Máster en

Advanced Analytics on **Big Data**

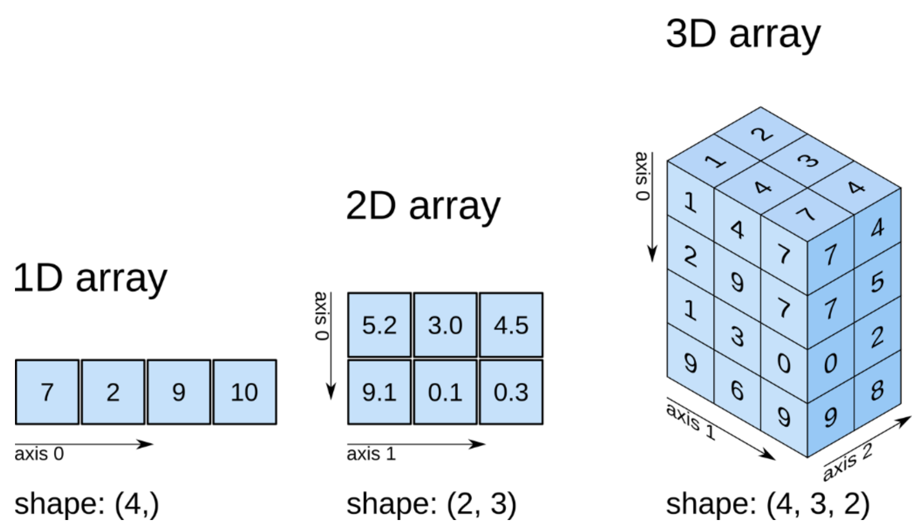
Principales Bibliotecas en Python

Ponente: José Manuel García Nieto

- Bibliotecas orientadas a las funciones principales en el análisis de datos
  - Cálculo numérico y estadístico (Numpy, Scipy)
  - Carga y manejo de datos (Pandas)
  - Visualización (Matplotlib)

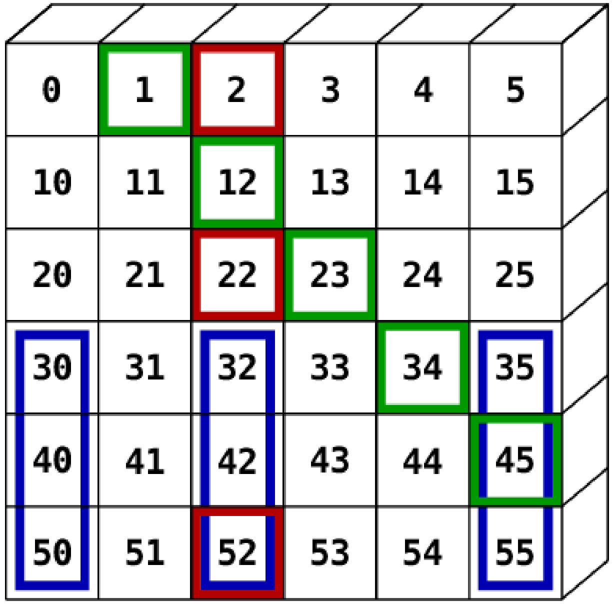
- Numpy
  - Principal biblioteca para la computación científica en Python
  - Genera objetos multidimensionales de tipo “Array” y herramientas para trabajar con estos arrays
  - Toma el modelo de datos y cálculo de Matlab
  - Referencia http://docs.scipy.org/doc/numpy/reference/

- Numpy Array
  - Es una matriz o rejilla de valores del mismo tipo (int, float, bool, etc.)
  - Se indexa mediante una tupla de valores enteros no negativos
  - El número de dimensiones es el rango (rank) del array
  - La forma del array (shape) es una tupla de enteros que contienen el tamaño del array para cada dimensión



- Numpy Array
  - Se indexa mediante una tupla de valores enteros no negativos

```
>>> a[(0,1,2,3,4), (1,2,3,4,5)]
array([1, 12, 23, 34, 45])

>>> a[3:, [0,2,5]]
array([[30, 32, 35],
       [40, 42, 45],
       [50, 52, 55]])

>>> mask = np.array([1,0,1,0,0,1], dtype=bool)
>>> a[mask, 2]
array([2, 22, 52])
```



- Numpy Array
  - Los arrays se pueden inicializar mediante listas Python
  - Se acceden a los elementos mediante índices en corchetes [i]

```
import numpy as np

a = np.array([1, 2, 3])             # Create a rank 1 array
print(type(a))                      # Prints "<class 'numpy.ndarray'>"
print(a.shape)                      # Prints "(3,)"
print(a[0], a[1], a[2])             # Prints "1 2 3"
a[0] = 5                            # Change an element of the array
print(a)                            # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])     # Create a rank 2 array
print(b.shape)                      # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])    # Prints "1 2 4"
```

- Numpy Array
  - Ofrece funciones para crear arrays desde cero

```
import numpy as np

a = np.zeros((2,2))          # Create an array of all zeros
print(a)                     # Prints "[[ 0.  0.] [ 0.  0.]]"

b = np.ones((1,2))           # Create an array of all ones
print(b)                     # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7)        # Create a constant array
print(c)                     # Prints "[[ 7.  7.]
                             #          [ 7.  7.]]"

d = np.eye(2)                # Create a 2x2 identity matrix
print(d)                     # Prints "[[ 1.  0.]
                             #          [ 0.  1.]]"

e = np.random.random((2,2))  # Create an array filled with random values
print(e)                     # Might print "[[ 0.91940167  0.08143941]
                             #               [ 0.68744134  0.87236687]]"
```

- Numpy Array Indexing
  - Ofrece varias formas:
    - Directamente (slicing) mediante subarrays, igual que las listas

```
import numpy as np

a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])  # Create the following rank 2 array with shape (3, 4)
                                                     # [[ 1  2  3  4]
                                                     #  [ 5  6  7  8]
                                                     #  [ 9 10 11 12]]

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
b = a[:2, 1:3]                                       # [[2 3]
                                                     #  [6 7]]

# A slice of an array is a view into the same data, so modifying it will modify the original array.
print(a[0, 1])                                       # Prints "2"
b[0, 0] = 77                                         # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])                                       # Prints "77"
```

