598 |
| 2012 | 1 | 15614 |
| | 2 | 18123 |
| | 3 | 18604 |
| | 4 | 16590 |
| | 5 | 17014 |
| | 6 | 16911 |
| | 7 | 17718 |
| | 8 | 16492 |

# Tablas Pivot

- O, también podríamos hacer una **tabla pivot**, usando como índice el **Año**, y como columna los **Meses**, donde la visualización de los datos es mejor y a su vez las posibilidades de calculo son mayores.
- pd.**pivot_table**(df, index=['Year'], columns=['Month'], aggfunc=['size'])

| | size | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Month | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Year | | | | | | | | | | | | |
| 2011 | 1.000 | nan | nan | nan | 1.000 | 2.000 | nan | nan | 4.000 | 3.000 | 31.000 | 598.000 |
| 2012 | 15614.000 | 18123.000 | 18604.000 | 16590.000 | 17014.000 | 16911.000 | 17718.000 | 16492.000 | 15613.000 | 18515.000 | 17416.000 | 15025.000 |
| 2013 | 16638.000 | 17119.000 | 17397.000 | 17685.000 | 17786.000 | 16421.000 | 19085.000 | 16660.000 | 18946.000 | 24546.000 | 18617.000 | 17398.000 |
| 2014 | 18013.000 | 18214.000 | 19839.000 | 18077.000 | 16639.000 | 17762.000 | 18867.000 | 16021.000 | 18970.000 | 19579.000 | 17486.000 | 16937.000 |
| 2015 | 18245.000 | 18116.000 | 19122.000 | 17603.000 | 15572.000 | 18413.000 | 19127.000 | 15507.000 | 16656.000 | 17785.000 | 17335.000 | 16180.000 |
| 2016 | 14515.000 | 16130.000 | 17282.000 | 15223.000 | 14487.000 | 16418.000 | 15319.000 | 15333.000 | 15376.000 | 16191.000 | 17335.000 | 15036.000 |
| 2017 | 53.000 | nan | nan | nan | nan | nan | nan | nan | nan | nan | nan | nan |

# Tabla pivot vs groupby

- El resultado de aplicar **pivot_table** a un **DataFrame** es otro DataFrame

- Cuando aplicamos **groupby** obtenemos una **Serie**.

# Índices múltiple

- Con pivot_table se puede crear un índice múltiple utilizando dos niveles → dos columnas.

- Ejemplo, índice multinivel con las columnas: Product_Code y Year

  - pd.pivot_table(df, index=[‘**Product_Code**’, ‘**Year**’], aggfunc=[‘size’])[:10]

| | | size |
|---|---|---|
| **Product_Code** | **Year** | |
| **Product_0001** | **2012** | 139 |
| | **2013** | 117 |
| | **2014** | 129 |
| | **2015** | 107 |
| | **2016** | 103 |
| **Product_0002** | **2012** | 83 |
| | **2013** | 97 |
| | **2014** | 127 |
| | **2015** | 68 |
| | **2016** | 70 |

# Índices múltiple

- pd.pivot_table(df, index=[‘Product_Code’, ’Year’, ‘Month’], aggfunc=[‘size’])[:30]

| Product_Code | Year | Month | size |
|---|---|---|---|
| Product_0001 | 2012 | 1 | 15 |
| | | 2 | 13 |
| | | 3 | 13 |
| | | 4 | 7 |
| | | 5 | 10 |
| | | 6 | 7 |
| | | 7 | 10 |
| | | 8 | 14 |
| | | 9 | 9 |
| | | 10 | 20 |
| | | 11 | 11 |
| | | 12 | 10 |
| | 2013 | 1 | 7 |

# Índices múltiple + columnas

- pd.pivot_table(df, index=[‘**Product_Code**’, ‘**Year**’],columns=[‘Warehouse’], aggfunc=[‘size’])[:10]

| | | size | | | |
|---|---|---|---|---|---|
| | Warehouse | Whse_A | Whse_C | Whse_J | Whse_S |
| Product_Code | Period | | | | |
| Product_0001 | 2012-01 | 7.000 | nan | 8.000 | nan |
| | 2012-02 | 5.000 | nan | 8.000 | nan |
| | 2012-03 | 6.000 | nan | 7.000 | nan |
| | 2012-04 | 2.000 | nan | 5.000 | nan |
| | 2012-05 | 4.000 | nan | 6.000 | nan |
| | 2012-06 | 1.000 | nan | 6.000 | nan |
| | 2012-07 | 3.000 | nan | 7.000 | nan |
| | 2012-08 | 8.000 | nan | 6.000 | nan |
| | 2012-09 | 5.000 | nan | 4.000 | nan |

Existen valores **NaN** para algunas Almacenes, esto significa que para ese producto en ese periodo y para esa almacén no existe un registro.

