# PREPROCESAMIENTO DE DATOS CON PYTHON

Abraham Zamudio

VI Programa de Especialización en Machine Learning con Python

2020


CTIC-UNI
BUSINESS SCHOOL

Numpy

Matplotlib

Scipy

Pandas

# NumPy

# *Una introducción a Numpy*

`NumPy` (*Numerical Python*) es una biblioteca/librería de Python de código abierto que se utiliza en casi todos los campos de la ciencia y la ingeniería. Es el estándar universal para trabajar con datos numéricos en Python, y está en el núcleo de los ecosistemas científicos de Python y PyData. Los usuarios de NumPy incluyen a todos, desde codificadores principiantes hasta investigadores experimentados que realizan investigación y desarrollo científico e industrial de vanguardia. La API NumPy se usa ampliamente en Pandas, SciPy, Matplotlib, scikit-learn, scikit-image y la mayoría de los demás paquetes de Python.

# *¿Qué es NumPy?*

- Es una biblioteca de Python para trabajar con arreglos multidimensionales.
- El principal tipo de dato es el arreglo o *array*
- También nos permite trabajar con la semántica de matrices
- Nos ofrece muchas funciones útiles para el procesamiento de números

# *Numpy : Formas de cargar el modulo*

Puede importar una función particular del módulo como se muestra a continuación y trabajar con ella como cualquier otra función.

```
1 from numpy import arange
2 a = arange(15)
```

Puede importar todo el módulo con un nombre corto como se muestra a continuación. Esto le permite trabajar con todas las funciones presentes en el módulo.

```
1 import numpy as np
2 a = np.arange(15)
3 b = np.array([1, 5, 4, 3])
```

# *¿Cuál es la diferencia entre una lista de Python y una matriz NumPy?*

NumPy le ofrece una enorme variedad de opciones rápidas y eficientes relacionadas con los números. Si bien una lista de Python puede contener diferentes tipos de datos dentro de una sola lista, todos los elementos en una matriz NumPy deben ser homogéneos. **Las operaciones matemáticas que deben realizarse en matrices no serían posibles si las matrices no fueran homogéneas.**

Las matrices NumPy son más rápidas y compactas que las listas de Python. Una matriz consume menos memoria y es mucho más conveniente de usar. NumPy utiliza mucha menos memoria para almacenar datos y proporciona un mecanismo para especificar los tipos de datos, lo que permite optimizar aún más el código.

# *Numpy : Documentación de referencia*

```
1 np.array?
2
3 help(np.array)
4
5 np.lookfor("create array")
6
7 np.con*?
```

# *El Array*

- Es una tabla de elementos
  - ☑ normalmente números
  - ☑ todos del mismo tipo
  - ☑ indexados por enteros
- Ejemplo de arreglos multidimensionales
  - ☑ Vectores
  - ☑ Matrices
  - ☑ Imágenes
  - ☑ Planillas
- ¿Multidimensionales?
  - ☑ Que tiene muchas dimensiones o ejes
  - ☑ Un poco ambiguo, mejor usar *ejes*
  - ☑ Rango: cantidad de ejes

# *Propiedades del Array*

- Como tipo de dato se llama `ndarray`
- `ndarray.ndim`: cantidad de ejes
- `ndarray.shape`: una tupla indicando el tamaño del array en cada eje
- `ndarray.size`: la cantidad de elementos en el array
- `ndarray.dtype`: el tipo de elemento que el array contiene
- `ndarray.itemsize`: el tamaño de cada elemento en el array

# *Propiedades del Array*

```
1  from  numpy import *
2
3  a = arange(10).reshape(2,5)
4
5  a.shape
6
7  a.ndim
8
9  a.size
10
11 a.dtype
12
13 a.itemsize
```

# *Creando Arrays*

➔ Con funciones específicas en función del contenido

```
np.arange(5)
np.zeros((2, 3))
np.ones((3, 2), dtype=int)
np.empty((2, 2))
np.linspace(-np.pi, np.pi, 5)
np.array([-3.141592,-1.570796,1.570796,3.141592])
np.random.rand(4)
np.random.randn(4)
np.random.seed(1234)
```

# *Operaciones básicas*

➔ Los operadores aritméticos se aplican por elemento

```
1 a = arange(20, 60, 10)
2 a
3 a + 1
4 a * 2
```

➔ Si es *inplace*, no se crea otro array

```
1 a
2 a /= 2
3 a
```

# *Operaciones básicas*

➔ Podemos realizar comparaciones para construir arrays con elementos booleanos.

```
a = np.arange(5)
a >= 3
a % 2 == 0

a = np.reshape(np.arange(16), (4,4))
large_values = (a > 10)
even_values = (a%2 == 0)
```

# *Operaciones básicas*

➔ Indexado mediante arrays de tipo boolean : Las matrices booleanas se pueden usar para seleccionar elementos de otras matrices numpy. Si a es una matriz numpy y b es una matriz booleana de las mismas dimensiones, entonces a[b] selecciona todos los elementos de a para los cuales el valor correspondiente de b es `True`.

```
1 a = np.reshape(np.arange(16), (4,4))
2 b = (a%2 == 0)
3 print(a[b])
```

Podemos usar esto para modificar elementos de una matriz que satisfagan una condición lógica:

```
1 a[a%2 == 0] = 100
```

# *Operaciones básicas*

➔ Tenemos algunos métodos con cálculos típicos

```
1 c = np.random.random(8)
2 c.min(), c.max()
3 c.mean()
4 c.sum()
5 c.cumsum()
```

https://numpy.org/doc/1.17/reference/routines.html

# *Trabajando con los elementos*

➔ La misma sintaxis de Python

```
1 a = arange(10)
2 a
3 a[2]
4 a[2:5]
5 a[1] = 88
6 a[-5:] = 100
7 a
```

# *Trabajando con los elementos*

➔ Pero también podemos trabajar por eje

```
1 a = arange(8).reshape((2,4))
2 a
3 a[:,1]
4 a[0,-2:]
```

# *Cambiando la forma del array*

➔ Transponer y aplanar (ravel)

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

# *Juntando y separando arrays*

➔ Tenemos vstack y hstack

```
1  a = ones((2,5)); b = arange(5)
2  juntos = vstack((a,b))
3
4  a =ones((20,100,3))
5  b = vstack((a,a))
6  print(b.shape)
7
8  P = np.random.normal(size=(20,100,3))
9  Q = np.random.normal(size=(20,100,3))
10
11 # Otra forma es con la funcion concatenate
12 # Midamos los tiempos
13 # concatenate  vs.  vstack
14 %timeit b = np.concatenate((P,Q),axis=0)
15 %timeit b = np.vstack((P,Q))
```

# *Juntando y separando arrays*

- ➔ `numpy.hsplit(array,escalarseq)|` : Divide array en subarrays por columnas. Misma forma de subarray si se da escalar. Corta por las columnas dadas en seq.
- ➔ `numpy.vsplit(array,escalarseq)|` :Divide en subarrays por filas.

```
1 from numpy import *
2 a =  array ([[1 , 2, 3] , [4 , 5, 6] , [7 , 8, 9]])
3
4 # Esto es un ERROR
5 print(vsplit (a , 2))
6
7 print (vsplit (a , (1 ,2)))
8
9 print (hsplit (a , (1 ,)))
```

# *Indexado avanzado*

➔ Podemos indizar con otros arrays

```
1 a = arange(10) ** 2
2 i = array([ (2,3), (6,7) ])
3 a[i]
```

➔ O elegir elementos

```
1 a = arange(5)
2 b = a % 2 == 0
3
4 a[b]
```

# *Matrices*

➔ Es un caso específico del array . Si queremos realizar la multiplicación de matrices con dos matrices numpy (ndarray), tenemos que usar el producto punto.

```
1 x = np.array( ((2,3), (3, 5)) )
2 y = np.matrix( ((1,2), (5, -1)) )
3 np.dot(x,y)
4
5 # Alternativamente, podemos convertirlos en objetos de
  ↪  matriz
6 # y usar el operador "*"
7 np.mat(x) * np.mat(y)
```

# *Valores especiales*

Además de los objetos con tipo **dtype**, **NumPy** introduce valores numéricos especiales: **nan** y **inf**. Estos pueden surgir en los cálculos matemáticos. **Not A Number** (**nan**). Indica que un valor que debe ser numérico no está, de hecho, definido matemáticamente. Por ejemplo, 0/0 produce **nan**. A veces, **nan** también se usa para indicar información faltante; por ejemplo, **pandas** usa esto. **inf** indica una cantidad que es arbitrariamente grande, por lo que en la práctica significa más grande que cualquier número que la computadora pueda concebir. **-inf** también está definido y significa arbitrariamente pequeño. Esto podría ocurrir si una operación numérica explota, es decir, crece rápidamente sin límites.

```
1 vec1 = np.array([1,-1,0],dtype=np.float16)
2 vec2 = vec1 / 0
3
4 array([ inf, -inf,  nan], dtype=float16)
```

# *Valores especiales*

Ejecutemos un bucle que es incorrecto:

```
for i in vec2:
    print(i)
    print("---------")
    print("Inf " + str(i==np.inf))
    print("-Inf " + str(i==-np.inf))
    print("Nan " + str(i==np.nan))
    print("/n/n")
```

Vamos a recorrer cada valor posible de vec2 e imprimir los resultados de `i == inf`, `i == -inf`, y si i es igual a `nan`, `i== nan`. Lo que obtenemos es una lista; los dos primeros bloques de inf y -inf están bien, pero **nan** no está bien.

# *Valores especiales*

Queríamos que detectara un **nan** pero no lo hizo. Entonces, intentémoslo usando la función isnan:

```
1 for i in vec2:
2     print(i)
3     print("----------")
4     print("Inf " + str(i==np.inf))
5     print("-Inf " + str(i==-np.inf))
6     print("Nan " + str(np.isnan(i)))
7     print("/n/n")
```

# *Valores especiales*

Ahora detectemos números finitos y números infinitos

```
1 for i in vec2:
2     print(i)
3     print("---------")
4     print("Es finito ? " + str(np.isfinite(i)))
5     print("Es infinito ? " + str(np.isinf(i)))
6     print("\n\n")
```

Como es de esperar `inf` no es finito, pero `nan` no cuenta como ni finito ni infinito; es indefinido.Probemos que sucede con `inf +1`,`inf-1`, `nan +1`, `2**ver[0]` , `2**vec2[1]` y `inf-inf`.

# *Valores especiales*

Ahora, vamos a crear un vector y colocarle como único elemento el un número 999. Si tuviéramos que elevar vector a sí misma, en otras palabras, 999 a la potencia de 999 (sabemos que $999^{999}$ es un numero finito), nos enseñara que no podemos confiar **por completo** en los cálculos que realiza una computadora.

```
1  vec3 = np.array([999])
2  vec3.dtype
3  vec3[0]**vec3[0]
4
5
6  vec3 = np.array([999],dtype=np.float64)
7  vec3[0]**vec3[0]
8
9
10 vec3 = np.array([999],dtype=np.float128)
11 vec3[0]**vec3[0]
```

# *Valores especiales*

Ahora, vamos a crear una vector y asignarle al primer elemento de este el valor de **nan**. Si sumamos los elementos de este vector, lo que obtenemos es **nan** porque **nan** + cualquier cosa es **nan**.

```
1 vec4 = np.ones(5)
2 vec4[4] = np.nan
3
4 sum(vec4)
5
6 # sumamos todos los elementos NO nan
7 np.nansum(vec4)
```

# *Numpy where*

El objetivo de reemplazar este tipo de bucle **for** usado junto con un **if else**

```
a = (np.random.rand(1,10))
result = np.zeros(10)
result = (a.reshape(1,10))
print ("a: \n" + str(a))
for i in range(10):
    if a[0,i]>0.5:
        result[0,i] = 1
    else:
        result[0,i] = 0
print ("result.: \n" +str(result))
```

# *Numpy where*

Reemplazamos el codigo anterior con esto :

```
a = (np.random.rand(1,10))
print ("a: \n" + str(a))
result = np.where(a>0.5,1,0)
print ("result: \n ", result)
```

Lo que **np.where** hace es que primero crea una matriz del mismo tamaño que el primer argumento. El primer argumento es la matriz (o vector) por la que pasamos y revisamos cada entrada si el valor es mayor a 0.5 (condicional), el segundo argumento es el valor que se reemplaza en la nueva matriz (o vector) si la condición es verdadera, y el tercer argumento es el valor que es reemplazado en la nueva matriz (0 vector) si la condición es falsa.