- Numpy Array Indexing
  - Indexado entero:
    - Permite construir otros arrays a partir del original

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

                                          # An example of integer array indexing.
                                          # The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])            # Prints "[1 4 5]"

# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))          # Prints "[1 4 5]"

# When using integer array indexing, you can reuse the same element from the source array:
print(a[[0, 0], [1, 1]])                  # Prints "[2 2]"

# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))       # Prints "[2 2]"
```

- Numpy Array Indexing
  - Indexado booleano:
    - Permite indexar mediante condiciones booleanas

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2)              # Find the elements of a that are bigger than 2;
                                # this returns a numpy array of Booleans of the same shape as a,
                                # where each slot of bool_idx tells whether element of a is > 2.

print(bool_idx)                 # Prints "[[False False]
                                #          [ True  True]
                                #          [ True  True]]"

print(a[bool_idx])              # We use boolean array indexing to construct a rank 1 array
                                # consisting of the elements of a corresponding to the True
                                # values of bool_id.  Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print(a[a > 2])                 # Prints "[3 4 5 6]"
```

- Numpy Array Math
  - Operaciones matemáticas sobre arrays

```
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

                                        # Elementwise sum; both produce the array
                                        # [[ 6.0  8.0]
                                        #  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

                                        # Elementwise difference; both produce the array
                                        # [[-4.0 -4.0]
                                        #  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))
```

- Numpy Array Math
  - Operaciones matemáticas sobre arrays

```
# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

- Numpy Array Math
  - El operador * realiza la multiplicación entre los elementos
  - Para la multiplicación de matrices se usa la función dot()

```
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

print(v.dot(w))                 # Inner product of vectors; both produce 219
print(np.dot(v, w))

print(x.dot(v))                 # Matrix / vector product; both produce the rank 1 array [29 67]
print(np.dot(x, v))

print(x.dot(y))                 # Matrix / matrix product; both produce the rank 2 array
print(np.dot(x, y))             # [[19 22]
                                #  [43 50]]
```

- Numpy Array Math
  - Otra función muy útil para el cómputo con arrays es sum()

```
import numpy as np

x = np.array([[1,2],[3,4]])

print(np.sum(x))                    # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))            # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))            # Compute sum of each row; prints "[3 7]"
```

- Numpy Array Math
  - También muy útil el cálculo de la traspuesta, mediante el atributo T

```
import numpy as np

x = np.array([[1,2], [3,4]])
print(x)                        # Prints "[[1 2]
                                #          [3 4]]"

print(x.T)                      # Prints "[[1 3]
                                #          [2 4]]"

                                # Note that taking the transpose of a rank 1 array does nothing:
v = np.array([1,2,3])
print(v)                        # Prints "[1 2 3]"
print(v.T)                      # Prints "[1 2 3]"
```

- Numpy Array Math Broadcasting
  - Permite realizar operaciones con arrays de diferentes formas
  - Muy común, aplicar el array pequeño mediante operaciones sobre el array grande (ej., procesamiento de imagen)

```
import numpy as np

                                        # We will add the vector v to each row of the matrix x,
                                        # storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)   # Create an empty matrix with the same shape as x

for i in range(4):          # Add the vector v to each row of the matrix x with an explicit loop
    y[i, :] = x[i, :] + v

print(y)                    # Now y is the following
                            # [[ 2  2  4]
                            #  [ 5  5  7]
                            #  [ 8  8 10]
                            #  [11 11 13]]
```

- Pandas
  - Biblioteca para la carga y el manejo eficiente de datos
  - Dos estructuras básicas: Series y Dataframe
  - Permite índices y etiquetas en los datos
  - Soporta gran variedad de cálculos
  - Se utiliza en gran número de aplicaciones Python
  - Referencia: https://pandas.pydata.org

- Pandas
  - Biblioteca para la carga y el manejo eficiente de datos

```
In [1]: import pandas as pd

In [2]: import numpy as np
```

  - Creación de estructuras: Series
    - Se deja a Pandas crear un índice de valores enteros

```
In [4]: s = pd.Series([1,3,5,np.nan,6,8])

In [5]: s
Out[5]:
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

- Pandas
  - Creación de estructuras: Dataframe
    - Ej. Dataframe a partir de Numpy array, con datatime de índice y columnas etiquetadas

```
In [6]: dates = pd.date_range('20130101', periods=6)
In [7]: dates
Out[7]:
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04', '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')

In [8]: df = pd.DataFrame(np.random.randn(6,4), index=dates, columns=list('ABCD'))
In [9]: df
Out[9]:
                   A         B         C         D
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
2013-01-05 -0.424972  0.567020  0.276232 -1.087401
2013-01-06 -0.673690  0.113648 -1.478427  0.524988
```

- Pandas
  - Creación de Dataframe a partir de diccionario de objetos que se pueden convertir en forma de series

```
In [10]: df2 = pd.DataFrame({ 'A' : 1.,
   ....:                      'B' : pd.Timestamp('20130102'),
   ....:                      'C' : pd.Series(1,index=list(range(4)),dtype='float32'),
   ....:                      'D' : np.array([3] * 4,dtype='int32'),
   ....:                      'E' : pd.Categorical(["test","train","test","train"]),
   ....:                      'F' : 'foo' })
   ....:

In [11]: df2
Out[11]:
     A          B    C  D      E    F
0  1.0 2013-01-02  1.0  3   test  foo
1  1.0 2013-01-02  1.0  3  train  foo
2  1.0 2013-01-02  1.0  3   test  foo
3  1.0 2013-01-02  1.0  3  train  foo
```

- Pandas
  - Visualización inicial de datos de Dataframe: head() y tail()

```
In [14]: df.head()
Out[14]:
                   A         B         C         D
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
2013-01-05 -0.424972  0.567020  0.276232 -1.087401

In [15]: df.tail(3)
Out[15]:
                   A         B         C         D
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
2013-01-05 -0.424972  0.567020  0.276232 -1.087401
2013-01-06 -0.673690  0.113648 -1.478427  0.524988
```

