Machine Learning

Preliminares

Tipos de datos

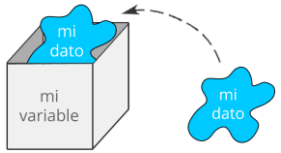
En un lenguaje de programación como Python, nos encontramos con diferentes tipos de datos:

Enteros	Floats	Strings	Booleanos
Es el conjunto de números Naturales	Es el conjunto de Números Reales	Es Texto, caracteres alfanuméricos. Se introducen entre comillas dobles, “”, o simples, ‘’.	Variables de “verdad”: verdadero o Falso
-2, -1, 0, 2, 4	-2.5, 0.7, 3.1	“Hola Mundo”	True ó False (1==1, 1==0)

También con diferentes estructuras de datos que nos permiten agrupar o hacer conjuntos de variables y objetos que también tienen sus propias funcionalidades y atributos, que no tenemos que programar.

Operaciones con variables

Una variable es un espacio de memoria donde guardamos un dato y le damos un nombre:




```
[6]: mi_variable = 'mi_dato'
```

Con diferentes tipos de datos, podemos hacer diferentes tipos de operaciones:

```
[1]: a = 'Hola '
     b = 'Mundo!'
     print(a + b)

Hola Mundo!
```

```
[2]: x = 3
     y = 12
     print(x + y)

15
```

```
[3]: mi_variable1 = True
     mi_variable2 = False
     print(mi_variable1 or mi_variable2)

True
```

No siempre se puede realizar operaciones entre variables que son de diferente tipo de dato:

```
[5]: a = 'Hola '
     z = 27
     print(a + z)

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-97bd203ea70a> in <module>
      1 a = 'Hola '
      2 z = 27
----> 3 print(a + z)

TypeError: can only concatenate str (not "int") to str
```

Operaciones con variables

```
[7]: x = 2
     y = 0.5
     print(x/y)
```

```
4.0
```

```
[8]: #Realiza la potencia x a la y
     x = 2
     y = 5
     print(x**y)
```

```
32
```

```
[9]: #Devuelve el resto de la division
     x = 15
     y = 4
     print(x%y)
```

```
3
```

Operación	Operador	Ejemplo
Suma	+	3 + 5.5 = 8.5
Resta	-	4 - 1 = 3
Multiplicación	*	3 * 6 = 18
Potencia	**	3 ** 2 = 9
División (cociente)	/	15.0 / 2.0 = 7.5
División (parte entera)	//	15.0 // 2.0 = 7
División (resto)	%	7 % 2 = 1

Operaciones con variables

Cuando tratamos con texto podemos concatenar o multiplicar:

[12]:
```
texto1 = 'el texto '
texto2 = 'se puede CONCATENAR!'
print(texto1 + texto2)
```

```
el texto se puede CONCATENAR!
```

[13]:
```
texto1 = ' el texto se puede MULTIPLICAR!'
print(texto1 * 3)
```

```
 el texto se puede MULTIPLICAR! el texto se puede MULTIPLICAR! el texto se puede MULTIPLICAR!
```

Operaciones con variables

Las operaciones lógicas se realizan con variables booleanas. Las operaciones lógicas verifican una condición que puede resultar cierta o falsa, de aquí surgen las tablas de verdad:

A && B		
A	B	A And B
0	0	0
0	1	0
1	0	0
1	1	1

A || B		
A	B	A Or B
0	0	0
0	1	1
1	0	1
1	1	1

Para el operador && se verifica A ***y*** B,
Mientras que para el operador || se verifica A ***o*** B.
También existe la negación:
not(False) == True

```
[14]: var1 = True
      var2 = False
      print(var1 and var2)

      False

[15]: print(var1 or var2)

      True

[16]: print(not(var2))

      True
```

Operaciones con variables

La Or exclusiva:

$$A \oplus B: A \cdot \overline{B} + \overline{A} \cdot B \equiv (A + B) \cdot (\overline{A} + \overline{B})$$

A Or Exclusiva B		
A	B	A Xor B
0	0	0
0	1	1
1	0	1
1	1	0

[21]:
```
var1 = False
var2 = False
print((var1 and not(var2)) or (not(var1) and var2))
```
```
False
```

[22]:
```
var1 = False
var2 = True
print((var1 and not(var2)) or (not(var1) and var2))
```
```
True
```

[23]:
```
var1 = True
var2 = False
print((var1 and not(var2)) or (not(var1) and var2))
```
```
True
```

[24]:
```
var1 = True
var2 = True
print((var1 and not(var2)) or (not(var1) and var2))
```
```
False
```

Listas

Una estructura de dato muy importante en Python es la lista, que consiste en una serie de elementos ordenados.

Esos elementos pueden ser de distinto tipo, e incluso pueden ser de tipo lista también:
Se pueden sumar entre sí, logrando concatenarlas, y también se les puede agregar elementos con el **método append()**

Un método es una función asociada a un objeto y sus atributos

```
[26]: lista_1 = [1,3.14,True,'Texto']
      type(lista_1)

[26]: list

[27]: print(lista_1)

      [1, 3.14, True, 'Texto']

[28]: lista_2 = [4,1.61,False, lista_1,'Otro texto']
      print(lista_2)

      [4, 1.61, False, [1, 3.14, True, 'Texto'], 'Otro texto']

[29]: lista_3 = ['Empieza lista3', 45]
      print(lista_1 + lista_3)

      [1, 3.14, True, 'Texto', 'Empieza lista3', 45]

[30]: lista_3.append('nuevo elemento')
      print(lista_3)

      ['Empieza lista3', 45, 'nuevo elemento']
```

Ciclos iterativos o loops

Son bloques de código que se repiten una cierta cantidad de veces en función de ciertas condiciones.

Un ciclo **for** repite un bloque de código tantas veces como elementos haya en una lista dada:

[31]:
```
lista_1 = [1,2,3,4]
for item in lista_1:
    print('Elemento: ',item)
```

```
Elemento:  1
Elemento:  2
Elemento:  3
Elemento:  4
```

Un ciclo **while** repite un código hasta que cierta condición se deje de cumplir:

[33]:
```
numero = 0
while(numero < 5):
    print(numero)
    numero = numero + 1
```

```
0
1
2
3
4
```

Condicionales

Los condicionales son bloques de código que se ejecutan únicamente si se cumple una condición. El resultado de esta condición debe ser booleano (True o False). Esto se logra mediante la sentencia **if.**

Con la sentencia **elif** se puede agregar un número arbitrario de condiciones. Por otra parte, se puede ejecutar código si la/s condición/es no se cumple/n con la sentencia **else**.