# *Numpy where*

Veamos otro ejemplo.

```
# Creamos un array (2,2) de valores booleanos
array =  [[True, False], [True, True]]

# matriz de dónde escoger los valores si la
# matriz en la posición es verdadera
array_cond_true = [[1, 2], [3, 4]]

# matriz de dónde escoger valores si la matriz
# en la posición es falsa
array_cond_false = [[9, 8], [7, 6]]

print ("result: \n" +
↪  str(np.where(array,array_cond_true,array_cond_false)))
```

# *Numpy where*

Veamos un poco la eficiencia de usar **where** :

```
import time
a = (np.random.rand(1,1000000))
result1 = np.zeros((1,1000000),dtype=np.int)
tic = time.time()
for i in range(1000000):
    if a[0,i]>0.5:
        result1[0,i] = 1
    else:
        result1[0,i] = 0
toc = time.time()
print ("time passed for result1: " + str(toc-tic) + "ms")

tic = time.time()
result2 = (np.where(a>0.5,1,0))
toc = time.time()
print ("time passed for result2: " + str(toc-tic) + "ms")
```

# *Numpy Vectorize*

Es posible vectorizar cualquier función que definamos con la función **vectorize**. Obsérvese el siguiente ejemplo: si tenemos nuestra propia función

```
1 import math
2 def myfunc(a):
3 return math.sin(a)
```

entonces, podemos usarla para trabajar sobre arrays del siguiente modo:

```
1 vfunc = np.vectorize(myfunc)
2 a = np.arange(4)
3 print vfunc(a)
```

# *Numpy vs Python puro*

```
1  import numpy as np
2  from math import log10 as lg10
3  import time
4  import matplotlib.pyplot as plt
5  import random
6
7  # Num. de elementos a procesar
8  N = 1000000
9
10 # Una lista para almacenar los tiempos
11 speed = []
12
13 l1 = list(100*(np.random.random(N))+1)
14 print("Length of l1:",len(l1))
15
16 print("Primeros elementos de la lista :", l1[:4])
17
18 a1 = np.array(l1)
19 print("Shape:",a1.shape)
20 print("Type:",type(a1))
21
22
23 # Almacenamiento de la operacion : log10
24 l2=[]
```

# *Numpy vs Python puro*

```
# Usando un bucle : FOR
t1=time.time()
for item in l1:
    l2.append(lg10(item))
t2 = time.time()
print("Tiempo usando buble for {} segundos ".format(t2-t1))
speed.append(t2-t1)
print("Primeros elementos del array resultante:", l2[:4])

# Usando comprehension de listas
t1=time.time()
l2 = [lg10(i) for i in range(1,1000001)]
t2 = time.time()
print("Tiempo usado con list comprehension, tomo {}
↪  segundos".format(t2-t1))
speed.append(t2-t1)
print("Primeros elementos del array resultante:", l2[:4])
```

# *Numpy vs Python puro*

```
# Usando un funcion
def op1(x):
    return (lg10(x))
t1=time.time()
l2=list(map(op1,l1))
t2 = time.time()
print("Con una funcion demoro {} segundos ".format(t2-t1))
speed.append(t2-t1)
print("Primeros elementos del array resultante:", l2[:4])


# Usando Numpy : vector vectorizado
t1=time.time()
a2=np.log10(a1)
t2 = time.time()
print("Con la funcion log10 de numpy demoro {} segundos
↪  ".format(t2-t1))
speed.append(t2-t1)
l3 = list(a2)
print("Primeros elementos del array resultante:", l3[:4])
```

# *Numpy vs Python puro*

```
speed

[0.22377943992614746,
0.13420581817626953,
0.1909801959991455,
0.030059814453125]
```

Por lo tanto, vemos la evidencia de que las operaciones NumPy sobre los objetos ndarray son mucho más rápidas que las operaciones matemáticas de Python sobre la lista correspondiente. La velocidad exacta de las operaciones regulares de Python varía un poco, pero siempre son mucho más lentas en comparación con la operación NumPy vectorizada.

▸ Otro ejemplo.

# *Benchmarking sobre numpy*

```
1 import numpy as np
2 np.__config__.show()
3 np.__version__
```

Algunos enlaces de interes :

- NumPy User Guide

- Routines

# *Benchmarking sobre numpy : Multiplicación de matrices*

```python
import numpy as np
from time import time

# Tomemos la aleatoriedad de números aleatorios
# (para reproducibilidad)
np.random.seed(0)

size = 4096
A, B = np.random.random((size, size)), np.random.random((size,
↪  size))

N = 20
t = time()
for i in range(N):
   np.dot(A, B)
delta = time() - t
print('Producto Matricial de dos matrices de dimension %dx%d \n
↪  Tiempo %0.2f s.' % (size, size, delta / N))
del A, B
```

# *Benchmarking sobre numpy : Multiplicación de vectores*

```python
import numpy as np
from time import time

# Tomemos la aleatoriedad de números aleatorios (para reproducibilidad)
np.random.seed(0)

size = 4096
C, D = np.random.random((size * 128,)), np.random.random((size * 128,))

N = 5000
t = time()
for i in range(N):
    np.dot(C, D)
delta = time() - t
print('Producto interno de  dos vectores  de dimension %d \n Tiempo : %0.2f ms.' % (size * 128, 1e3 * delta / N))
del C, D
```

# *Benchmarking sobre numpy : Descomposición de valores singulares (SVD)*

```python
import numpy as np
from time import time

# Tomemos la aleatoriedad de números aleatorios (para reproducibilidad)
np.random.seed(0)

size = 4096

E = np.random.random((int(size / 2), int(size / 4)))

N = 3
t = time()
for i in range(N):
   np.linalg.svd(E, full_matrices = False)
delta = time() - t
print("SVD de una matriz de dimension %dx%d \nTiempo :%0.2f s."
 % (size / 2, size / 4, delta / N))
del E
```

# *Benchmarking sobre numpy : Descomposición de Cholesky*

```python
import numpy as np
from time import time

# Tomemos la aleatoriedad de números aleatorios (para reproducibilidad)
np.random.seed(0)

size = 4096
F = np.random.random((int(size / 2), int(size / 2)))
F = np.dot(F, F.T)

N = 3
t = time()
for i in range(N):
    np.linalg.cholesky(F)
delta = time() - t
print("Descomposicion de Cholesky de una matriz de tamaño  %dx%d \nTiempo %0.2f s." % (size / 2, size / 2, delta / N))
```

# *Benchmarking sobre numpy : Autovalores-Autovectores*

```python
import numpy as np
from time import time

# Tomemos la aleatoriedad de números aleatorios (para reproducibilidad)
np.random.seed(0)

size = 4096

G = np.random.random((int(size / 2), int(size / 2)))

t = time()
for i in range(N):
   np.linalg.eig(G)
delta = time() - t
print("Autovalores/Autovectores de una matriz de dimension %dx%d \nTiempo %0.2f s." % (size / 2, size / 2, delta / N))
```

# *Benchmarking sobre numpy*

Otras tecnologías para acelerar los cálculos numéricos son

- Cython

  Por una parte, Cython es un lenguaje de programación (un superconjunto de Python) que une Python con el sistema de tipado estático de C y C++.

  Por otra parte, cython es un compilador que traduce codigo fuente escrito en Cython en eficiente código C o C++. El código resultante se podría usar como una extensión Python o como un ejecutable.

- Numba

# *Benchmarking sobre numpy*

Otras tecnologías para acelerar los cálculos numéricos son

- Cython

- Numba Numba tiene dos modos de funcionamiento básicos: el modo object y el modo nopython.

  El modo object genera código que gestiona todas las variables como objetos de Python y utiliza la API C de Python para operar con ellas.

  El modo nopython genera código independiente de la API C de Python. Esto tiene la desventaja de que no podemos usar todas las características del lenguaje, pero tiene un efecto significativo en el rendimiento.

# *Lectura/Escritura de archivos*

**NumPy** nos ofrece varias funciones para cargar datos en forma matricial, pero la que usaremos con más frecuencia es la función **loadtxt**. Su único argumento obligatorio es un nombre de archivo o un objeto file desde el que leer los datos. El comportamiento por defecto de loadtxt será:

- ➔ Leer todas las líneas (se pueden saltar las n primeras utilizando el argumento skiprows,
- ➔ salvo las que empiecen por # (se puede cambiar esto utilizando el argumento comments),
- ➔ esperará que los datos estén separados por espacios (se puede cambiar utilizando el argumento delimiter),
- ➔ y devolverá un array de NumPy de tipo float (el tipo se puede asignar con el argumento dtype).

# *Lectura/Escritura de archivos*

Partimos del hecho que tenemos un archivo de texto :

```
1.0000e+00   2.0000e+00
-1.0000e+00   0.0000e+00
```

```
1 import numpy as np
2
3 A  =  np.loadtxt("matriz_a.dat")
4 type(A)
5 A.ndim
6 A.shape
7 A.size
8 A.dtype
9 A.itemsize
```

# *Lectura/Escritura de archivos*

Otras funciones que también sirven para leer datos son:

- ➔ La función **load** sirve para leer datos en el formato comprimido de NumPy, que suele tener las extensiones .npy o .npz.
- ➔ La función **fromfile** sirve para leer datos en formato binario.
- ➔ La función **genfromtxt** es mucho más flexible que loadtxt, y es crucial cuando el archivo está mal formateado o faltan valores en los datos. En la gran mayoría de los casos es suficiente con usar loadtxt.

```
1 data = np.genfromtxt("stockholm_td_adj.dat.txt")
2
3 Anomalias = np.genfromtxt("Anomalias-1880-2017.csv",
  ↪  delimiter=",",skip_header=5,dtype=np.float128)
4
5 Senamhi1 = np.genfromtxt("qc00000547.txt",delimiter="
  ↪  ",dtype=np.float128)
```

▸ Descarga de datos Meteorológicos - Senamhi

# *Lectura/Escritura de archivos*

```
1  data = np.genfromtxt("stockholm_td_adj.dat.txt")
2  type(data)
3  data.ndim
4  data.shape
5  data.size
6  data.dtype
7  data.itemsize
8  data[:,3] #  4ta columna
9  type(data[:,3])
10 data[:,3].sum() # suma de elementos de la 4ta columna
11 data[:,3].min() # minimo
12 data[:,3].argmin() # indice donde ocurre el minimo
13 data[:,3].max()
14 data[:,3].argmax()
15 np.std(data[:,3])
16 def var1(x):
17    return np.mean((x - x.mean())**2)
18 def var2(x):
19    return (np.sum((x - x.mean())**2)/(x.shape[0]-1))
20 var1(data[:,3]) # calculo de la varianza
21 var2(data[:,3]) # calculo de la varianza
```

# *Lectura/Escritura de archivos*

La contrapartida de la función **loadtxt** para escritura la función **savetxt**. Tiene dos argumentos obligatorios: el nombre del archivo y el array que se guardará. Su comportamiento por defecto es guardar los datos con 18 cifras decimales, pero esto se puede cambiar con el argumento fmt.