- Pandas
  - Visualización del índice (.index), las columnas (.columns) y el Numpy array (.values)

```
In [16]: df.index
Out[16]:
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')

In [17]: df.columns
Out[17]: Index(['A', 'B', 'C', 'D'], dtype='object')

In [18]: df.values
Out[18]:
array([[ 0.4691, -0.2829, -1.5091, -1.1356],
       [ 1.2121, -0.1732,  0.1192, -1.0442],
       [-0.8618, -2.1046, -0.4949,  1.0718],
       [ 0.7216, -0.7068, -1.0396,  0.2719],
       [-0.425 ,  0.567 ,  0.2762, -1.0874],
       [-0.6737,  0.1136, -1.4784,  0.525 ]])
```

- Pandas
  - Primeras estadísticas. La función describe()

```
In [19]: df.describe()
Out[19]:
              A         B         C         D
count  6.000000  6.000000  6.000000  6.000000
mean   0.073711 -0.431125 -0.687758 -0.233103
std    0.843157  0.922818  0.779887  0.973118
min   -0.861849 -2.104569 -1.509059 -1.135632
25%   -0.611510 -0.600794 -1.368714 -1.076610
50%    0.022070 -0.228039 -0.767252 -0.386188
75%    0.658444  0.041933 -0.034326  0.461706
max    1.212112  0.567020  0.276232  1.071804
```

- Pandas
  - Traspuesta del dataframe T

```
In [20]: df.T
Out[20]:
   2013-01-01  2013-01-02  2013-01-03  2013-01-04  2013-01-05  2013-01-06
A    0.469112    1.212112   -0.861849    0.721555   -0.424972   -0.673690
B   -0.282863   -0.173215   -2.104569   -0.706771    0.567020    0.113648
C   -1.509059    0.119209   -0.494929   -1.039575    0.276232   -1.478427
D   -1.135632   -1.044236    1.071804    0.271860   -1.087401    0.524988
```

- Pandas: Ordenación por eje (axis)

```
In [21]: df.sort_index(axis=1, ascending=False)
Out[21]:
                   D         C         B         A
2013-01-01 -1.135632 -1.509059 -0.282863  0.469112
2013-01-02 -1.044236  0.119209 -0.173215  1.212112
2013-01-03  1.071804 -0.494929 -2.104569 -0.861849
2013-01-04  0.271860 -1.039575 -0.706771  0.721555
2013-01-05 -1.087401  0.276232  0.567020 -0.424972
2013-01-06  0.524988 -1.478427  0.113648 -0.673690
```

```
In [22]: df.sort_values(by='B')
Out[22]:
                   A         B         C         D
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-06 -0.673690  0.113648 -1.478427  0.524988
2013-01-05 -0.424972  0.567020  0.276232 -1.087401
```

- Pandas: Selección de valores
  - Selección de una columna, por lo que se obtiene una Serie

```
n [23]: df['A']
Out[23]:
2013-01-01    0.469112
2013-01-02    1.212112
2013-01-03   -0.861849
2013-01-04    0.721555
2013-01-05   -0.424972
2013-01-06   -0.673690
Freq: D, Name: A, dtype: float64
```

- Pandas: Selección de valores
  - Selección mediante corchetes []

```
In [24]: df[0:3]
Out[24]:
                   A         B         C         D
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804

In [25]: df['20130102':'20130104']
Out[25]:
                   A         B         C         D
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
```

- Pandas: Selección de valores
  - Selección mediante corchetes [ ]

```
In [24]: df[0:3]
Out[24]:
                   A         B         C         D
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804

In [25]: df['20130102':'20130104']
Out[25]:
                   A         B         C         D
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
```

- Pandas: Selección por etiqueta
  - Mediante el tributo .loc[]

```
In [26]: df.loc[dates[0]]
Out[26]:
A    0.469112
B   -0.282863
C   -1.509059
D   -1.135632
Name: 2013-01-01 00:00:00, dtype: float64
```

```
In [27]: df.loc[:,['A','B']]
Out[27]:
                   A         B
2013-01-01  0.469112 -0.282863
2013-01-02  1.212112 -0.173215
2013-01-03 -0.861849 -2.104569
2013-01-04  0.721555 -0.706771
2013-01-05 -0.424972  0.567020
2013-01-06 -0.673690  0.113648
```

- Pandas: Selección por posición
  - Mediante el tributo .iloc[]

```
In [32]: df.iloc[3]
Out[32]:
A    0.721555
B   -0.706771
C   -1.039575
D    0.271860
Name: 2013-01-04 00:00:00, dtype: float64
```

```
In [33]: df.iloc[3:5,0:2]
Out[33]:
                   A         B
2013-01-04  0.721555 -0.706771
2013-01-05 -0.424972  0.567020
```

- Pandas: Selección por condición booleana

```
In [39]: df[df.A > 0]
Out[39]:
                   A         B         C         D
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
```

- Pandas: Inserción
  - Añadir una columna alineando los índices

```
In [45]: s1 = pd.Series([1,2,3,4,5,6], index=pd.date_range('20130102', periods=6))

In [46]: s1
Out[46]:
2013-01-02    1
2013-01-03    2
2013-01-04    3
2013-01-05    4
2013-01-06    5
2013-01-07    6
Freq: D, dtype: int64

In [47]: df['F'] = s1
```

```
In [48]: df.at[dates[0],'A'] = 0
```

- Pandas: Operaciones
  - Estadística descriptiva básica. Función mean()

```
In [61]: df.mean()
Out[61]:
A   -0.004474
B   -0.383981
C   -0.687758
D    5.000000
F    3.000000
dtype: float64
```

```
In [62]: df.mean(1)
Out[62]:
2013-01-01    0.872735
2013-01-02    1.431621
2013-01-03    0.707731
2013-01-04    1.395042
2013-01-05    1.883656
2013-01-06    1.592306
Freq: D, dtype: float64
```