[37]:
```
#Solo ejecuta el codigo si se verifica la condicion
valor = -3
if (valor > 0):
    print('El valor es mayor que 0')
```

[39]:
```
valor = 1
if (valor > 0):
    print('El valor es mayor que 0')
```

```
El valor es mayor que 0
```

[40]:
```
valor = 0
if (valor < 0):
    print('Es un numero negativo')
elif (valor > 0):
    print('Es un numero positivo')
else:
    print('El numero es igual a 0')
```

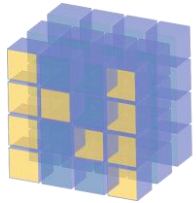
```
El numero es igual a 0
```

NumPy

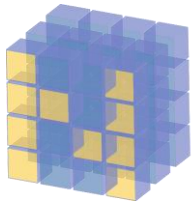
No siempre alcanzan las estructuras de datos y funcionalidades asociadas que vienen con Python, razón por la cual necesitaremos usar librerías.
Numpy es fundamental para hacer cálculo numérico y trae una estructura de datos propia: los **arrays** o **arreglos**.

Los arreglos son en principio como una lista, pero con muchas más funcionalidades. Por ejemplo, sumar un número al arreglo impactando en cada uno de sus elementos.

```
[41]: import numpy as np

[42]: arreglo = np.array([1,2,3,4])
      arreglo

[42]: array([1, 2, 3, 4])

[43]: print(arreglo)

      [1 2 3 4]

[44]: arreglo + 2

[44]: array([3, 4, 5, 6])
```



NumPy



Existen varias formas de crear arreglos en Numpy:

- A partir de una lista.
- Rango de valores de a “saltos” con el **método arange().**
- Rango de valores equidistantes con el **método linspace().**

```
[45]: arreglo1 = np.array([1,2,3,4,5])
      arreglo1

[45]: array([1, 2, 3, 4, 5])

[46]: arreglo2 = np.arange(0,10,2)
      arreglo2

[46]: array([0, 2, 4, 6, 8])

[48]: arreglo3 = np.linspace(0,10,20)
      arreglo3

[48]: array([ 0.        ,  0.52631579,  1.05263158,  1.57894737,  2.10526316,
              2.63157895,  3.15789474,  3.68421053,  4.21052632,  4.73684211,
              5.26315789,  5.78947368,  6.31578947,  6.84210526,  7.36842105,
              7.89473684,  8.42105263,  8.94736842,  9.47368421, 10.        ])
```

NumPy

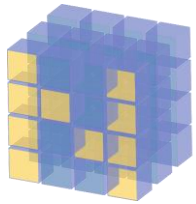
Seleccionando se puede asignar valores:

```
[60]: arreglo = np.arange(0,30)
      arreglo
```

```
[60]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
             17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29])
```

```
[61]: arreglo[0:30:3] = 100
      arreglo
```

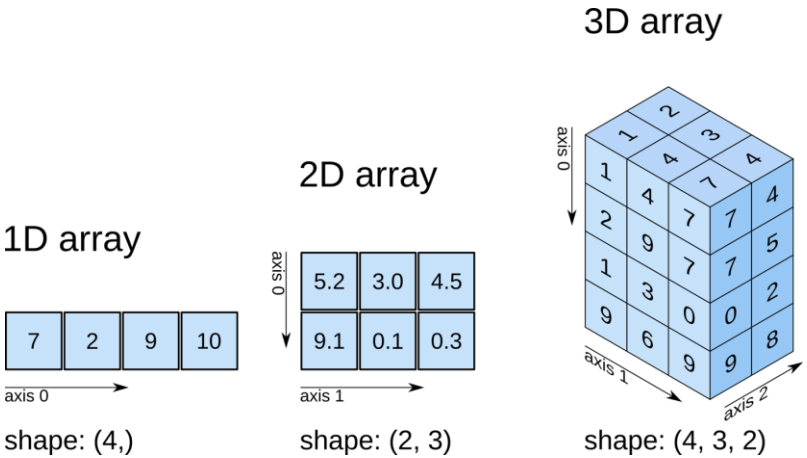
```
[61]: array([100,   1,   2, 100,   4,   5, 100,   7,   8, 100,  10,  11, 100,
              13,  14, 100,  16,  17, 100,  19,  20, 100,  22,  23, 100,  25,
              26, 100,  28,  29])
```

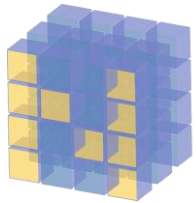


NumPy

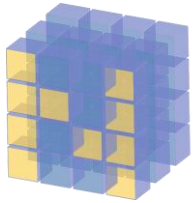
Los arreglos pueden ser de más de una dimensión, dando lugar a **arreglos multidimensionales**.

Nótese las propiedades axis y shape:





NumPy

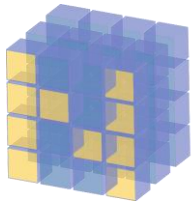


Una forma de manipulación muy común es el redimensionamiento con el **método reshape()**.
En un ejemplo, se convierte un arreglo de 30 elementos, en uno de 2x15 elementos, es decir, de una dimensión a dos dimensiones, y en el siguiente ejemplo, el mismo arreglo de 30 elementos, se convierte en uno de 2x3x5 elementos, pasando de una dimensión a tres dimensiones. Para hacer esta conversión sólo se necesita que la cantidad total de elementos sea la misma:

- ✔ 30 = 2x15 = 2x3x5

```
[65]: arreglo = np.arange(0,30)
      arreglo

[65]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
             17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29])

[66]: arreglo2d = arreglo.reshape(2,15)
      arreglo2d

[66]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14],
             [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]])

[67]: arreglo3d = arreglo.reshape(2,3,5)
      arreglo3d

[67]: array([[[ 0,  1,  2,  3,  4],
              [ 5,  6,  7,  8,  9],
              [10, 11, 12, 13, 14]],

             [[15, 16, 17, 18, 19],
              [20, 21, 22, 23, 24],
              [25, 26, 27, 28, 29]]])
```

NumPy



Se puede crear filtros que se denominan **máscaras** en base a una condición dada, para que posteriormente se puedan seleccionar los elementos que la cumplan.

De un arreglo de 2x15 elementos, tomaremos únicamente aquellos que sean par:

```
[68]: arreglo2d = np.arange(0,30).reshape(2,15)
      arreglo2d

[68]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14],
             [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]])

[70]: mascara = (arreglo2d % 2) == 0
      mascara

[70]: array([[ True, False,  True, False,  True, False,  True, False,  True,
              False,  True, False,  True, False,  True],
             [False,  True, False,  True, False,  True, False,  True, False,
               True, False,  True, False,  True, False]])

[71]: arreglo2d[mascara]

[71]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28])
```