Para guardar nuestro array en un archivo, simplemente tendremos que hacer:

```
1 A = np.array([[1, 2], [-1, 0]])
2
3 np.savetxt("matriz_a.dat", A, fmt='%.4e')
```

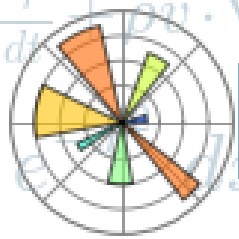
Numpy

Matplotlib

Scipy

Pandas

# Matplotlib



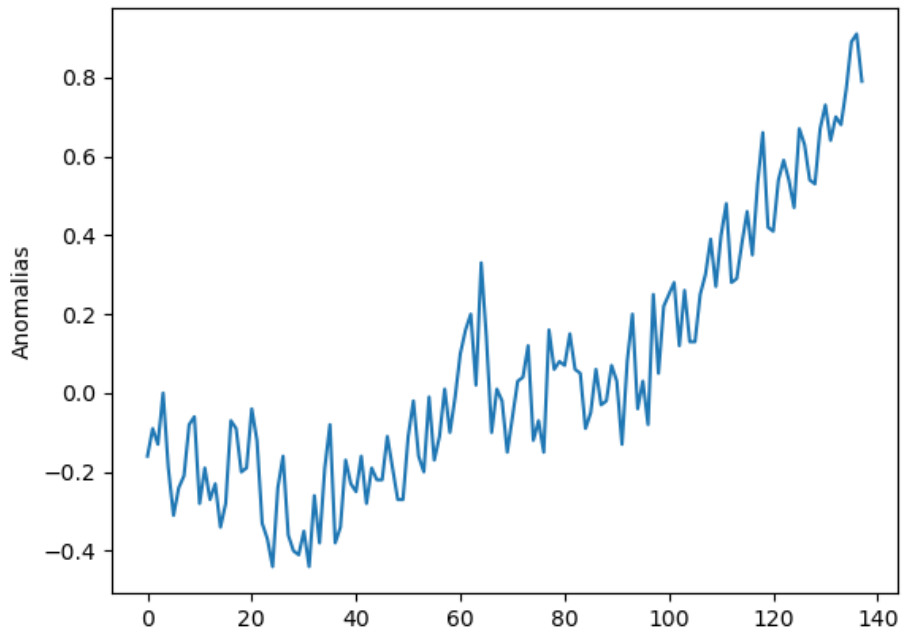
matplotlib

# *Matplotlib : Una introduccion*

Matplotlib es el módulo de dibujo de gráficas 2D y 3D que vamos a utilizar aunque no es el único existente. Matplotlib tiene multitud de librerías de las cuales nosotros, por semejanza a Matlab, utilizaremos pyplot. La web del proyecto `https://matplotlib.org`, donde puede encontrar multitud de programas y ejemplos de como hacer dibujos con Matplotlib.

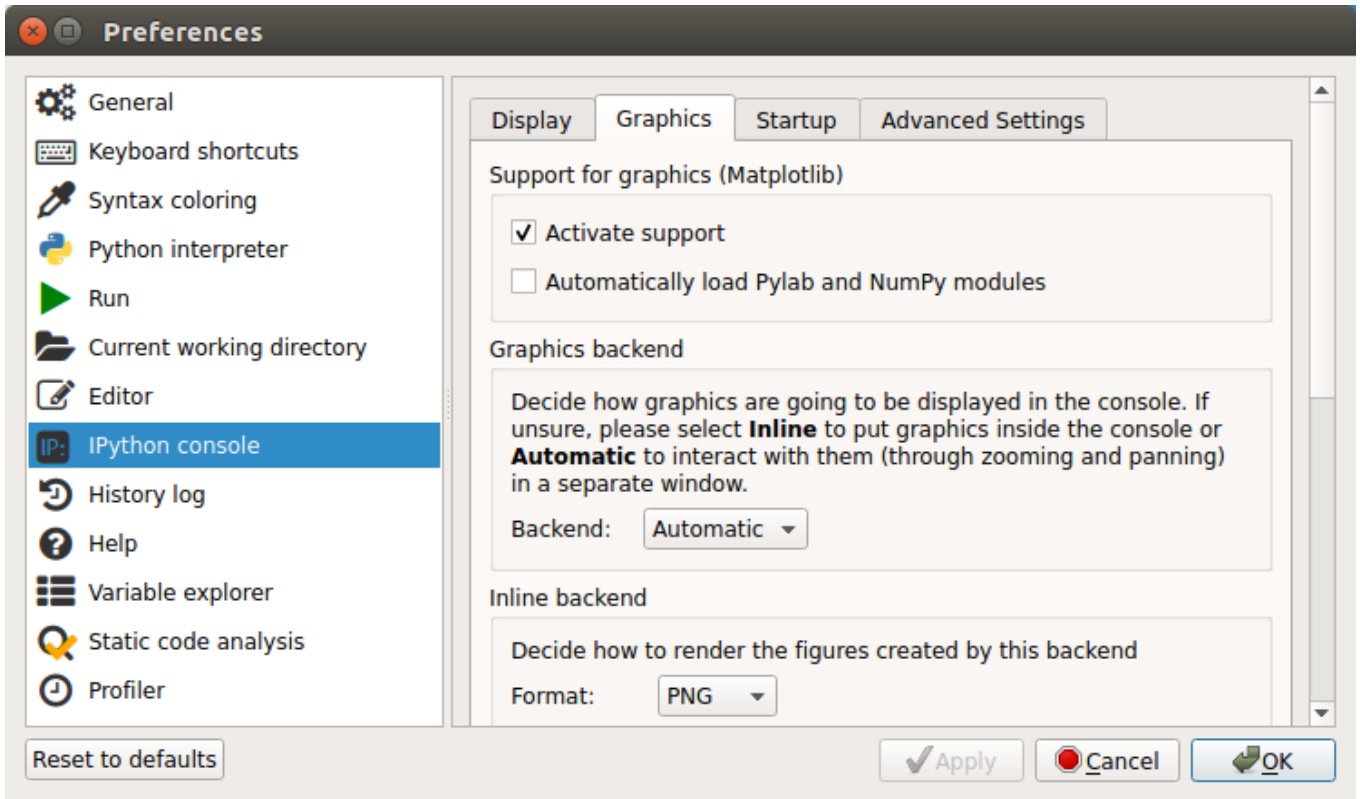
# *Matplotlib : Una introduccion*

```
1  import numpy as np
2  Anomalias = np.genfromtxt("Anomalias-1880-2017.csv",
3                            delimiter=",",
4                            skip_header=5,
5                            dtype=np.float128)
6  Anomalias.shape
7
8  import matplotlib.pyplot as plt
9  plt.plot(Anomalias[:,1])
10 plt.ylabel('Anomalias')
11 plt.show()
```



# *Configuracion de Spyder:Cambiar el backend en spyder*

```
Tools >
  preferences >
    IPython console >
      Graphics >
        Graphics backend >
          Backend: Automatic
```



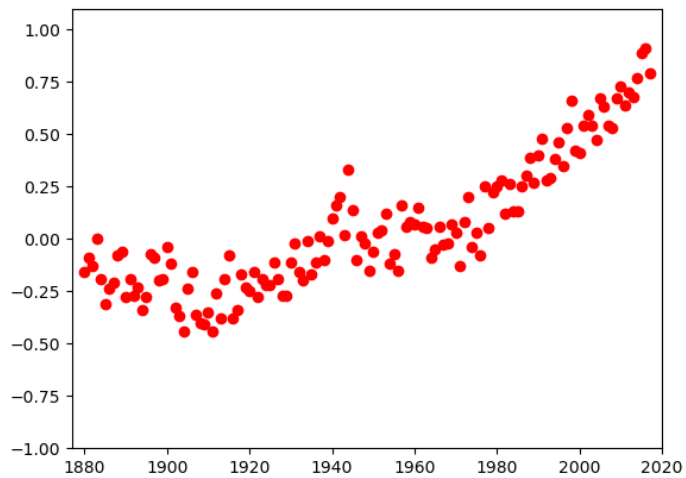
# *Matplotlib : comando plot()*

`plot()` es un comando versátil, y tomará una cantidad arbitraria de argumentos. Por ejemplo, para plotear el conjunto de pares ordenados $(x_i, y_i)$, puede usar el comando:

```
1 plt.plot(Anomalias[:,0],Anomalias[:,1])
```

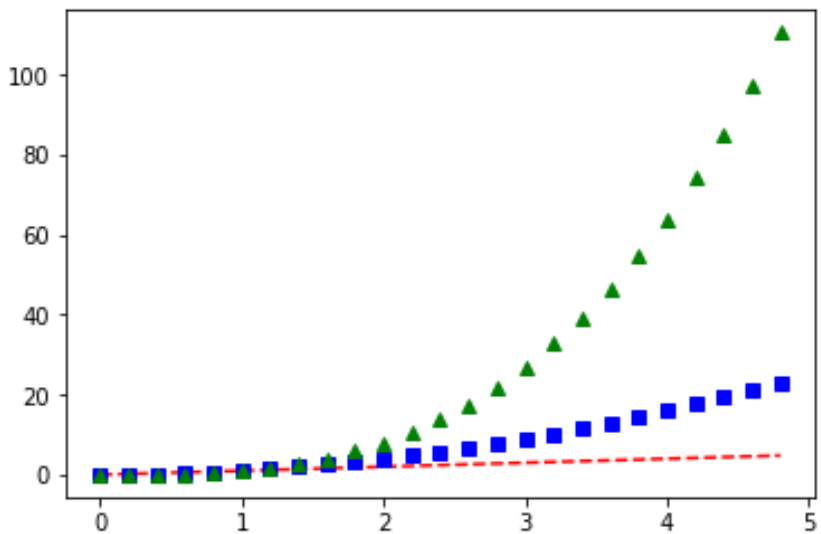
Una modificación al estilo matlab :

```
1 import matplotlib.pyplot as plt
2 Anomalias[:,0].min()
3 Anomalias[:,0].max()
4 Anomalias[:,1].min()
5 Anomalias[:,1].max()
6 plt.plot(Anomalias[:,0],Anomalias[:,1],
7           'ro')
8 plt.axis([1877, 2020, -1, 1.1])
9 plt.show()
```