- Pandas: Operaciones
  - Apply. Aplica una función determinada a todo el datframe
    - Ejemplo: suma acumulativa (cumsum)

```
In [66]: df.apply(np.cumsum)
Out[66]:
                   A         B         C   D     F
2013-01-01  0.000000  0.000000 -1.509059   5   NaN
2013-01-02  1.212112 -0.173215 -1.389850  10   1.0
2013-01-03  0.350263 -2.277784 -1.884779  15   3.0
2013-01-04  1.071818 -2.984555 -2.924354  20   6.0
2013-01-05  0.646846 -2.417535 -2.648122  25  10.0
2013-01-06 -0.026844 -2.303886 -4.126549  30  15.0
```

- Pandas: Operaciones
  - Histogramming y discretización. Función .value_counts()

```
In [68]: s = pd.Series(np.random.randint(0, 7, size=10))

In [69]: s
Out[69]:
0    4
1    2
2    1
3    2
4    6
5    4
6    4
7    6
8    4
9    4
dtype: int64
```

```
In [70]: s.value_counts()
Out[70]:
4    5
6    2
2    2
1    1
dtype: int64
```

- Pandas: Operaciones
  - Mezcla. Concatenar mediante .concat()

```
In [73]: df = pd.DataFrame(np.random.randn(10, 4))

In [74]: df
Out[74]:
          0         1         2         3
0 -0.548702  1.467327 -1.015962 -0.483075
1  1.637550 -1.217659 -0.291519 -1.745505
2 -0.263952  0.991460 -0.919069  0.266046
3 -0.709661  1.669052  1.037882 -1.705775
4 -0.919854 -0.042379  1.247642 -0.009920
5  0.290213  0.495767  0.362949  1.548106
6 -1.131345 -0.089329  0.337863 -0.945867
7 -0.932132  1.956030  0.017587 -0.016692
8 -0.575247  0.254161 -1.143704  0.215897
9  1.193555 -0.077118 -0.408530 -0.862495
```

```
# break it into pieces
In [75]: pieces = [df[:3], df[3:7], df[7:]]
```

```
In [76]: pd.concat(pieces)
Out[76]:
          0         1         2         3
0 -0.548702  1.467327 -1.015962 -0.483075
1  1.637550 -1.217659 -0.291519 -1.745505
2 -0.263952  0.991460 -0.919069  0.266046
3 -0.709661  1.669052  1.037882 -1.705775
4 -0.919854 -0.042379  1.247642 -0.009920
5  0.290213  0.495767  0.362949  1.548106
6 -1.131345 -0.089329  0.337863 -0.945867
7 -0.932132  1.956030  0.017587 -0.016692
8 -0.575247  0.254161 -1.143704  0.215897
9  1.193555 -0.077118 -0.408530 -0.862495
```

- Pandas: Agrupación
  - Mediante la operación “group by” nos referimos a un proceso que implica los siguientes pasos:
    - Splitting: separar los datos en grupos en base a algún criterio
    - Applying: aplicar una función a cada grupo de manera separada
    - Combining: combinar los resultados en una estructura común

```
In [91]: df = pd.DataFrame({'A' : ['foo', 'bar', 'foo', 'bar',
   ....:                          'foo', 'bar', 'foo', 'foo'],
   ....:                   'B' : ['one', 'one', 'two', 'three',
   ....:                          'two', 'two', 'one', 'three'],
   ....:                   'C' : np.random.randn(8),
   ....:                   'D' : np.random.randn(8)})
   ....:
```

```
In [92]: df
Out[92]:
     A      B         C         D
0  foo    one -1.202872 -0.055224
1  bar    one -1.814470  2.395985
2  foo    two  1.018601  1.552825
3  bar  three -0.595447  0.166599
4  foo    two  1.395433  0.047609
5  bar    two -0.392670 -0.136473
6  foo    one  0.007207 -0.561757
7  foo  three  1.928123 -1.623033
```

```
In [93]: df.groupby('A').sum()
Out[93]:
            C        D
A
bar -2.802588  2.42611
foo  3.146492 -0.63958
```

- Pandas: Agrupación
  - Al agrupar por varias columnas se genera un índice jerárquico
  - Podemos aplicar de nuevo la función sum()

```
In [94]: df.groupby(['A','B']).sum()
Out[94]:
                  C         D
A   B
bar one   -1.814470  2.395985
    three -0.595447  0.166599
    two   -0.392670 -0.136473
foo one   -1.195665 -0.616981
    three  1.928123 -1.623033
    two    2.414034  1.600434
```

- Pandas: Series Temporales
  - Representación de zona temporal

```
In [111]: rng = pd.date_range('3/6/2012 00:00', periods=5, freq='D')

In [112]: ts = pd.Series(np.random.randn(len(rng)), rng)

In [113]: ts
Out[113]:
2012-03-06    0.464000
2012-03-07    0.227371
2012-03-08   -0.496922
2012-03-09    0.306389
2012-03-10   -2.290613
Freq: D, dtype: float64
```

```
In [114]: ts_utc = ts.tz_localize('UTC’)

In [115]: ts_utc
Out[115]:
2012-03-06 00:00:00+00:00    0.464000
2012-03-07 00:00:00+00:00    0.227371
2012-03-08 00:00:00+00:00   -0.496922
2012-03-09 00:00:00+00:00    0.306389
2012-03-10 00:00:00+00:00   -2.290613
Freq: D, dtype: float64
```

```
In [116]: ts_utc.tz_convert('US/Eastern')
Out[116]:
2012-03-05 19:00:00-05:00    0.464000
2012-03-06 19:00:00-05:00    0.227371
2012-03-07 19:00:00-05:00   -0.496922
2012-03-08 19:00:00-05:00    0.306389
2012-03-09 19:00:00-05:00   -2.290613
Freq: D, dtype: float64
```

- Pandas: Series Temporales
  - Convertir espacios temporales (timespam)

```
In [117]: rng = pd.date_range('1/1/2012', periods=5, freq='M')

In [118]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [119]: ts
Out[119]:
2012-01-31   -1.134623
2012-02-29   -1.561819
2012-03-31   -0.260838
2012-04-30    0.281957
2012-05-31    1.523962
Freq: M, dtype: float64
```

```
In [120]: ps = ts.to_period()

In [121]: ps
Out[121]:
2012-01   -1.134623
2012-02   -1.561819
2012-03   -0.260838
2012-04    0.281957
2012-05    1.523962
Freq: M, dtype: float64
```

```
In [122]: ps.to_timestamp()
Out[122]:
2012-01-01   -1.134623
2012-02-01   -1.561819
2012-03-01   -0.260838
2012-04-01    0.281957
2012-05-01    1.523962
Freq: MS, dtype: float64
```

- Pandas: Carga/Escritura de datos (read_/to_)
  - read_/to_ https://pandas.pydata.org/pandas-docs/stable/io.html

IO Tools (Text, CSV, HDF5, …)

The pandas I/O API is a set of top level `reader` functions accessed like `pandas.read_csv()` that generally return a pandas object. The corresponding `writer` functions are object methods that are accessed like `DataFrame.to_csv()`. Below is a table containing available `readers` and `writers`.