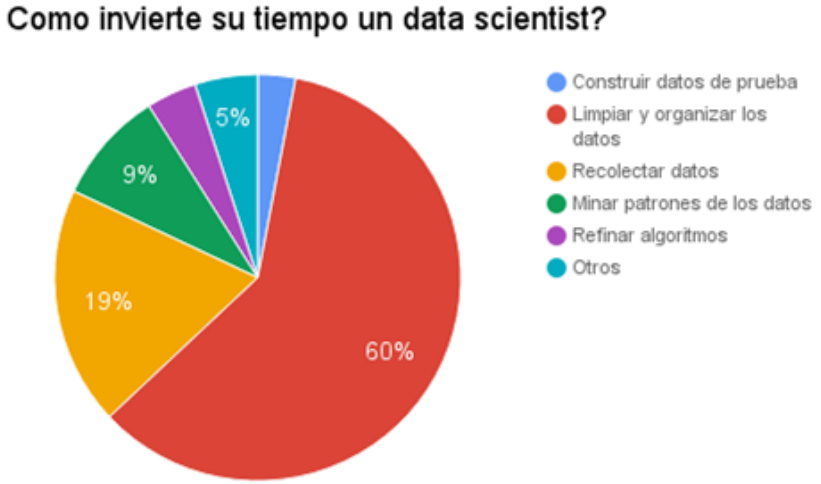
NumPy

Documentación recomendada

Fuente: *https://jakevdp.github.io/PythonDataScienceHandbook/*

Limpieza y preparación de los datos

En 2009 Mike Driscoll (científico de datos y CEO de Metamarkets) popularizo el termino «**data munging**» para referirse al arduo proceso de **limpiar, preparar y validar datos.**



Fuente: *http://www.forbes.com/sites/gilpress/2016/03/23/data-preparation-most-time-consuming-least-enjoyable-data-science-task-survey-says*

Limpieza y preparación de los datos

En 2013, Josh Wills (ex director de Data Science de Cloudera y actual Director of Data Engineering en Slack) comenta:

> *"**I'm a data janitor**. That's the sexiest job of the 21st century. It's very flattering, but it's also a little baffling."*

Traducción:
En Data Science, se invierte un 80% del tiempo en preparar los datos y el 20% restante en quejarse de la necesidad de preparar los datos.

Limpieza y preparación de los datos con Python

- Proceso reproducible.
- Control de versiones.
- Creación de tests automáticos.
- Facilita el mantenimiento.
- Lenguaje dinámico que permite alta productividad.
- Posibilidad de interfacear con C para performance.
- Preferido en ambientes de data science.

http://lemire.me/blog/2014/05/23/you-shouldnt-use-a-spreadsheet-for-important-work-i-mean-it/

Pandas

- **Colección de funciones y estructuras de datos** que facilitan el trabajo con datos estructurados.
- Construído **en base a Numpy** inicialmente por Wes McKinney.
- Nombre derivado de "**Pa**nel **Da**ta **S**ystem" (término econométrico para datasets multidimensionales).
- Brinda capacidades flexibles de manipulación de datos similares a spreadsheets y bases de datos relacionales.

Pandas

- Combina la alta performance de las **operaciones sobre arrays de NumPy con la flexibilidad en la manipulación de datos** de un spreadsheet o una base de datos relacional.
- Provee **funcionalidades de indexación avanzadas** para facilitar la manipulación, agregación y selección de partes de un dataset.
- Provee **operaciones de agrupación por columnas, filtros y sumarizaciones**.

Pandas:

Series

	Animales
0	Perro
1	Oso
2	Jirafa
3	Tigre
4	Serpiente
5	Ratón

Name

Index

Values

Pandas - Series

- Una Series es un objeto similar a un **vector uni-dimensional.**
- Contiene **un array de valores y un array asociado de etiquetas** de estos valores denominado como índice.
- Una colección Series también puede ser considerado como un **diccionario de tamaño fijo con sus claves ordenadas.**
- Al igual que los arrays de NumPy, permiten pasar una lista de valores con índices para seleccionar un subconjunto de valores.

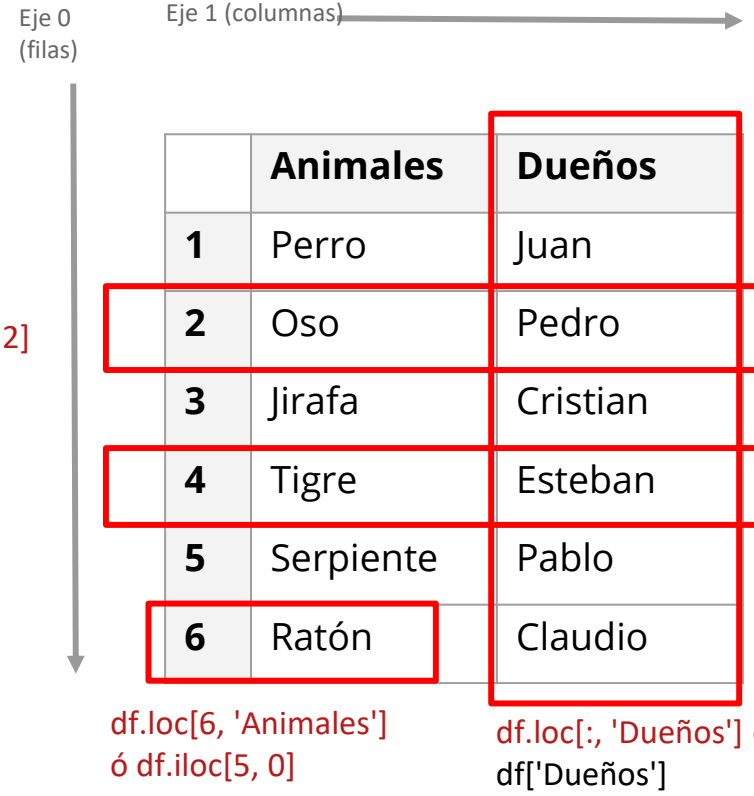
Pandas:
Data frames

- Para acceder a los objetos en Pandas es necesario introducir los siguientes métodos:
  - .loc
  - .iloc

df.iloc[1, 0:2]

df.loc[4, [‘Animales', ‘Dueños']]

- Notar que iloc refiere a la posición del valor en el dataframe, mientras que loc al valor del índice.



	Animales	Dueños
1	Perro	Juan
2	Oso	Pedro
3	Jirafa	Cristian
4	Tigre	Esteban
5	Serpiente	Pablo
6	Ratón	Claudio

Pandas:

Data frames

- Permite seleccionar los datos en un DataFrame en base condiciones lógicas. Las cuales se pueden obtener a partir de los valores.

[2]:
```
animales = {'Animales': ['Perro','Oso','Jirafa','Tigre', 'Serpiente', 'Raton'],
            'Dueños': ['Juan','Pedro','Cristian','Esteban', 'Pablo', 'Claudio']}
df = pd.DataFrame(data=animales, index=[1,2,3,4,5,6])
df
```

[2]:

	Animales	Dueños
1	Perro	Juan
2	Oso	Pedro
3	Jirafa	Cristian
4	Tigre	Esteban
5	Serpiente	Pablo
6	Raton	Claudio

[3]:
```
filtro = df.loc[:, 'Animales'] == 'Jirafa'
df.loc[filtro]
```

[3]:

	Animales	Dueños
3	Jirafa	Cristian

Pandas - Data frames

- Representa una **estructura de datos tabular** que contiene una **colección de columnas**, cada una de las cuales tiene un tipo determinado (number, string, boolean, etc.)
- Inspirados en la estructura data.frame de R.
- Permiten operaciones “ricas” sobre índices como los JOIN y GROUP BY en SQL.
- Ideales para organizar el resultado de un análisis en un formato útil para graficar el resultado o mostrarlo.