# Relleno de Ceros

- Las celdas donde se da el valor **NaN** se pueden rellenar de ceros. Podemos utilizar el parámetro **fill_value** = 0
- pd.pivot_table(df, index=[‘Product_Code’,’Period’], columns=[‘Warehouse’], aggfunc=[‘size’], **fill_value**=0)

# Parámetro aggfunc

- La función aggfunc se puede sustituir por count , o se puede añadir varias → aggfunc=[‘count’,’size’]

| | | count | size |
|---|---|---|---|
| | | Order_Demand | 0 |
| Product_Code | Period | | |
| Product_0001 | 2012-01 | 12 | 15 |
| | 2012-02 | 13 | 13 |
| | 2012-03 | 12 | 13 |
| | 2012-04 | 7 | 7 |
| | 2012-05 | 0 | 10 |
| | 2012-06 | 6 | 7 |
| | 2012-07 | 10 | 10 |
| | 2012-08 | 13 | 14 |

- Las funciones se pueden aplicar sobre una determinada columna, para ello utilizamos el parámetro **values**
- **pd.pivot_table(df, index=[‘Product_Code’,’Period’], values=[‘Order_Demand’], aggfunc=[‘count’,’size’], fill_value=0)[:10]**

# Series temporales con pandas

- Con la librería pandas se pueden crear series temporales para ello podemos utilizar la función **pd.date_range** con una fecha **inicial** y otra **final**, además de indicar la **frecuencia**.

```
import pandas as pd
from datetime import datetime
import numpy as np

date_rng = pd.date_range(start='2018/07/01', end='2018/07/15', freq='H')
print(date_rng)
```

# Series temporales con pandas

```
print(date_range)
DatetimeIndex(['2018-07-01 00:00:00', '2018-07-01 01:00:00',
               '2018-07-01 02:00:00', '2018-07-01 03:00:00',
               '2018-07-01 04:00:00', '2018-07-01 05:00:00',
               '2018-07-01 06:00:00', '2018-07-01 07:00:00',
               '2018-07-01 08:00:00', '2018-07-01 09:00:00',
               ...
               '2018-07-14 15:00:00', '2018-07-14 16:00:00',
               '2018-07-14 17:00:00', '2018-07-14 18:00:00',
               '2018-07-14 19:00:00', '2018-07-14 20:00:00',
               '2018-07-14 21:00:00', '2018-07-14 22:00:00',
               '2018-07-14 23:00:00', '2018-07-15 00:00:00'],
              dtype='datetime64[ns]', length=337, freq='H')
```

# Series temporales con pandas

- A partir de la serie temporal se puede crear un DataFrame:
  - ts = pd.DataFrame(date_rng, columns=['date'])
  - ts['data'] = np.random.randint(0,100,size=(len(**date_rng**)))
  - ts.head(5) **# los 5 primeros datos.**

  - ts = ts.**set_index**('date') **# Indexar por la columna date.**
  - ts.head(5)

  - Para **filtrar** por el campo **date**:
    - **df[(df['date']>pd.to_datetime('2018-07-01')) & (df['date']<pd.to_datetime('2018-07-04'))]**
    - *En este caso filtramos el rango de fechas entre el 1 de Julio y el 3 de julio*

# Visualización de datos

- # Mostrar todas las filas del DataFrame o de la Serie:
  - pd.set_option('display.max_rows', None)

# Enlaces

- https://www.kaggle.com/datasets
- https://medium.com/@matiasscorsetti/ingenier%C3%ADa-de-variables-con-pivot-table-pandas-5f4a0e0a8454