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	Local clipboard	read_clipboard	to_clipboard
binary	MS Excel	read_excel	to_excel
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Parquet Format	read_parquet	to_parquet
binary	Msgpack	read_msgpack	to_msgpack
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google Big Query	read_gbq	to_gbq

Here is an informal performance comparison for some of these IO methods.

- Pandas: Carga/Escritura de datos (read_/to_)
  - CSV: read_csv / to_csv

```
In [141]: df.to_csv('foo.csv')
```

```
In [142]: pd.read_csv('foo.csv')
Out[142]:
    Unnamed: 0          A          B         C          D
0   2000-01-01   0.266457  -0.399641 -0.219582   1.186860
1   2000-01-02  -1.170732  -0.345873  1.653061  -0.282953
2   2000-01-03  -1.734933   0.530468  2.060811  -0.515536
3   2000-01-04  -1.555121   1.452620  0.239859  -1.156896
4   2000-01-05   0.578117   0.511371  0.103552  -2.428202
5   2000-01-06   0.478344   0.449933 -0.741620  -1.962409
6   2000-01-07   1.235339  -0.091757 -1.543861  -1.084753
..         ...        ...        ...       ...        ...
993 2002-09-20 -10.628548  -9.153563 -7.883146  28.313940
994 2002-09-21 -10.390377  -8.727491 -6.399645  30.914107
995 2002-09-22  -8.985362  -8.485624 -4.669462  31.367740
996 2002-09-23  -9.558560  -8.781216 -4.499815  30.518439
997 2002-09-24  -9.902058  -9.340490 -4.386639  30.105593
998 2002-09-25 -10.216020  -9.480682 -3.933802  29.758560
999 2002-09-26 -11.856774 -10.671012 -3.216025  29.369368

[1000 rows x 5 columns]
```

- Pandas: Carga/Escritura de datos (read_/to_)
  - CSV: read_excel / to_Excel

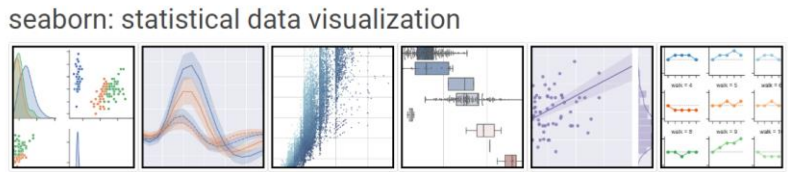
```
In [145]: df.to_excel('foo.xlsx', sheet_name='Sheet1')
```

```
In [146]: pd.read_excel('foo.xlsx', 'Sheet1', index_col=None, na_values=['NA'])
Out[146]:
                    A          B         C          D
2000-01-01   0.266457  -0.399641 -0.219582   1.186860
2000-01-02  -1.170732  -0.345873  1.653061  -0.282953
2000-01-03  -1.734933   0.530468  2.060811  -0.515536
2000-01-04  -1.555121   1.452620  0.239859  -1.156896
2000-01-05   0.578117   0.511371  0.103552  -2.428202
2000-01-06   0.478344   0.449933 -0.741620  -1.962409
2000-01-07   1.235339  -0.091757 -1.543861  -1.084753
...               ...        ...       ...        ...
2002-09-20 -10.628548  -9.153563 -7.883146  28.313940
2002-09-21 -10.390377  -8.727491 -6.399645  30.914107
2002-09-22  -8.985362  -8.485624 -4.669462  31.367740
2002-09-23  -9.558560  -8.781216 -4.499815  30.518439
2002-09-24  -9.902058  -9.340490 -4.386639  30.105593
2002-09-25 -10.216020  -9.480682 -3.933802  29.758560
2002-09-26 -11.856774 -10.671012 -3.216025  29.369368

[1000 rows x 4 columns]
```