Valores faltantes

En general todo conjunto de datos suele tener datos faltantes, ya sea porque esos datos no fueron recolectados o nunca existieron:

- Debemos poder detectar, rellenar y eliminar datos faltantes.
- Hay que utilizar conocimiento del dominio para definir cuáles datos faltantes se completarán y cómo.
- Pandas ofrece varias formas de hacer esto.

Valores faltantes

Diferencia entre none y np.nan

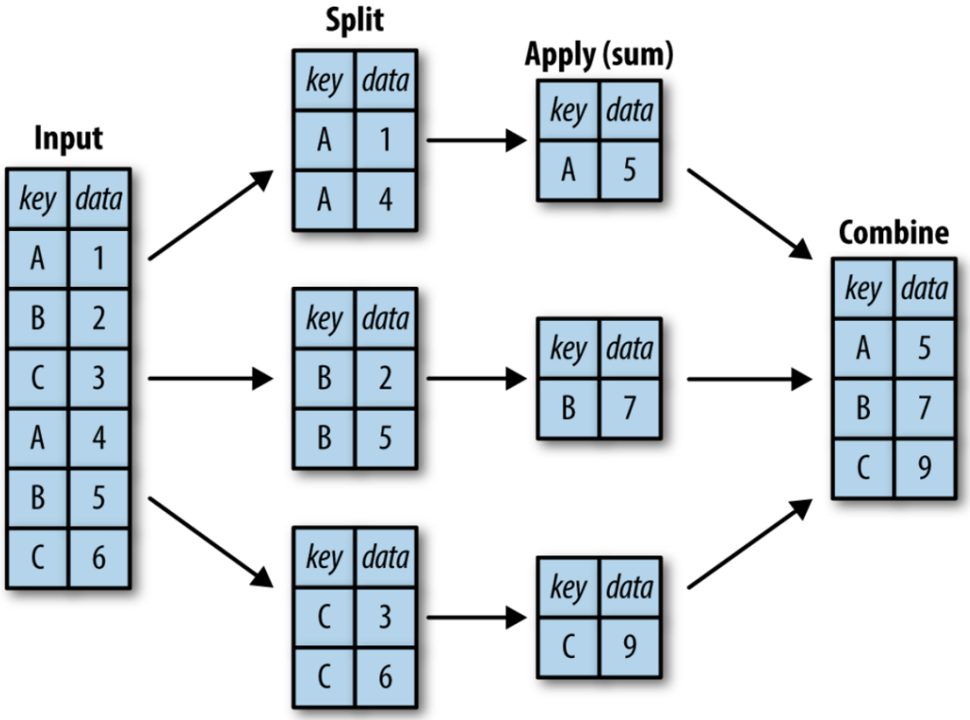
```
[29]: import numpy as np
      import pandas as pd

[30]: vals1 = np.array([1, None, 3, 4])

[31]: vals1

[31]: array([1, None, 3, 4], dtype=object)
```

- Cuando tenemos un objeto None incluído en una serie, el “upcasting” de Numpy se resuelve a “object”. Cuando tenemos un np.nan, conservamos una columna de tipo float y podemos seguir operando:

```
[31]: array([1, None, 3, 4], dtype=object)

[32]: vals2 = np.array([1, np.nan, 3, 4])

[34]: vals2.dtype

[34]: dtype('float64')
```

```
[35]: for dtype in ['object','int']:
          print("dtype=", dtype)
          %timeit np.arange(1E6, dtype=dtype).sum()
          print()

dtype= object
91.1 ms ± 9.1 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

dtype= int
3.34 ms ± 140 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Group by y Sumarización



Tablas pivot

- Una **tabla pivot** es una herramienta utilizada en sistemas de visualización de datos para sumarizar los mismos.
- Permite **ordenar, contar, sumar o aplicar otra función** de sumarización de los de una tabla inicial y genera una nueva con el resultado.
- Hay muchas formas de visualizar el resultado pero mayoritariamente los valores de las filas de una columna pasan a ser columnas por eso se denomina **pivot.**

- Los DataFrame de pandas poseen el método **pivot_table()** que genera otro DataFrame de salida con el resultado del pivot según los argumentos pasados al método.

Tablas pivot

Ejemplos

Fecha	Vendedor	Unidades	Monto
10/03/2017	Juan	10	500
10/03/2017	Pedro	4	200
11/03/2017	Juan	5	500
11/03/2017	Pedro	2	50

Pivot 1

Vendedor	Cantidad Total
Juan	15
Pedro	6

Pivot 2

Métrica	Juan	Pedro
Monto Total	1000	250

Tablas pivot

Ejemplos

ix	Item	CType	USD	EU
0	Item0	Gold	1	1
1	Item0	Bronze	2	2
2	Item0	Gold	3	3
3	Item1	Silver	4	4

ix=Item	Bronze	Gold	Silver
Item0	2	2 = mean(1,3)	NaN
Item1	NaN	NaN	4

```
d.pivot_table(index='Item', columns='CType', values='USD',  aggfunc=np.mean)
```

Matplotlib

No siempre alcanzan los indicadores numéricos para describir características de los datos. Por eso, recurrimos a la visualización, que no solo nos sirve para comunicar, una parte fundamental del trabajo, sino que también es una herramienta esencial para comprender el dataset con el que estamos trabajando.

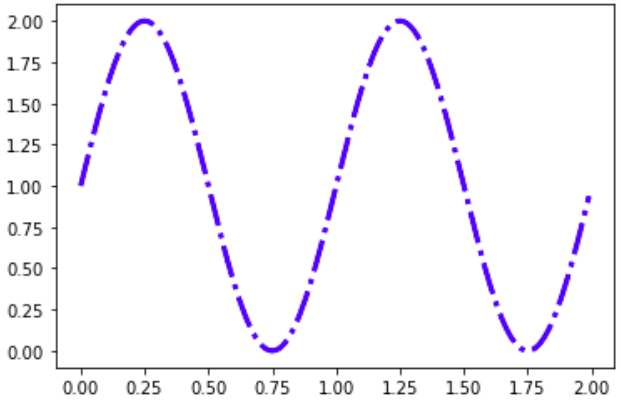
Matplotlib es una librería que utilizaremos para trabajar, con las funcionalidades que brinda podremos hacer mejores exploraciones del dato a nivel visual.

Matplotlib - Figuras y ejes

- Lo primero es crear la figura, que es el objeto correspondiente al recuadro sobre el cual se graficará, y luego, los distintos pares de ejes que se van a agregar dentro de una figura.
- Un gráfico sencillo puede ser **plotear** una línea. En el ejemplo, **x** define la coordenada horizontal, **y** define la vertical.
- Notar que se le puede pasar argumentos a la función plot(), algunos de ellos son color, linewidth y linestyle.

Documentación: *https://matplotlib.org/index.html*

```
[8]: import matplotlib.pyplot as plt
     %matplotlib inline

[9]: x = np.arange(0,2,0.01)
     y = 1 + np.sin(2 * np.pi * x)

[10]: fig = plt.figure()
      eje = plt.axes()
      eje.plot(x,y, color='blue', linewidth=3, linestyle='-.')