# *Matplotlib : Algunas otras caracteristicas*

```python
import numpy as np
import matplotlib.pyplot as plt

# dominio
t = np.arange(0., 5., 0.2)

# rojo, azul y verde
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```



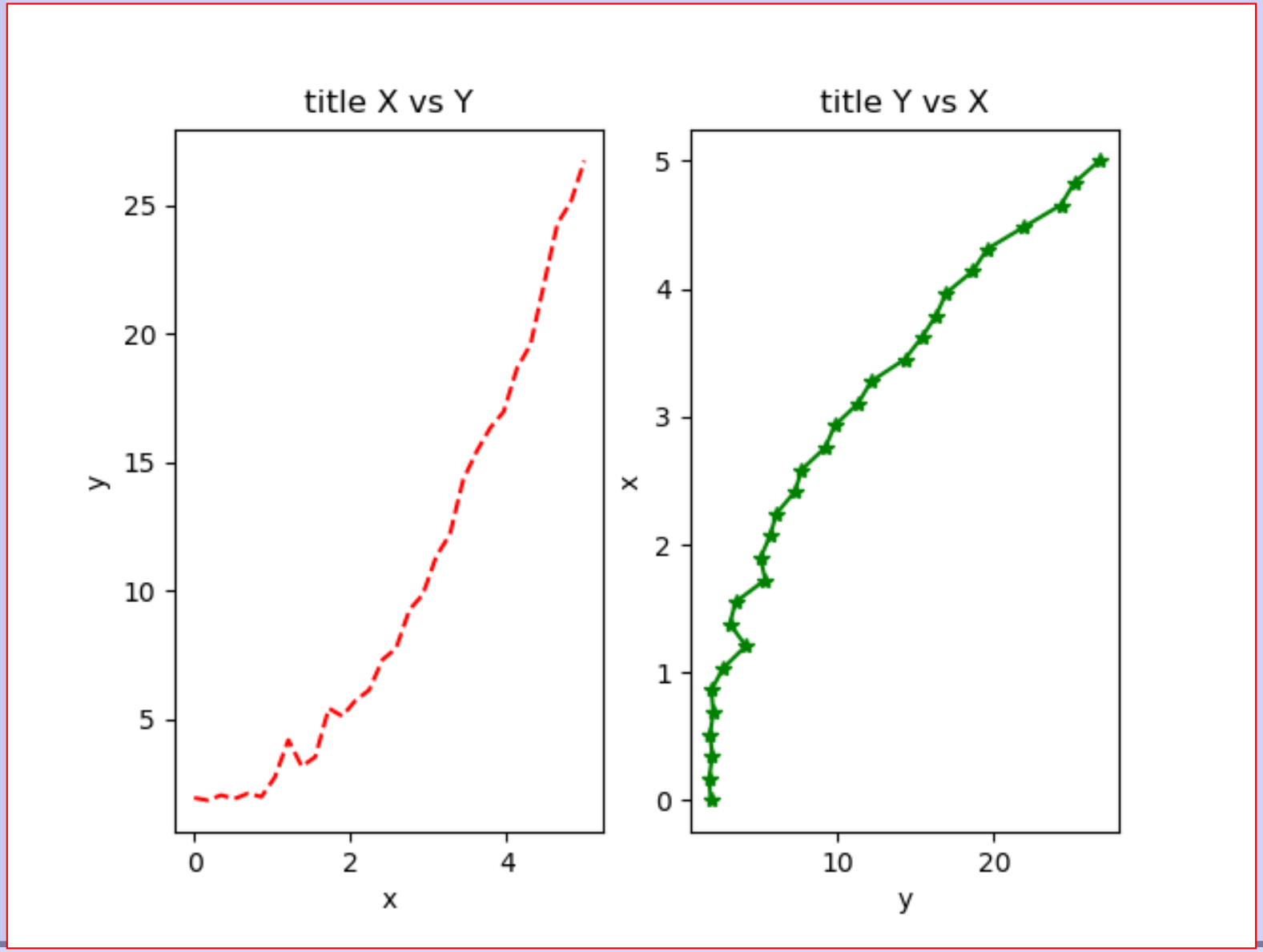
# *Matplotlib : Algunas otras caracteristicas*

```python
import matplotlib.pyplot as plt
import numpy as np

# Generamos data con ruido
x = np.linspace(0, 5, 30)
y = x ** 2 + np.exp(np.random.rand(30))

# Funciones para configurar el grafico
plt.figure()
plt.plot(x, y, 'r')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Funcion cuadratica + Ruido')
plt.show()
```

# *Matplotlib : Varias figuras*

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  x = np.linspace(0, 5, 30)
5  y = x ** 2 + np.exp(np.random.rand(30))
6
7  plt.figure()
8  # ================================================
9  plt.subplot(1,2,1)
10 plt.xlabel('x')
11 plt.ylabel('y')
12 plt.title('title X vs Y')
13 plt.plot(x, y, 'r--')
14 # ================================================
15 plt.subplot(1,2,2)
16 plt.xlabel('y')
17 plt.ylabel('x')
18 plt.title('title Y vs X')
19 plt.plot(y, x, 'g*-');
20 # ================================================
21 plt.show()
```

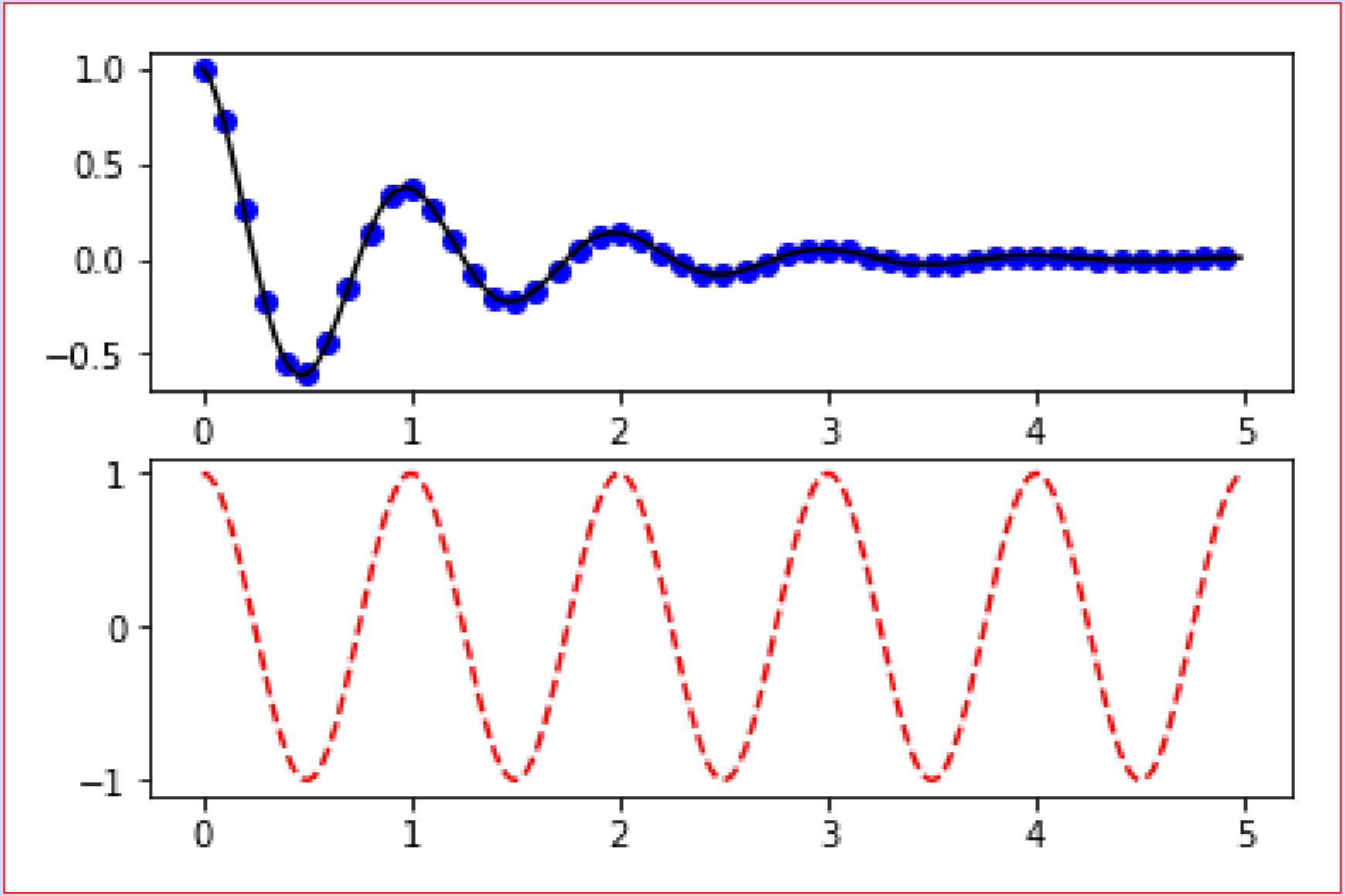
# *Matplotlib : Varias figuras*



# *Matplotlib : Varias figuras*

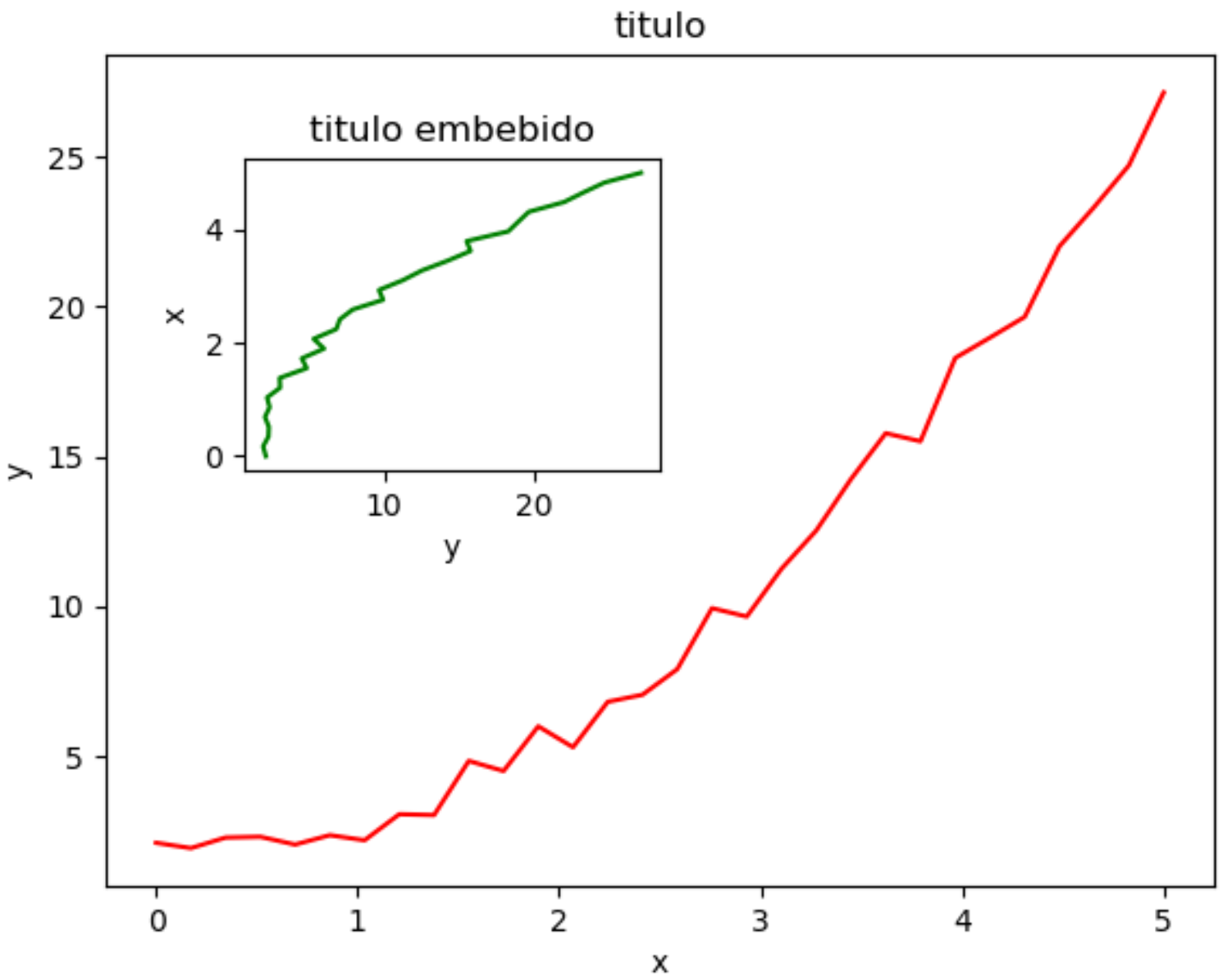
```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  def f(t):
5      return np.exp(-t) * np.cos(2*np.pi*t)
6
7  t1 = np.arange(0.0, 5.0, 0.1)
8  t2 = np.arange(0.0, 5.0, 0.02)
9
10 plt.figure(1)
11 plt.subplot(211)
12 plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
13
14 plt.subplot(212)
15 plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
16 plt.show()
```