- Visualización
  - Matplotlib

    https://matplotlib.org/
  - Seaborn

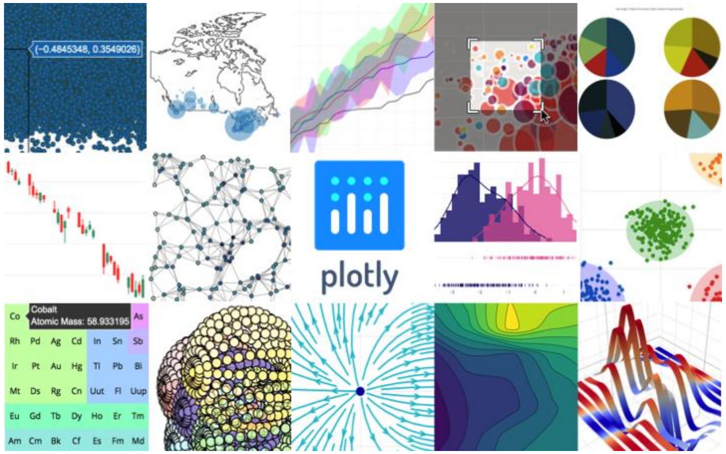
    https://seaborn.pydata.org/



  - Plotly

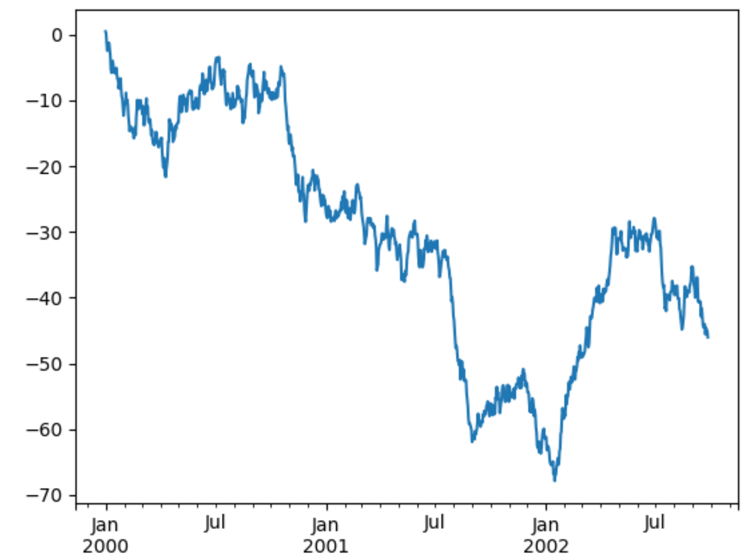
    https://plot.ly/

    https://plot.ly/python/



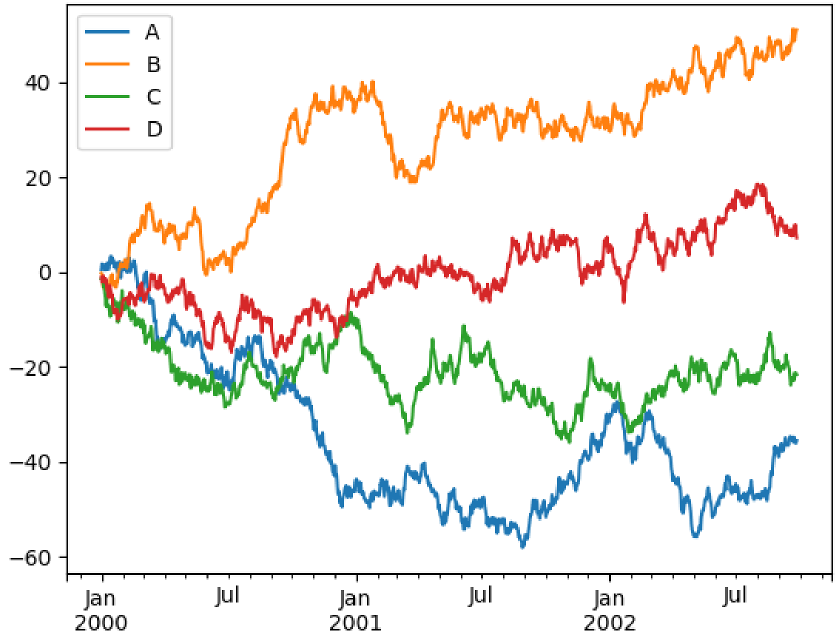
- Matplotlib: Ploteado. Primer paso hacia la visualización
  - Plotting

```
In [1]: import matplotlib.pyplot as plt
In [2]: ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000', periods=1000))
In [3]: ts = ts.cumsum()
In [4]: ts.plot()
Out[4]: <matplotlib.axes._subplots.AxesSubplot at 0x7f20d5690710>
```



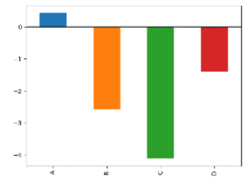
- Biblioteca Matplotlib para Visualización
  - Plotting

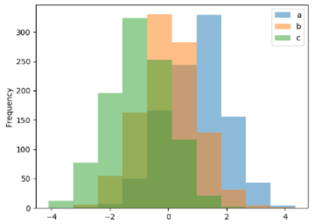
```
In [5]: df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index, columns=list('ABCD'))
In [6]: df = df.cumsum()
In [7]: plt.figure(); df.plot();
```

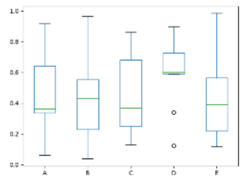


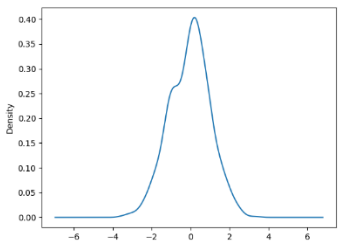
Biblioteca Matplotlib para Visualización

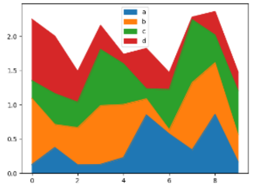
- Diagramas básicos
  - Barras: bar
  - Histograma : hist
  - Bloxplot: box
  - Densidad: kde o density
  - Area: área
  - Puntos: scatter
  - Sectores: pie