[10]: [<matplotlib.lines.Line2D at 0x21bdd78e148>]
```

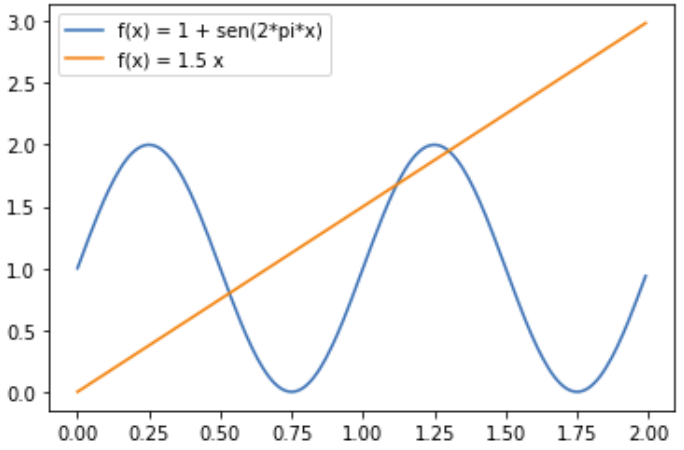


Matplotlib - Leyendas

Un gráfico tiene que ser principalmente claro y explícito. Para esto vamos a ayudarnos con las leyendas, las cuales nos permiten explicitar información sobre los distintos objetos que graficamos.

```
[16]: fig = plt.figure()
      eje = plt.axes()
      eje.plot(x,y, label='f(x) = 1 + sen(2*pi*x)')
      eje.plot(x,y2, label='f''(x) = 1.5 x')
      eje.legend()
```

```
[16]: <matplotlib.legend.Legend at 0x21be187f848>
```



Documentación: *https://matplotlib.org/index.html*

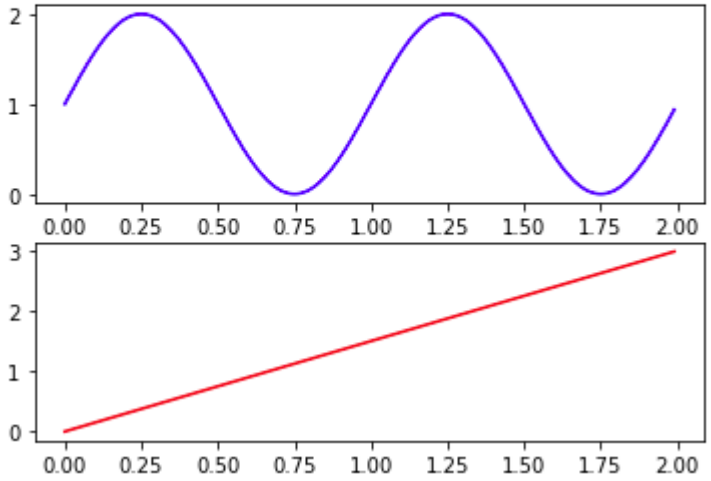
Matplotlib - Subplots

Por motivos de claridad del gráfico, eventualmente vamos a querer presentar en una misma figura varios ejes distintos. Esto lo podemos lograr con la función subplots.

En el ejemplo, definimos dos filas y una columna y luego graficamos cada eje por separado.

```
[21]: fig, ejes = plt.subplots(2, 1)
      ejes[0].plot(x, y, color='blue')
      ejes[1].plot(x, y2, color='red')
```

```
[21]: [<matplotlib.lines.Line2D at 0x21be2c7bfc8>]
```

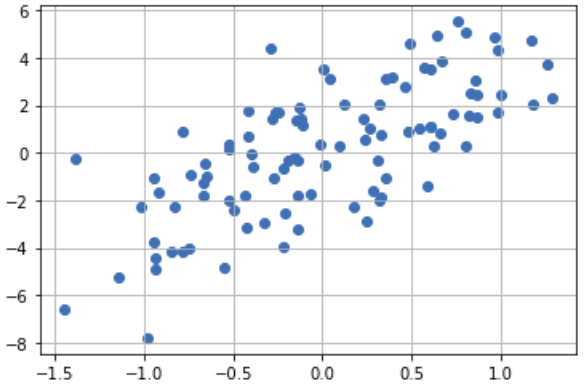


Documentación: *https://matplotlib.org/index.html*

Matplotlib - Scatter plot

Un Scatter Plot consiste en graficar una serie de puntos, con coordenadas (x,y) , sobre un plano.

[30]:
```
n = 100
x = np.linspace(-1,1,n) + 0.25*np.random.normal(size = n)
y = 3*x + 2*np.random.normal(size = n)

# Graficamos
plt.scatter(x, y)
plt.grid()
plt.show()
```



Documentación: *https://matplotlib.org/index.html*

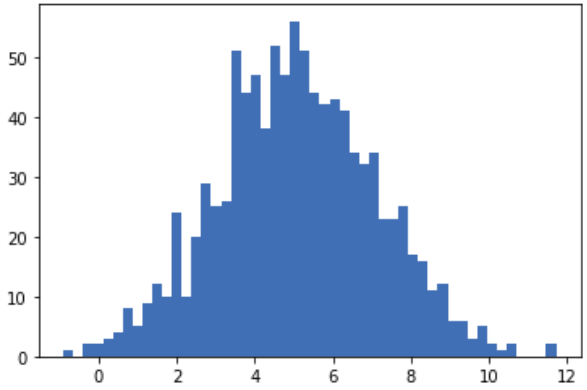
Matplotlib - Histogramas

En Matplotlib se cuenta con la función 'hist', que permite hacer histogramas. Recibe como argumento una lista o vector sobre el cual queremos hacer el histograma. Además de graficar, nos devuelve los vectores 'n' y 'bins', que son la cantidad de elementos que quedaron en cada bin y en qué valores sobre el eje x están ubicados los separadores de esos bins.

```
[36]: n

[36]: array([ 1.,  0.,  2.,  2.,  3.,  4.,  8.,  5.,  9., 12., 10., 24., 10.,
        20., 29., 25., 26., 51., 44., 47., 38., 52., 47., 56., 51., 44.,
        42., 43., 41., 34., 32., 34., 23., 23., 25., 17., 16., 11., 12.,
         6.,  6.,  3.,  5.,  2.,  1.,  2.,  0.,  0.,  0.,  2.])