# *Matplotlib : Varias figuras*



# *Matplotlib : Varias figuras*

```python
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 5, 30)
y = x ** 2 + np.exp(np.random.rand(30))

fig = plt.figure()

axes1 = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # Principal
axes2 = fig.add_axes([0.2, 0.5, 0.3, 0.3]) # Embebido

# Figura principal
axes1.plot(x, y, 'r')
axes1.set_xlabel('x')
axes1.set_ylabel('y')
axes1.set_title('titulo')

# Figura inbebida (insertada)
axes2.plot(y, x, 'g')
axes2.set_xlabel('y')
axes2.set_ylabel('x')
axes2.set_title('titulo embebido');

plt.show()
```




```
24 plt.show()
```

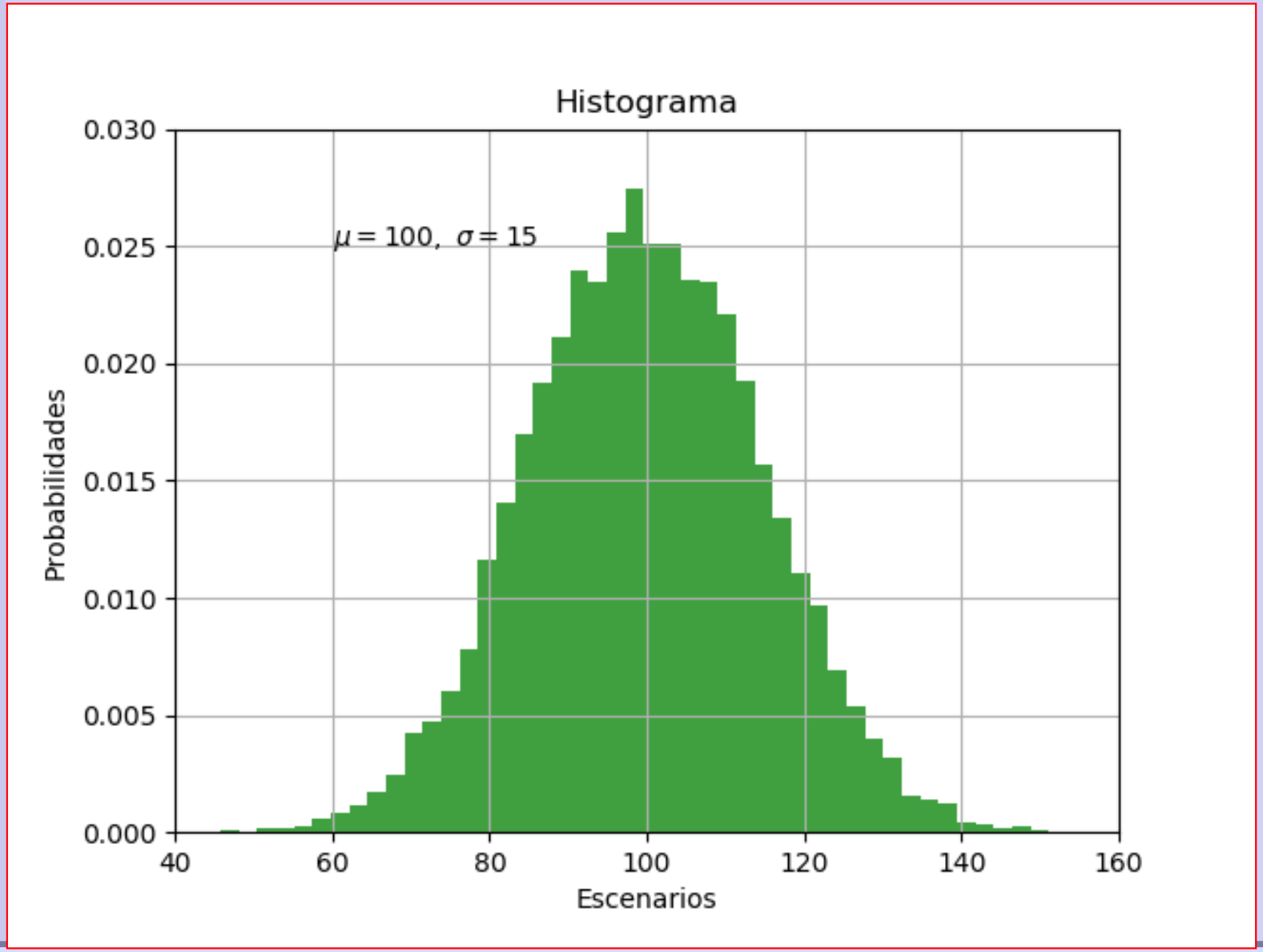
# *Matplotlib : Varias figuras - varias ventanas*

```
1  import matplotlib.pyplot as plt
2  plt.figure(1)                  # 1era figura
3  plt.subplot(211)               # 1er subplot de la 1era fig
4  plt.plot([1, 2, 3])
5  plt.subplot(212)               # 2do subplot de la 1era fig
6  plt.plot([4, 5, 6])
7
8
9  plt.figure(2)                  # 2da figura
10 plt.plot([4, 5, 6])            # crea subplot(111) por defecto
11
12 plt.figure(1)                  # fig 1 ; subplot(212)
13 plt.subplot(211)               # crea subplot(211) en fig1
14 plt.title('Facil como 1 2 3') # titulo subplot 211
```

# Matplotlib : Trabajando con textos

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # Fijamos la semilla
5  np.random.seed(19680801)
6
7  mu, sigma = 100, 15
8  x = mu + sigma * np.random.randn(10000)
9
10 # the histogram of the data
11 n, bins, patches = plt.hist(x, 50, normed=1, facecolor='g',
   ↪  alpha=0.75)
12
13
14 plt.xlabel('Escenarios')
15 plt.ylabel('Probabilidades')
16 plt.title('Histograma')
17 plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
18 plt.axis([40, 160, 0, 0.03])
19 plt.grid(True)
20 plt.show()
```

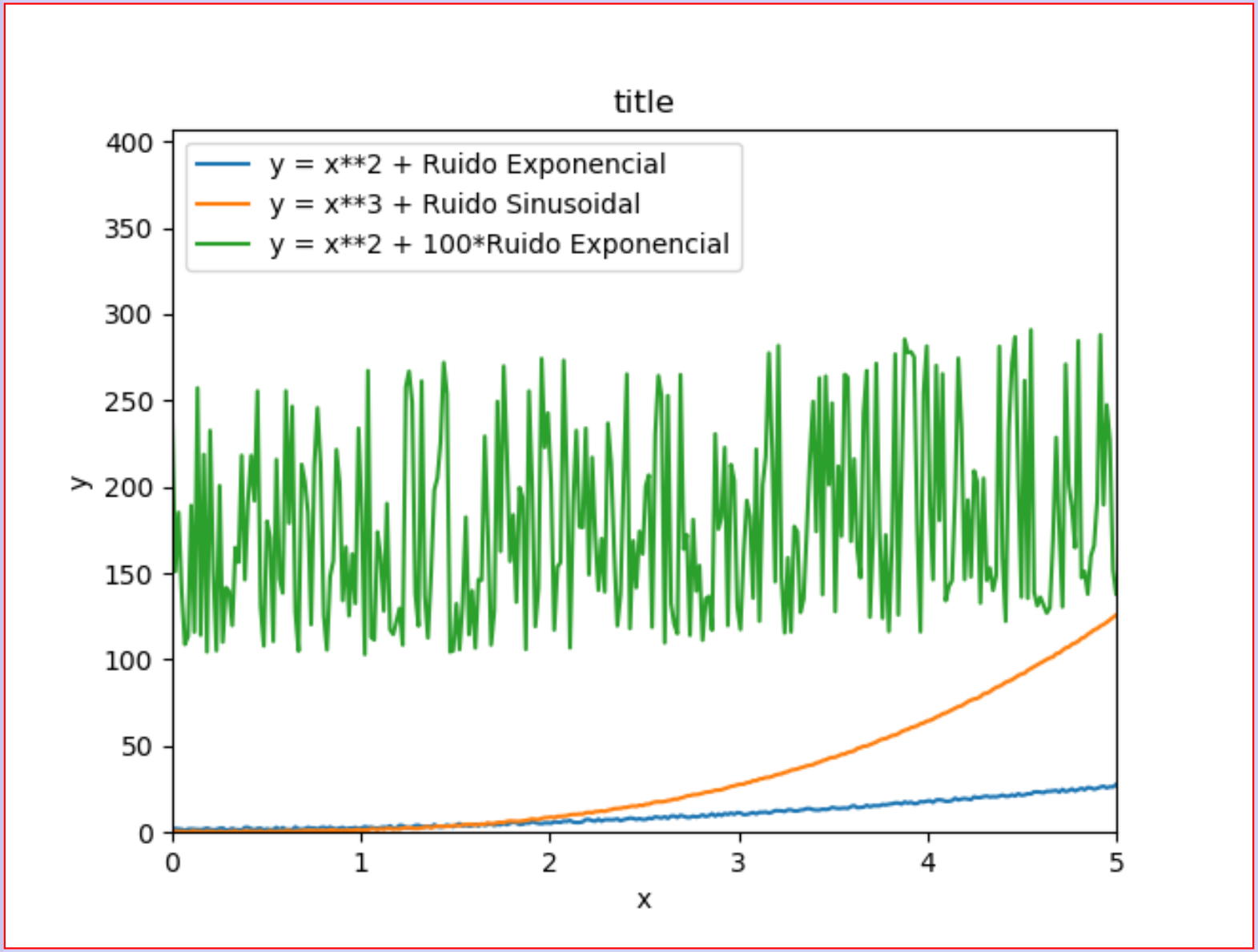
# *Matplotlib : Trabajando con textos*



# Matplotlib : Trabajando con leyendas

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  x = np.linspace(0, 5, 300)
5  y1 = x ** 2 + np.exp(np.random.rand(300))
6  y2 = x ** 3 + np.sin(np.random.rand(300))
7  y3 = x ** 2 + 100*np.exp(np.random.rand(300))
8  fig, ax = plt.subplots()
9
10 axes = plt.gca()
11 ymin = min(y1.min(),y2.min(),y3.min())
12 ymax = 1.4*max(y1.max(),y2.max(),y3.max())
13 axes.set_xlim([x.min(),x.max()])
14 axes.set_ylim([ymin,ymax])
15
16 ax.plot(x, y1, label="y = x**2 + Ruido Exponencial")
17 ax.plot(x, y2, label="y = x**3 + Ruido Sinusoidal")
18 ax.plot(x, y3, label="y = x**2 + 100*Ruido Exponencial")
19 ax.legend(loc=2); # upper left corner
20 ax.set_xlabel('x')
21 ax.set_ylabel('y')
22 ax.set_title('title');
23
24 plt.show()
```