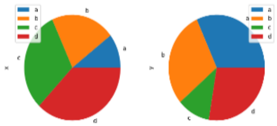


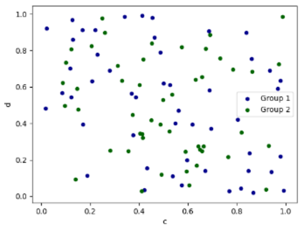






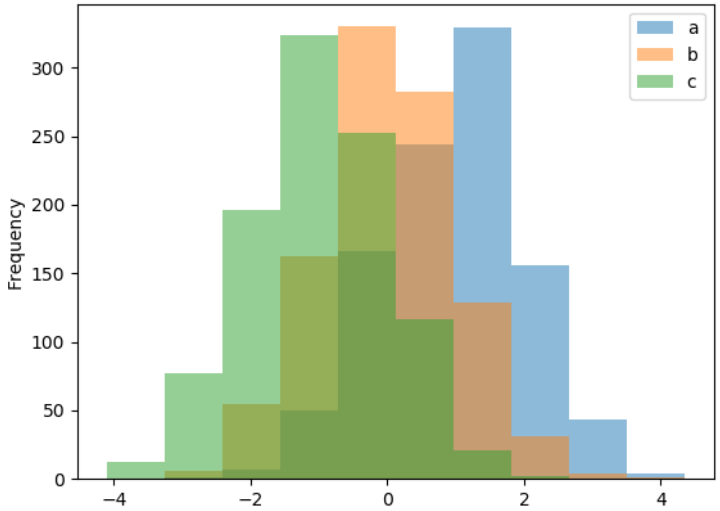






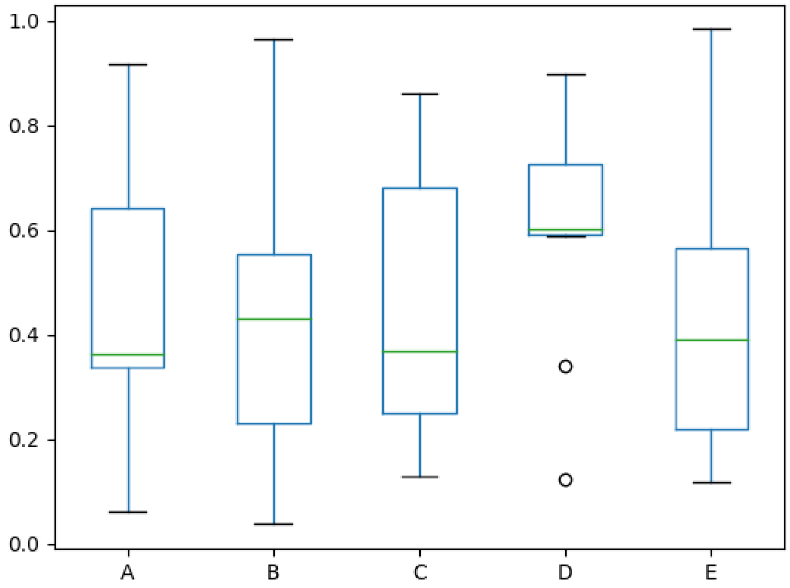
- Biblioteca Matplotlib para Visualización
  - Histograma

```
In [21]: df4 = pd.DataFrame({'a': np.random.randn(1000) + 1, 'b': np.random.randn(1000),
   ....:                    'c': np.random.randn(1000) - 1}, columns=['a', 'b', 'c'])
   ....:
In [22]: plt.figure();
In [23]: df4.plot.hist(alpha=0.5)
Out[23]: <matplotlib.axes._subplots.AxesSubplot at 0x7f20cf918908>
```



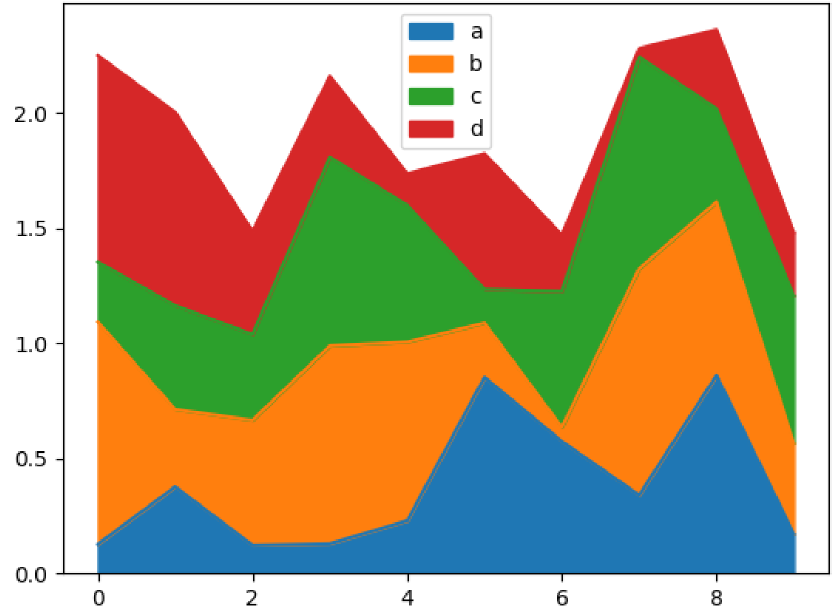
- Biblioteca Matplotlib para Visualización
  - Boxplot

```
In [34]: df = pd.DataFrame(np.random.rand(10, 5), columns=['A', 'B', 'C', 'D', 'E'])

In [35]: df.plot.box()
Out[35]: <matplotlib.axes._subplots.AxesSubplot at 0x7f20cf9400f0>
```

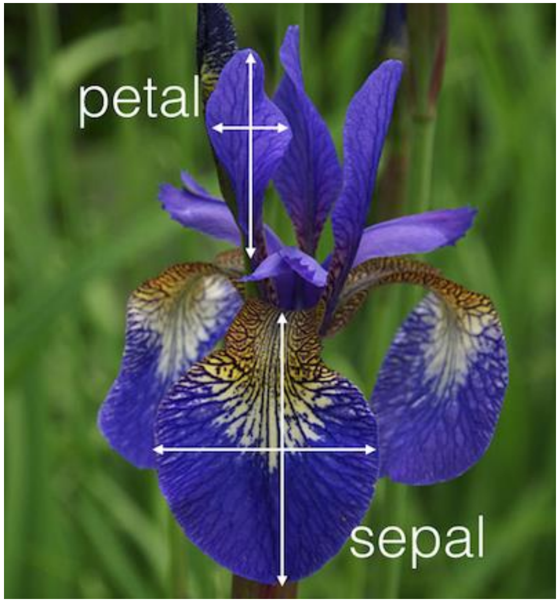


- Biblioteca Matplotlib para Visualización
  - Area

```
In [57]: df = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])

In [58]: df.plot.area();
```

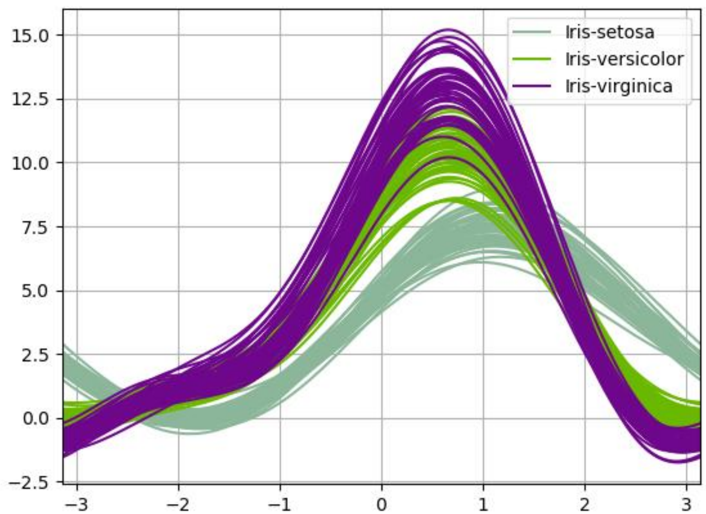


- Biblioteca Matplotlib para Visualización
  - Ploteando conjuntos de datos.
    - Ej. Flor Iris
    - https://raw.githubusercontent.com/pandas-dev/pandas/master/pandas/tests/data/iris.csv