[37]: bins

[37]: array([-0.89963006, -0.64732596, -0.39502187, -0.14271777,  0.10958633,
         0.36189043,  0.61419452,  0.86649862,  1.11880272,  1.37110681,
         1.62341091,  1.87571501,  2.12801911,  2.3803232 ,  2.6326273 ,
         2.8849314 ,  3.13723549,  3.38953959,  3.64184369,  3.89414778,
         4.14645188,  4.39875598,  4.65106008,  4.90336417,  5.15566827,
         5.40797237,  5.66027646,  5.91258056,  6.16488466,  6.41718876,
         6.66949285,  6.92179695,  7.17410105,  7.42640514,  7.67870924,
         7.93101334,  8.18331744,  8.43562153,  8.68792563,  8.94022973,
         9.19253382,  9.44483792,  9.69714202,  9.94944612, 10.20175021,
        10.45405431, 10.70635841, 10.9586625 , 11.2109666 , 11.4632707 ,
        11.71557479])
```

Documentación: *https://matplotlib.org/index.html*

```
[33]: # Elejimos una distribución con centro en 5
      # y una desviación igual a 2
      mu = 5  # Media de la distribución
      sigma = 2  # Desviación Estandar
      valores = mu + sigma * np.random.randn(1000)

[34]: # Definimos la cantidad de bins:
      n_bins = 50

[35]: # Creamos la figura y los ejes:
      fig, eje = plt.subplots()
      # Ploteamo el histograma:
      n, bins, _ = eje.hist(valores, n_bins)
```



Seaborn

- Seaborn corre sobre Matplotlib. Es por esto que trabaja con objetos definidos en esa librería, como por ejemplo figuras y ejes.
- La manera de utilizarlo es muy parecida (sabiendo usar Matplotlib resulta fácil usar Seaborn).
- La principal diferencia radica en la habilidad de Seaborn de poder importar los datos directamente desde un DataFrame.

Documentación: *https://seaborn.pydata.org/*

Seaborn

Ventajas:

- Facilita el trabajo con DataFrames.
- Mejora automáticamente la estética de los gráficos.
- Sintaxis sencilla para algunos gráficos complejos.

Desventajas:

- Precisa instalación de una librería adicional.
- Menos flexible que Matplotlib.

Documentación: *https://seaborn.pydata.org/*

Seaborn y Dataframes

Seaborn permite graficar tomando los datos directamente desde el dataframe. El conjunto de datos flor Iris o conjunto de datos iris de Fisher (estadístico y biólogo) es un conjunto de datos multivariante introducido en su artículo de 1936. Contiene 50 muestras de cada una de tres especies de Iris (Setosa, Virginica y Versicolor). Se midieron cuatro rasgos de cada muestra: el largo y ancho del sépalo y pétalo, en centímetros. Basado en la combinación de estos cuatro rasgos, Fisher desarrolló un modelo discriminante lineal para distinguir entre una especie y otra. ¡Aplicó ciencia de datos hace casi 100 años!

Documentación: *https://seaborn.pydata.org/*

Seaborn - Scatter plot

A la función scatterplot() se le pasa de parámetro, junto con el dataframe, qué variables del mismo tomará para cada eje (x=petal_width, y=petal_length). Asimismo, con el parámetro «hue» es posible separar las clases mediante el uso de alguna variable categórica (species). Por otra parte, con el parámetro «size» es posible diferenciar los datos según una variable numérica (sepal_length). Las leyendas se generan automáticamente explicitando las variables que no están en los ejes.

Documentación: *https://seaborn.pydata.org/*

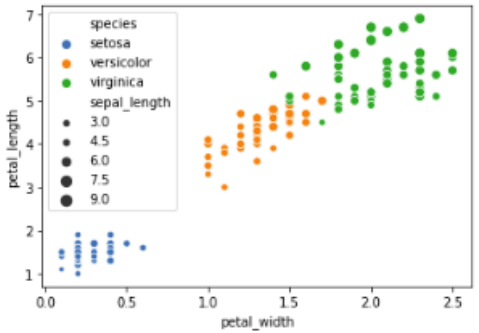
```
[39]: data = pd.read_csv("DataSets/iris_dataset.csv")
      data.head()
```

[39]:

	fila	sepal_length	sepal_width	petal_length	petal_width	species
0	fila1	5.1	3.5	1.4	0.2	setosa
1	fila2	4.9	3.0	1.4	0.2	setosa
2	fila3	4.7	3.2	1.3	0.2	setosa
3	fila4	4.6	3.1	1.5	0.2	setosa
4	fila5	5.0	3.6	1.4	0.2	setosa

```
[45]: sns.scatterplot(x='petal_width', y='petal_length', size='sepal_length', hue='species', data=data)
```

```
[45]: <matplotlib.axes._subplots.AxesSubplot at 0x21be3d13448>
```

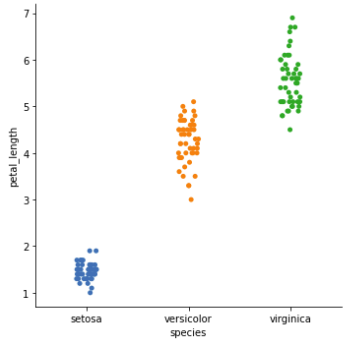


Seaborn - Categorical plot

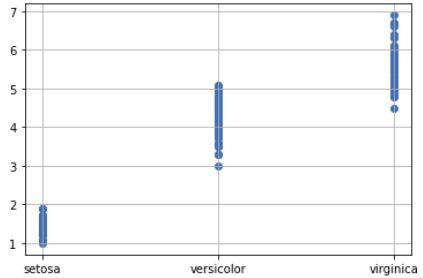
- La función catplot() permite graficar utilizando una variable categórica como eje.
- Seaborn es destacable sobre Matplotlib en este tipo de gráficos.
- Nótese que no “amontona” los puntos, permitiendo distinguir la cantidad.

```
sns.catplot(x='species', y='petal_length', data=data)
```

```
<seaborn.axisgrid.FacetGrid at 0x21be3eee908>
```




```
x = data.species
y = data.petal_length
plt.scatter(x, y)
plt.grid()
plt.show()
```

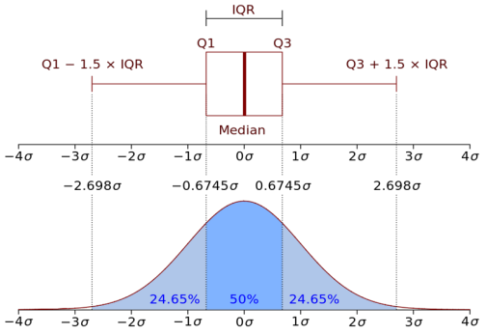


Documentación: *https://seaborn.pydata.org/*

Seaborn - Boxplot

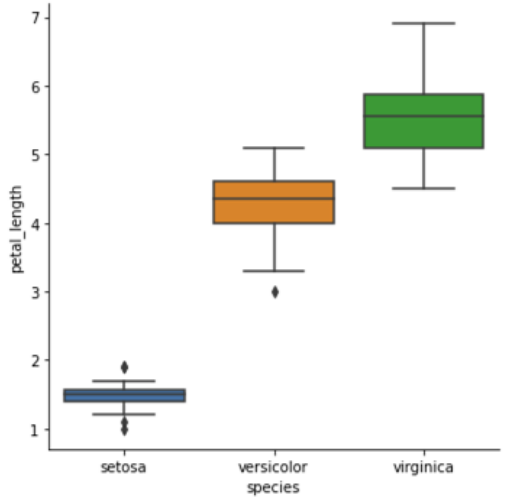
El diagrama de cajas es una forma de visualizar un conjunto de valores. Muchas veces resulta más informativa que simplemente dibujar un punto por cada valor, ya que nos permite tener una idea de como es la distribución subyacente.

Notar que esta vez, la función usada sigue siendo catplot() pero el parámetro «kind» toma el valor 'box'.