# *Matplotlib : Trabajando con leyendas*

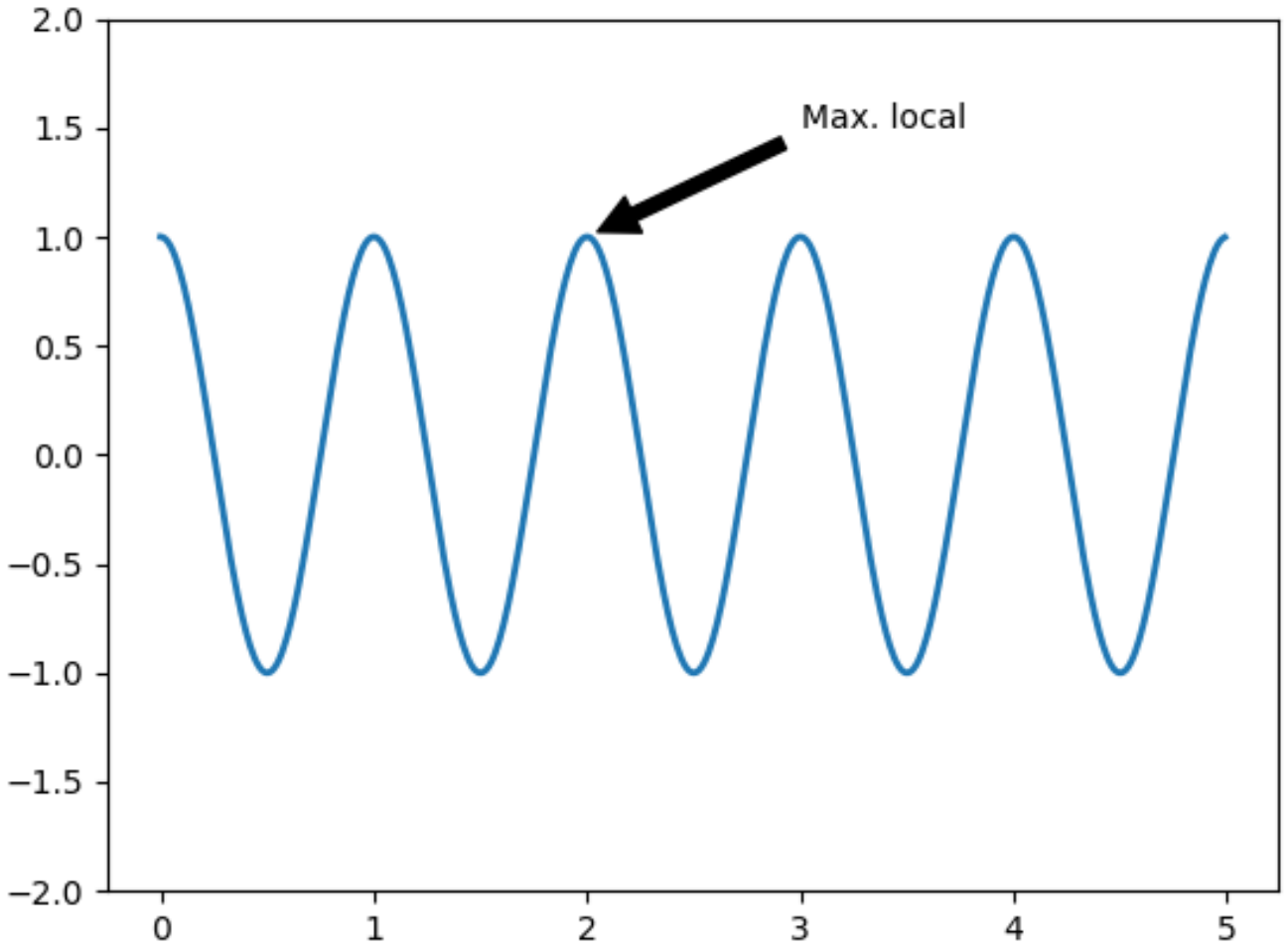



```
plt.show()
```

# *Matplotlib : Anotaciones*

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  ax = plt.subplot(111)
5
6  t = np.arange(0.0, 5.0, 0.01)
7  s = np.cos(2*np.pi*t)
8  line, = plt.plot(t, s, lw=2)
9
10 plt.annotate('Max. local', xy=(2, 1), xytext=(3, 1.5),
11 arrowprops=dict(facecolor='black', shrink=0.05),
12 )
13
14 plt.ylim(-2,2)
15 plt.show()
```

# *Matplotlib : Anotaciones*



# *Aplicacion : Analisis de Retornos : SP500*

```
1  from numpy import arange, loadtxt, zeros
2  import matplotlib.pyplot as plt
3
4  DataSP500_Open = loadtxt("GSPC_2010-2018.csv",skiprows=1,
   ↪  usecols=[1], delimiter=",")
5  print("Indice SP500 : 1/1/2010 - 1/1/2018")
6  print(DataSP500_Open[0:10])
7
8  # 1. Calculamos los retornos diarios
9  diffs = DataSP500_Open[1:] - DataSP500_Open[:-1]
10 returns = diffs / DataSP500_Open[:-1]
11
12 # 2. CReamos la linea de retorno 0
13 days = arange(len(returns))
14 zero_line = zeros(len(returns))
15
16 # 3. Creamos un grafico
17 plt.plot(days, zero_line, 'r*', days, returns * 100, 'b-')
18 plt.title("Retornos diarios del indice SP500 del 2010-2018 (%)")
19 plt.xlim(xmax=len(returns))
20 plt.show()
```

Numpy

Matplotlib

Scipy

Pandas

# SciPy

- ➔ Colección de algoritmos matemáticos y funciones
  - ➔ Construido sobre NumPy
- ➔ Poder al intérprete interactivo
  - ➔ Procesamiento de datos y prototipado de sistemas
  - ➔ Compite con Matlab, IDL, Octave, R, y SciLab

# ¿Que es Scipy?

**Scipy**

El paquete **scipy** contiene varias cajas de herramientas dedicadas a problemas comunes en informática científica. Sus diferentes submódulos corresponden a diferentes aplicaciones, tales como interpolación, integración, optimización, Procesamiento de imágenes, estadísticas, funciones especiales, etc.

**Scipy** puede compararse con otras bibliotecas estándar de computación científica, como GSL (GNU Scientific Biblioteca para C y C ++), o los toolboxes de Matlab. **Scipy** es el paquete central para rutinas científicas en Python; es destinado a funcionar de manera eficiente en matrices **numpy**, de modo que **numpy** y **scipy** trabajen de la mano.

# *Submodulos de Scipy*

| Submodulo | Campo de Aplicacion |
|---|---|
| scipy.cluster | Vector quantization / Kmeans |
| scipy.constants | Physical and mathematical constants |
| scipy.fftpack | Fourier transform |
| scipy.integrate | Integration routines |
| scipy.interpolate | Interpolation |
| scipy.io | Data input and output |
| scipy.linalg | Linear algebra routines |
| scipy.ndimage | n-dimensional image package |
| scipy.odr | Orthogonal distance regression |
| scipy.optimize | Optimization |
| scipy.signal | Signal processing |
| scipy.sparse | Sparse matrices |
| scipy.spatial | Spatial data structures and algorithms |
| scipy.special | Any special mathematical functions |
| scipy.stats | Statistics |

# *Paseo por Scipy : scipy.linalg*

El módulo **scipy.linalg** proporciona operaciones de álgebra lineal estándar, usando librerias clasicas (con probada eficiencias :BLAS, LAPACK).

La función **scipy.linalg.det()** calcula el determinante de una matriz cuadrada:

```
from scipy import linalg
arr = np.array([[1, 2],[3, 4]])
linalg.det(arr)

# Matrices Sparse : https://math.nist.gov/MatrixMarket/
from scipy import io as spio
spio.mminfo("mahindas.mtx")
mahindas = spio.mmread("mahindas.mtx")
type(mahindas)
mahindas.shape

# COnversion de sparse a densa
mahindas_Densa=mahindas.todense()
mahindas_Densa[1:10,1:10]
linalg.det(mahindas_Densa)
```

# *Paseo por Scipy : **scipy.linalg***

La función scipy.linalg.inv() calcula la matriz inversa de una matriz cuadrada:

```
1  from scipy import linalg
2  import numpy as np
3
4  # Funcion para generar una matriz cuadrada
5  def gen_ex(d0):
6     x = np.random.randn(d0,d0)
7     return x.T + x
8
9  # generamos una matriz : 1000x1000
10 mat1 =gen_ex(10**3)
11 # Memoria ram ocupada
12 mat1.nbytes
13
14 i_mat1 = linalg.inv(mat1)
15
16 # Comparamos el producto de una matriz con su inversa
17 # con la matriz identidad de las dimensiones correctas
18 np.allclose(np.dot(mat1, i_mat1), np.eye(10**3))
```

Para notar la importancia de mencionar este tema, recordemos que hace unos años la empresa netflix realizo un concurso, del cual aun se puede encontrar su pagina web

https://netflixprize.com

El equipo ganador se llevo un premio de $10^6$USD.Podemos ver el paper que gano el concurso :

https://www.netflixprize.com/assets/GrandPrize2009_BPC_BigChaos.pdf

Si buscamos la palabra **matrix** la podemos encontrar mas de 30 veces, y si buscamos la palabra **factorization** la podemos encontrar mas de 20 veces.El algoritmo que ganó el concurso de Netflix fue un sistema basado en métodos de Factorización de matrices.

Factorización de matrices en sistemas de ecuaciones lineales : Dos de las Factorización de matrices más utilizadas y que tal vez mucha gente las haya escuchado nombrar alguna vez son la factorización LU y la factorización QR; las cuales se utilizan a menudo para resolver sistemas de ecuaciones lineales.

**Factorización LU : $A = LU$**

En álgebra lineal, la **factorización** o descomposición LU (del inglés Lower-Upper) es una forma de **factorización** de una matriz como el producto de una matriz triangular inferior y una superior. La **factorización** LU expresa el método de Gauss en forma matricial. Así por ejemplo, tenemos que PA=LU donde P es una matriz de permutación. Una condición suficiente para que exista la factorización es que la matriz A sea una matriz no singular.

# *Paseo por Scipy :* ***scipy.linalg***

## *Factorizacion de Matrices : LU*

```
1  import numpy as np
2  import scipy.sparse as sp
3  import scipy.linalg as la
4
5  A = np.array([[7, 3, -1, 2] ,[3, 8, 1, -4] ,[-1, 1, 4, -1] ,[2,
   ↪  -4, -1, 6]])
6  P, L, U = la.lu(A)
7
8  # para verificar tenemos dos formas
9  Verif1 = (A == np.dot(L,U))
10 Verif2 = (L @ U == A)
```

## Factorización QR : A = QR

La descomposición o factorización QR consiste en la descomposición de una matriz como producto de una matriz ortogonal ($Q^T \cdot Q=I$) por una matriz triangular superior. la factorización QR es ampliamente utilizada en las finanzas cuantitativas como base para la solución del problema de los mínimos cuadrados lineales, que a su vez se utiliza para el análisis de regresión estadística.

# *Paseo por Scipy :* ***scipy.linalg***

## *Factorizacion de Matrices :QR*

```
A = np.array([[12, -51, 4],[6, 167, -68],[-4, 24, -41]])
Q, R = la.qr(A)