- Biblioteca Matplotlib para Visualización
  - Curvas de Andrews: series de Fourier a partir de las muestras

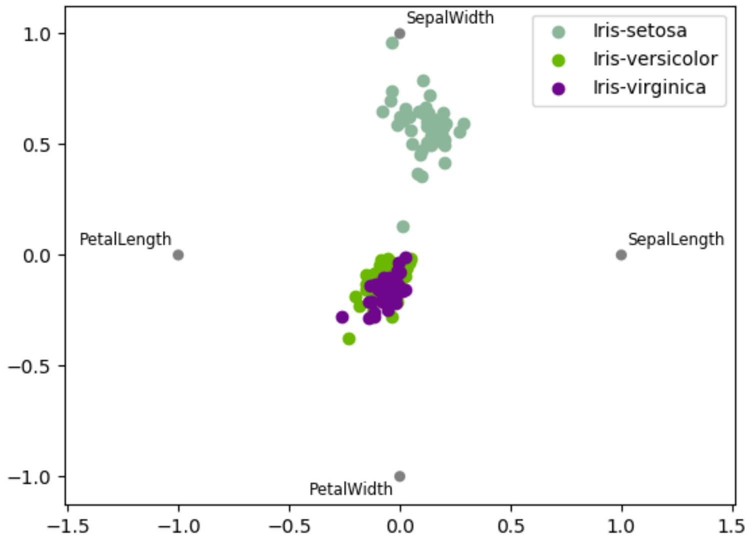
```
In [85]: from pandas.plotting import andrews_curves

In [86]: data = pd.read_csv('https://raw.githubusercontent.com/pandas-dev/pandas/master/pandas/tests/data/iris.csv ')
In [87]: plt.figure()
In [88]: andrews_curves(data, 'Name')
```



Iris Versicolor Iris Setosa Iris Virginica

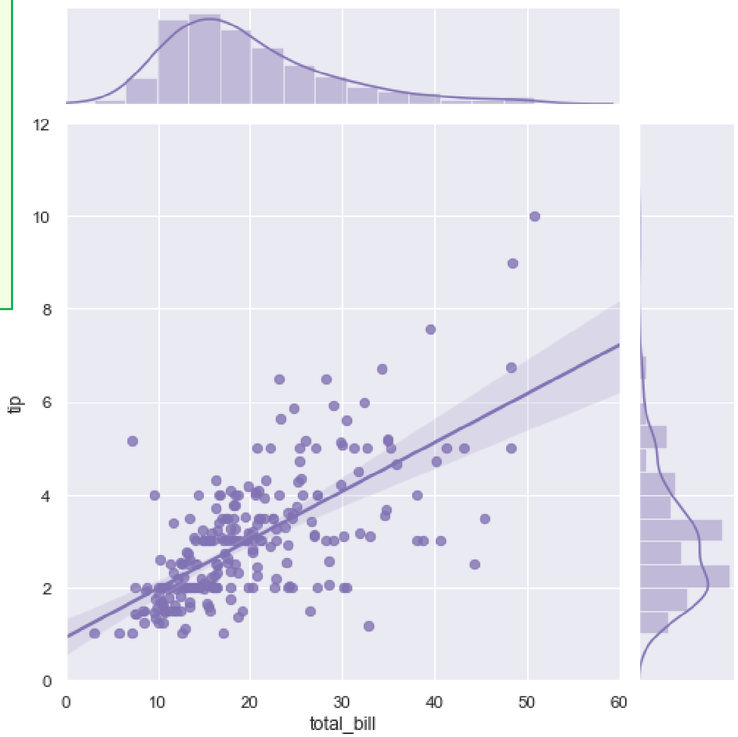
- Biblioteca Matplotlib para Visualización
  - RadViz: para datos multi-variable

```
In [104]: from pandas.plotting import radviz

In [105]: data = pd.read_csv('data/iris.data')
In [106]: plt.figure()
In [107]: radviz(data, 'Name')
```



- Otras bibliotecas para Visualización: Seaborn
  - Regresión lineal con distribuciones marginales

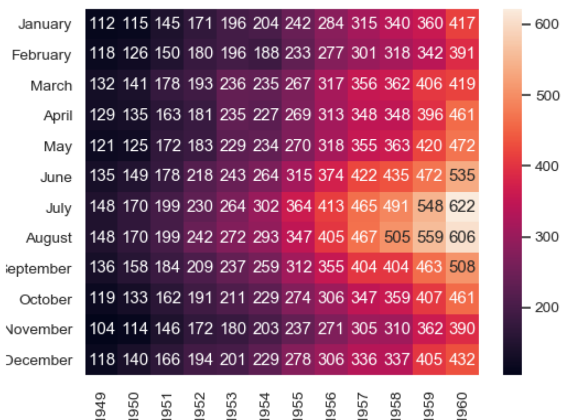
```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

sns.set(color_codes=True)

tips = sns.load_dataset("tips")

sns.jointplot(x="total_bill", y="tip", data=tips, kind="reg");
```



- Otras bibliotecas para Visualización: Seaborn
  - Mapas de calor: heatmap

```
import numpy as np
import seaborn as sns

flights = sns.load_dataset("flights")
flights = flights.pivot("month", "year", "passengers")
ax = sns.heatmap(flights)
```


```
ax = sns.heatmap(flights, annot=True, fmt="d")
```



- Otras bibliotecas para Visualización: Plotly
  - Plotly para Python https://plot.ly/python/
  - Ejemplo: burbujas dinámicas

https://plot.ly/python/gapminder-example/