```
sns.catplot(x='species', y='petal_length', kind='box', data=data)
```

```
<seaborn.axisgrid.FacetGrid at 0x21be3f2ad88>
```



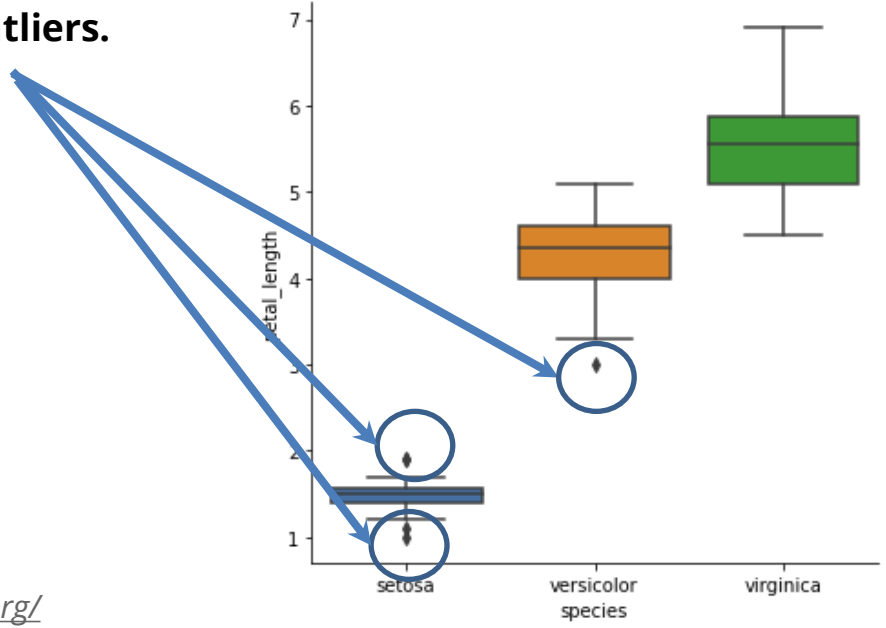
Documentación: *https://seaborn.pydata.org/*

Seaborn - Boxplot

Los puntos que están fuera de las extensiones de la caja pueden considerarse **Outliers.**

```
sns.catplot(x='species', y='petal_length', kind='box', data=data)
```

```
<seaborn.axisgrid.FacetGrid at 0x21be3f2ad88>
```



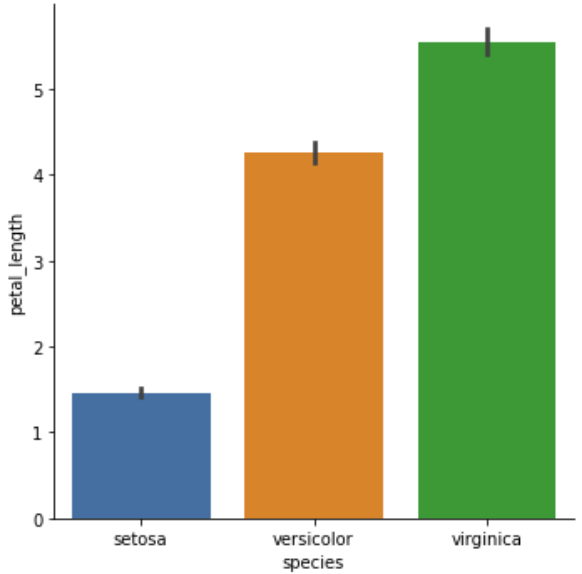
Documentación: *https://seaborn.pydata.org/*

Seaborn - Barplot

El gráfico de barras puede ser útil en ciertas circunstancias en que tomamos una única instancia (petal_length).

```
sns.catplot(x='species', y='petal_length', kind='bar', data=data)
```

```
<seaborn.axisgrid.FacetGrid at 0x21be3ed6588>
```



Documentación: *https://seaborn.pydata.org/*

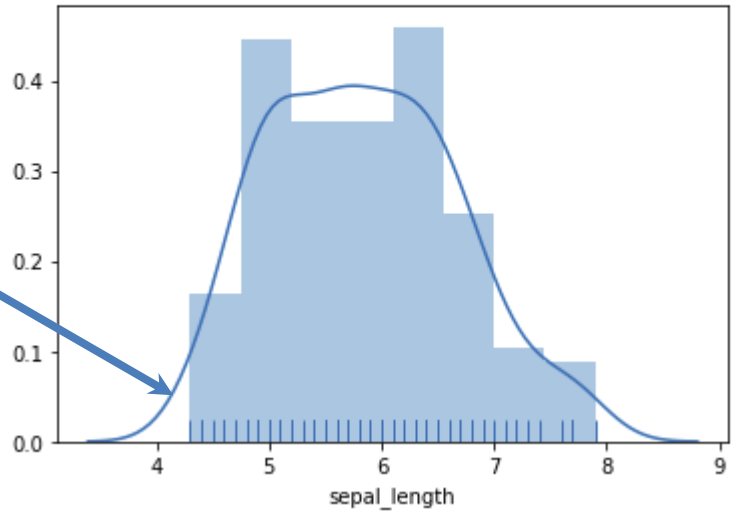
Seaborn - Histogramas

Por defecto, la función distplot(), además del histograma, grafica una curva que intenta aproximar la distribución de la que vienen los datos. Su nombre es KDE (kernel density estimation).

El parámetro rug=True muestra las instancias sobre el eje X.

```
sns.distplot(data['sepal_length'], rug=True)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x21be5f1c708>
```



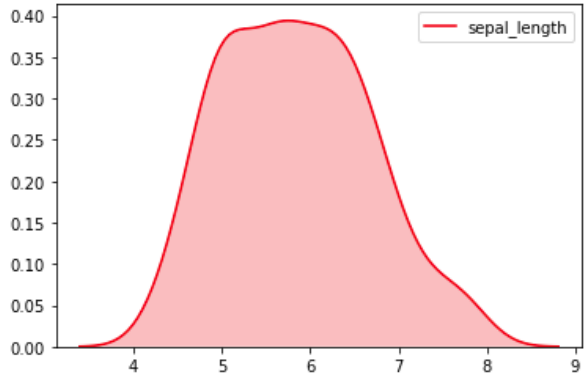
Documentación: *https://seaborn.pydata.org/*

Seaborn - KDE

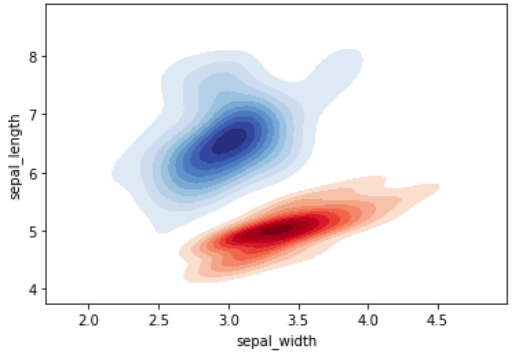
Con kdeplot() es posible representar la distribución de densidad de probabilidad del dato, lo que se puede hacer de manera univariante ó bivariante:

```
[55]: sns.kdeplot(data.sepal_length, shade=True, color="r")
```

```
[55]: <matplotlib.axes._subplots.AxesSubplot at 0x21be51e6488>
```




```
[57]: setosa = data.loc[data.species == "setosa"]
virginica = data.loc[data.species == "virginica"]
ax = sns.kdeplot(setosa.sepal_width, setosa.sepal_length,
                 cmap="Reds", shade=True, shade_lowest=False)
ax = sns.kdeplot(virginica.sepal_width, virginica.sepal_length,
                 cmap="Blues", shade=True, shade_lowest=False)
```



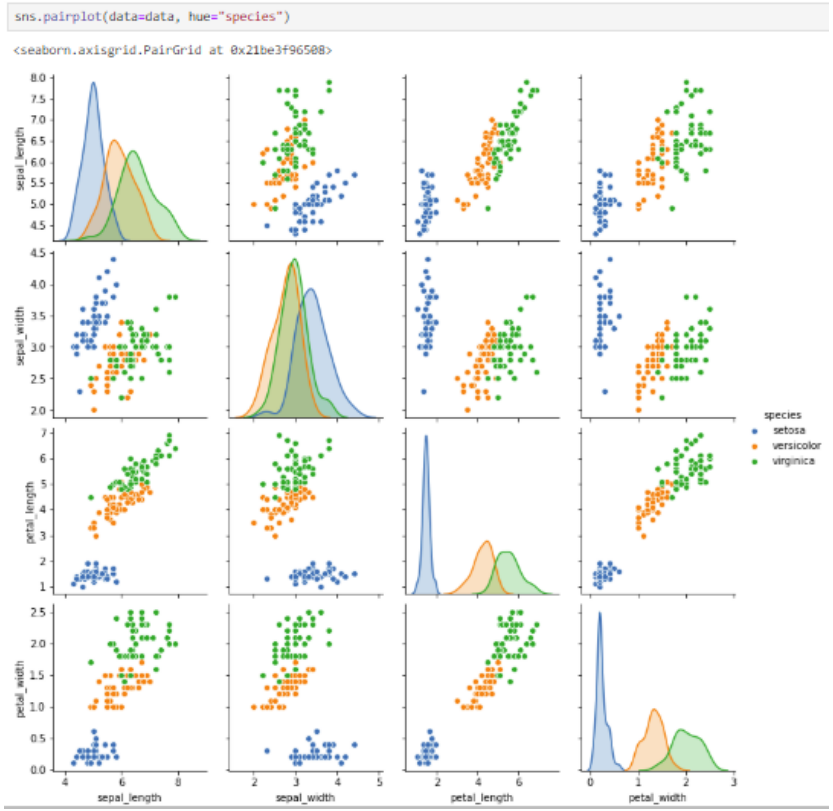
Documentación: *https://seaborn.pydata.org/*

Seaborn - Pair plots

La función pairplot() de Seaborn resulta muy útil a la hora de explorar los **features** o **variables** de un dataset debido a que compara todas las variables numéricas del dataset entre sí.

Al mismo tiempo, nos permite colorear según una variable categórica. En la horizontal vemos para cada variable numérica, su distribución respecto de la variable categórica. Y en el resto de intersecciones de la matriz, el gráfico scatter que permite ver la correlación.

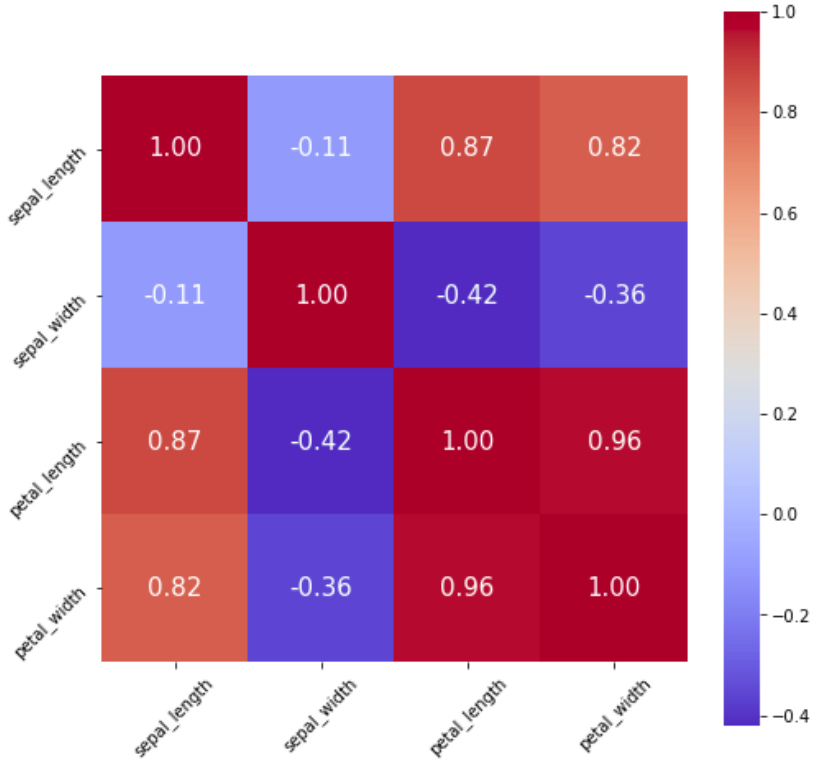
Documentación: *https://seaborn.pydata.org/*



Seaborn - Heatmaps

Si queremos una visión aún más explícita de la correlación entre variables, la función heatmap() representa el índice de Pearson mediante colores logrando ese objetivo.

```
[61]: #corr sera la matriz de correlaciones
corr = data.drop(columns = ['species','fila']).corr()
plt.figure(figsize=(8,8))
sns.heatmap(corr, cbar = True,  square = True, annot=True, fmt= '.2f',annot_kws={'size': 15},
            xticklabels= data.drop(columns = ['species','fila']).columns,
            yticklabels= data.drop(columns = ['species','fila']).columns,
            cmap= 'coolwarm')
plt.xticks(rotation = 45)
plt.yticks(rotation = 45)
plt.show()
```



Documentación: *https://seaborn.pydata.org/*

¡Muchas gracias!