# para verificar tenemos dos formas
Verif1 = (A == np.dot(Q,R))
Verif2 = (Q @ R == A)
```

# *Aproximación de funciones*

El problema que abordaremos es el de reconstruir una función $f$ definida sobre un dominio real y a valores en $\mathbb{R}$, a partir de información incompleta o bien contaminada por errores. En tales circunstancias la reconstrucción no podrá ser perfecta y por lo tanto se tratará de una *aproximación* de la función $f$. En cursos de matemáticas basicas esto se ha resuelto , en distintos contextos. Algunas de estas soluciones conocidas son:

- ➔ desarrollo en serie de Taylor en torno a un punto dado;
- ➔ desarrollo en serie de Fourier;
- ➔ aproximación polinomial de una función continua sobre un intervalo cerrado, según el teorema de Stone-Weierstrass.

# *Aproximación de funciones*

Los polinomios serán nuestra principal herramienta y por lo tanto recordaremos este importante teorema. Este dice que el conjunto de los polinomios es *denso* en el conjunto de las funciones continuas, lo que equivale a decir que toda función continua puede ser considerada como el límite de una sucesión de polinomios, en la norma de la convergencia uniforme.

**Stone-Weierstrass**

Sea $f : [a, b] \to \mathbb{R}$ continua. $\forall \varepsilon > 0$ $\exists p$, polinomio, tal que

$$\forall x \in [a, b] \qquad |f(x) - p(x)| \leq \varepsilon.$$

# *Interpolación Polinomial*

Sean $f : [a, b] \to \mathbb{R}$, una malla $T = \{x_i\}_{i=0}^{n} \subset [a, b]$, y los valores de $f$ $y_i = f(x_i)$ para $i = 0, 1, ..., n$. Se desea encontrar un polinomio de grado menor o igual que $n$, $p \in \mathcal{P}_n$, que interpole a $f$ sobre todos los puntos de la malla $T$, es decir,

$$p(x_i) = f(x_i) = y_i \quad \text{para todo } i = 0, 1, ..., n.$$

Este problema tiene solución única si los puntos de la malla $T$, también llamados nodos de interpolación, son todos distintos.

**Teorema**

Si los $(n + 1)$ nodos de interpolación de la malla $T$ son todos distintos, entonces existe un único polinomio de interpolación de grado menor o igual que $n$, $p \in \mathcal{P}_n$, $p(x_i) = y_i$, $\forall i = 0, ..., n$.

Proof.

Probaremos en primer lugar la unicidad. Supongamos que hay dos polinomios de interpolación de grado menor o igual que $n$, $p, q \in \mathcal{P}_n$,

$$p(x_i) = q(x_i) = y_i \quad \forall i = 0, 1, ..., n.$$

Entonces el polinomio $h = p - q$ será también un polinomio de grado menor o igual que $n$ que tendrá $(n + 1)$ raíces

$$h(x_i) = 0, \quad \forall i = 0, 1, ..., n.$$

Pero esto solo es posible si $h$ es el polinomio nulo, es decir $p = q$.

La demostración de la existencia del polinomio de interpolación es constructiva. □

Continuacion de la

Proof.

Sean

$$\ell_{n,i}(x) = \frac{\prod_{\substack{j=0 \\ j\neq i}}^{n}(x - x_j)}{\prod_{\substack{j=0 \\ j\neq i}}^{n}(x_i - x_j)} \quad \text{para } i = 0, 1, ..., n. \tag{1}$$

Se verifica fácilmente que $\forall i = 0, 1, ..., n$

$$\ell_{n,i} \in \mathcal{P}_n$$

$$\ell_{n,i}(x_k) = \delta_{ik} = \begin{cases} 1 & \text{si } i = k \\ 0 & \text{si } i \neq k \end{cases}$$

Fin de la

Proof.

y que por lo tanto

$$L_n(x) = \sum_{i=0}^{n} y_i \ell_{n,i}(x) \tag{2}$$

es de grado menor o igual a $n$ e interpola, es decir, es el único polinomio de interpolación. □

La expresión (2) del polinomio de interpolación recibe el nombre de polinomio de **Lagrange** y los polinomios definidos en (1) se llaman polinomios de base de Lagrange.

La calidad del polinomio de interpolación como aproximación de $f$ se establece en el teorema acerca del error que sigue. Denotaremos por $\overline{co}(x_0, x_1, ..., x_n, x)$ al menor intervalo cerrado que contenga a $x_0, x_1, ..., x_n, x$.

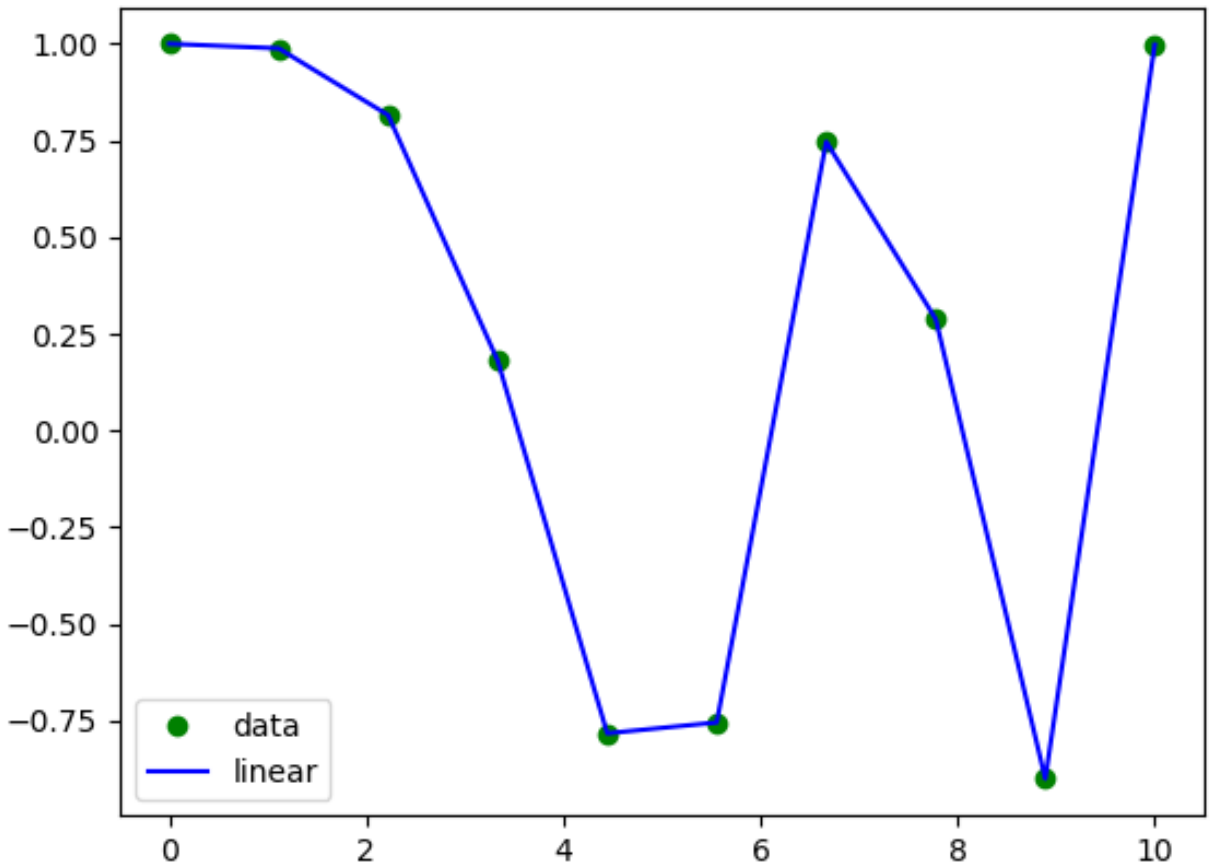
**Teorema**

Si $f$ es $(n+1)$ veces continuamente derivable sobre $[a, b]$, entonces $\forall x \in [a, b] \quad \exists \xi \in \overline{co}(x_0, x_1, ..., x_n, x)$ tal que

$$e_n(x) = f(x) - L_n(x) = \frac{\prod_{j=0}^{n}(x - x_j) f^{(n+1)}(\xi)}{(n+1)!}$$

# *Aproximación polinomial por pedazos : Funciones Spline*

Una idea muy intuitiva para diseñar una curva que interpole un conjunto de datos $\{(x_i, y_i)\}_{i=1}^{n}$ es la de unir los puntos de abscisas consecutivas mediante rectas:



# *Aproximación polinomial por pedazos : Funciones Spline*

La función $\sigma$ que resulta de este método es lineal por pedazos y continua. Claramente es única, pues existe una sola recta que une al punto $(x_i, y_i)$ con el punto $(x_{i+1}, y_{i+1})$. Se intuye que si los datos corresponden a valores de una función continua entonces al aumentar el número de puntos a interpolar, en el mismo intervalo inicial, la poligonal debería converger a la función, tal como lo hacen los dibujos generados por ploteos computacionales. Pero si no se pretende mejorar el aspecto de la curva interpolante por la vía de aumentar los puntos de interpolación, la alternativa sería hacerla menos quebrada, es decir que tuviera una o más derivadas continuas. Esto obliga a aumentar el grado de los polinomios a considerar. ¿Cómo se relaciona el grado con la suavidad y la unicidad de la interpolante? Las funciones **Spline** de interpolación que definimos a continuación entregan la respuesta a esta pregunta.

**Definicion**

Dados una malla ordenada $T = \{x_i\}_{i=1}^n$ en $[a, b]$, es decir, $a < x_1 < x_2 < ... < x_n < b$ y los valores $y_i, i = 1, 2, ..., n$, para un natural $m$, $1 \leq m \leq n$, se define la **función Spline de interpolación de orden** $m$ asociada a estos datos como $\sigma_m : [a, b] \to \mathbb{R}$, tal que

1. $\sigma_m(x_i) = y_i, \qquad \forall i = 1, 2, ..., n$
2. $\sigma_{m|_{[x_i, x_{i+1}]}} \in \mathcal{P}_{2m-1} \qquad \forall i = 1, 2, ..., n-1,$
   $\sigma_{m|_{[a, x_1]}} \in \mathcal{P}_{m-1}$
   $\sigma_{m|_{[x_n, b]}} \in \mathcal{P}_{m-1}$
3. $\sigma_m \in C^{2m-2}_{[a,b]}$.

Note que los polinomios interiores a la malla son siempre de grado impar.

La poligonal dibujada un par de slides atras satisface esta definición con $m = 1$ y por lo tanto es una función Spline. Para $m = 2$ se obtiene la más popular de las funciones Spline, la cúbica por pedazos con dos derivadas continuas. Esta interpolante se encuentra disponible en todos los software matemáticos. Nosotros mostraremos su construcción y la utilizaremos para facilitar la interpretación del modelo del cual surgen las funciones Spline.

La unicidad de la función $\sigma_m$ de la definicion anterior se obtiene luego de establecer una propiedad de optimalidad de gran importancia teórica y práctica: **la función Spline es una proyección ortogonal**.

# *Aproximación polinomial por pedazos : Funciones Spline*

Siempre que una aproximante corresponda a una proyección ortogonal, se tendrá la posibilidad de develar el modelo subyacente, al reconocer el *criterio de optimalidad* representado por la *distancia* que ella minimiza y los *grados de libertad* o la riqueza del *subespacio* donde se busca esta solución optimal. El caso de las funciones Spline es extremadamente sorprendente e interesante. Como veremos a continuación, en el problema optimal resuelto por la función Spline la única decisión a priori que restringirá el modelo subyacente, consiste en la elección de la distancia a minimizar. El subespacio donde se busca la proyección es el más amplio posible, no tiene más restricciones que la de poder definir correctamente la distancia elegida y por supuesto debe contener solamente funciones que interpolen los datos dados. En ninguna parte se pide que la búsqueda se limite a los polinomios ni a las polinomiales por pedazos, como se podría suponer.

**Definicion**

Se define el espacio de funciones $\mathcal{H}_m$ como

$$\mathcal{H}_m = \{u : [a,b] \to \mathbb{R} | \int_a^b (u^{(m)}(x))^2 dx < +\infty\}$$

y el producto de funciones de $\mathcal{H}_m$ como

$$\forall u, v \in \mathcal{H}_m \quad \langle u, v \rangle_m = \int_a^b u^{(m)}(x) v^{(m)}(x) dx.$$

El producto $\langle \cdot, \cdot \rangle_m$ de la definición anterior es un **semi producto interno** en $\mathcal{H}_m$, es decir, satisface todas las propiedades de un producto interno excepto tener núcleo reducido al cero,

$$\langle u, u \rangle_m = 0 \not\Rightarrow u = 0.$$

Consideremos el **subespacio vectorial** $I_0$ de $H_m$ definido por

$$I_0 = \{u \in \mathcal{H}_m | u(x_i) = 0, \forall i = 1, 2, ..., n\} \tag{3}$$

y el traslado de $I_0$, el **subespacio afín**

$$I_y = \{u \in \mathcal{H}_m | u(x_i) = y_i, \forall i = 1, 2, ..., n\}.$$

(**Recuerde** $I_0$ es subespacio vectorial pues $u, v \in I_0 \Rightarrow (u - v) \in I_0$ e $I_y$ es subespacio afín o trasladado de $I_0$ pues $u, v \in I_y \Rightarrow (u - v) \in I_0$)

La proyección ortogonal de la función nula o 0 de $\mathcal{H}_m$ en el subespacio afín $I_y$ con respecto al semiproducto $\langle \cdot, \cdot \rangle_m$, que denotaremos $u^*$, se caracteriza (como es sabido) por

$$\begin{aligned} u^* &\in I_y \\ \langle u - u^*, u^* \rangle_m &= 0 \quad \forall u \in I_y, \end{aligned} \tag{4}$$

Como en toda proyección ortogonal, se tiene que ésta realiza una distancia mínima, es decir, si definimos la seminorma inducida por el semiproducto $|u|_m = \sqrt{\langle u, u \rangle_m}$, entonces la proyección ortogonal del cero en $I_y$ se caracteriza por ser solución del problema de minimización

$$|u^*|_m = \min_{u \in I_y} |u|_m \tag{5}$$

A continuación veremos que $\sigma_m = u^*$ probando que $\sigma_m$ satisface (4) y que es única.

# *Paseo por Scipy : scipy.interpolate*

**scipy.interpolate** es útil para ajustar una función a partir de datos experimentales y así evaluar puntos donde ninguna medida existe El módulo se basa en las subrutinas de `Fortran FITPACK`.

Imaginemos datos experimentales cerca de una función seno:

```
1 import numpy as np
2 measured_time = np.linspace(0, 1, 10)
3 noise = (np.random.random(10)*2 - 1) * 1e-1
4 measures = np.sin(2 * np.pi * measured_time) + noise
```

**scipy.interpolate.interp1d** puede construir una función de interpolación lineal:

```
1 interpolation_time = np.linspace(0, 1, 50)
2 linear_interp = interp1d(measured_time, measures)
3 linear_results = linear_interp(interpolation_time)
```

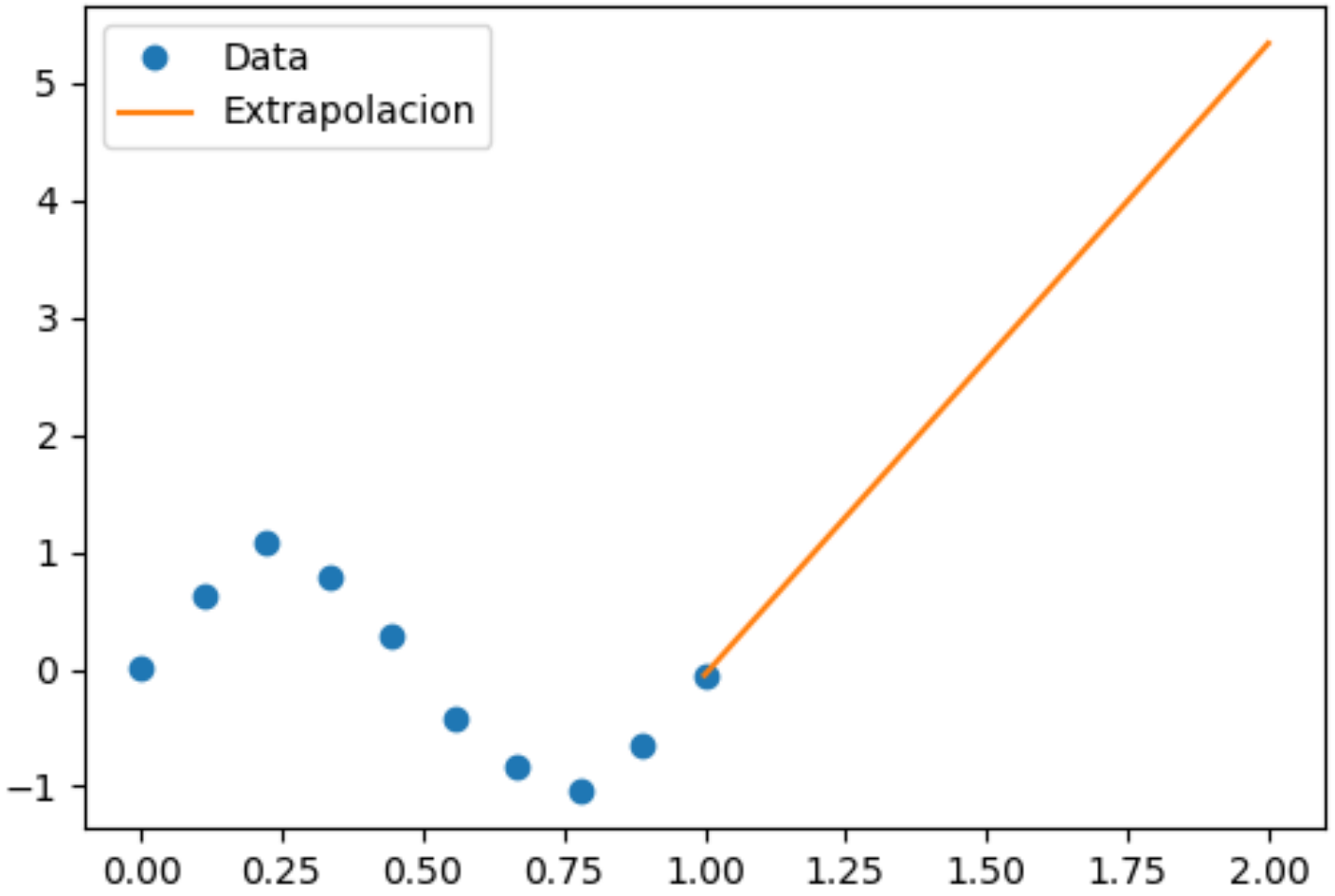
Hay que tener cuidado con lo que significa una interpolación. En el ejemplo anterior podriamos tener problemas al evaluar nuestra interpolacion lineal en el punto 1.1 (tener en cuenta que el dominio de estudio es el intervalo [0,1]).

```
1 linear_interp(1.1)
2
3 ValueError: A value in x_new is above the interpolation range.
```

# *Paseo por Scipy : scipy.interpolate*

Para "corregir" este ultimo problema agregaremos una opcion a la funcion **interp1d** : `fill_value='extrapolate'`

```
linear_interp = interp1d(measured_time, measures,
↪  fill_value='extrapolate')

ExtraDominio = np.linspace(1, 2, 50)
ExtraPolacion = linear_interp(ExtraDominio)
from matplotlib import pyplot as plt
plt.figure(figsize=(6, 4))
plt.plot(measured_time, measures, 'o', ms=6, label='measures')
plt.plot(ExtraDominio, ExtraPolacion, label='Extrapolacion')
plt.legend()
plt.show()
```

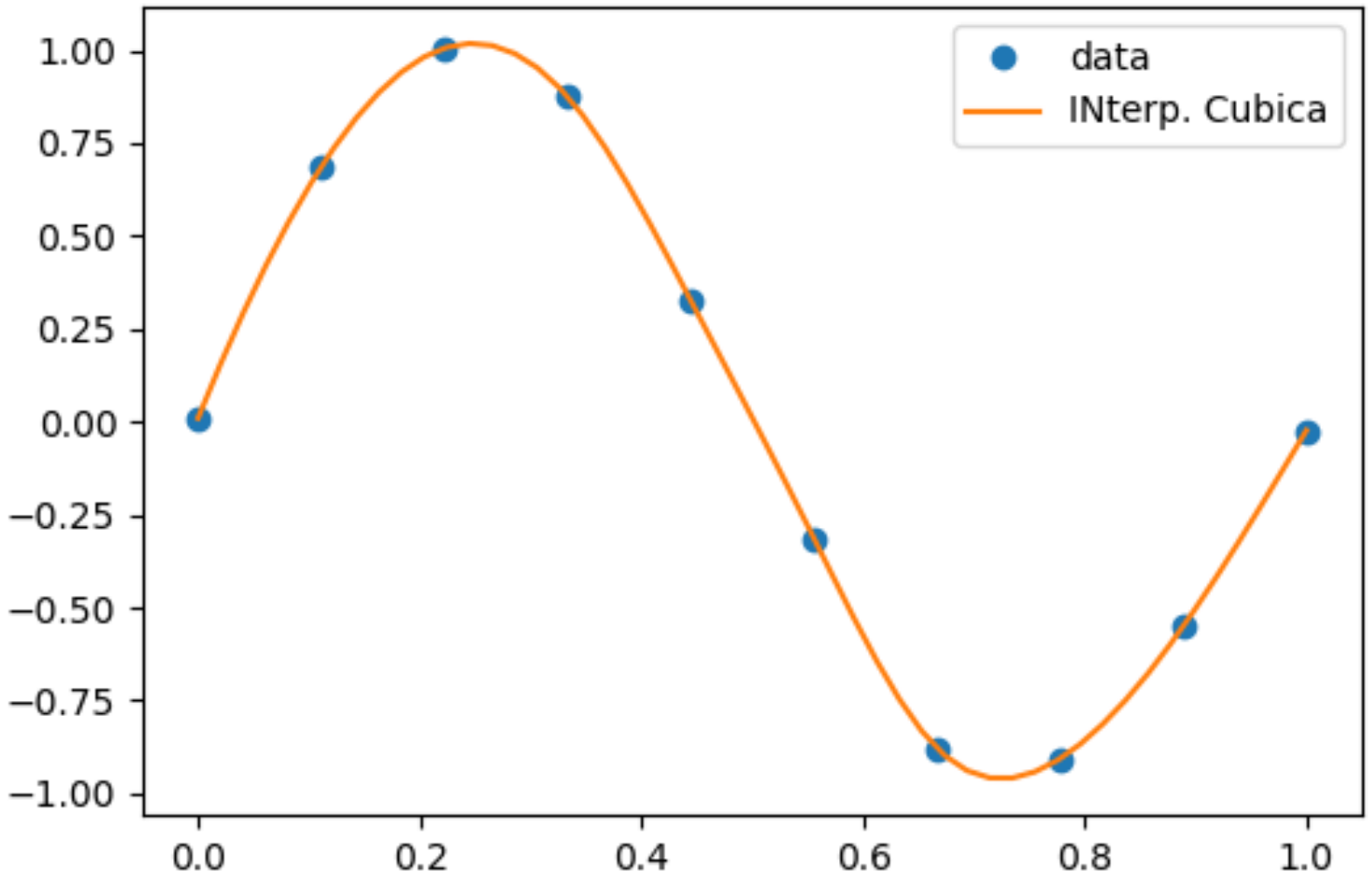

Data
Extrapolacion
5
4
3
2
1
0
−1
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00

# *Paseo por Scipy :* ***scipy.interpolate***

Interpolacion mediante un spline cubico :

```
# Generamos data simulada
import numpy as np
np.random.seed(0)
measured_time = np.linspace(0, 1, 10)
noise = 1e-1 * (np.random.random(10)*2 - 1)
measures = np.sin(2 * np.pi * measured_time) + noise

# Interpolamos
from scipy.interpolate import interp1d
interpolation_time = np.linspace(0, 1, 50)
cubic_interp = interp1d(measured_time, measures, kind='cubic')
cubic_results = cubic_interp(interpolation_time)

# Graficamos la data y las interpolaciones
from matplotlib import pyplot as plt
plt.figure(figsize=(6, 4))
plt.plot(measured_time, measures, 'o', ms=6, label='data')
plt.plot(interpolation_time, cubic_results, label='INterp.
↪  Cubica')
plt.legend()
plt.show()
```

# *Paseo por Scipy : scipy.interpolate*



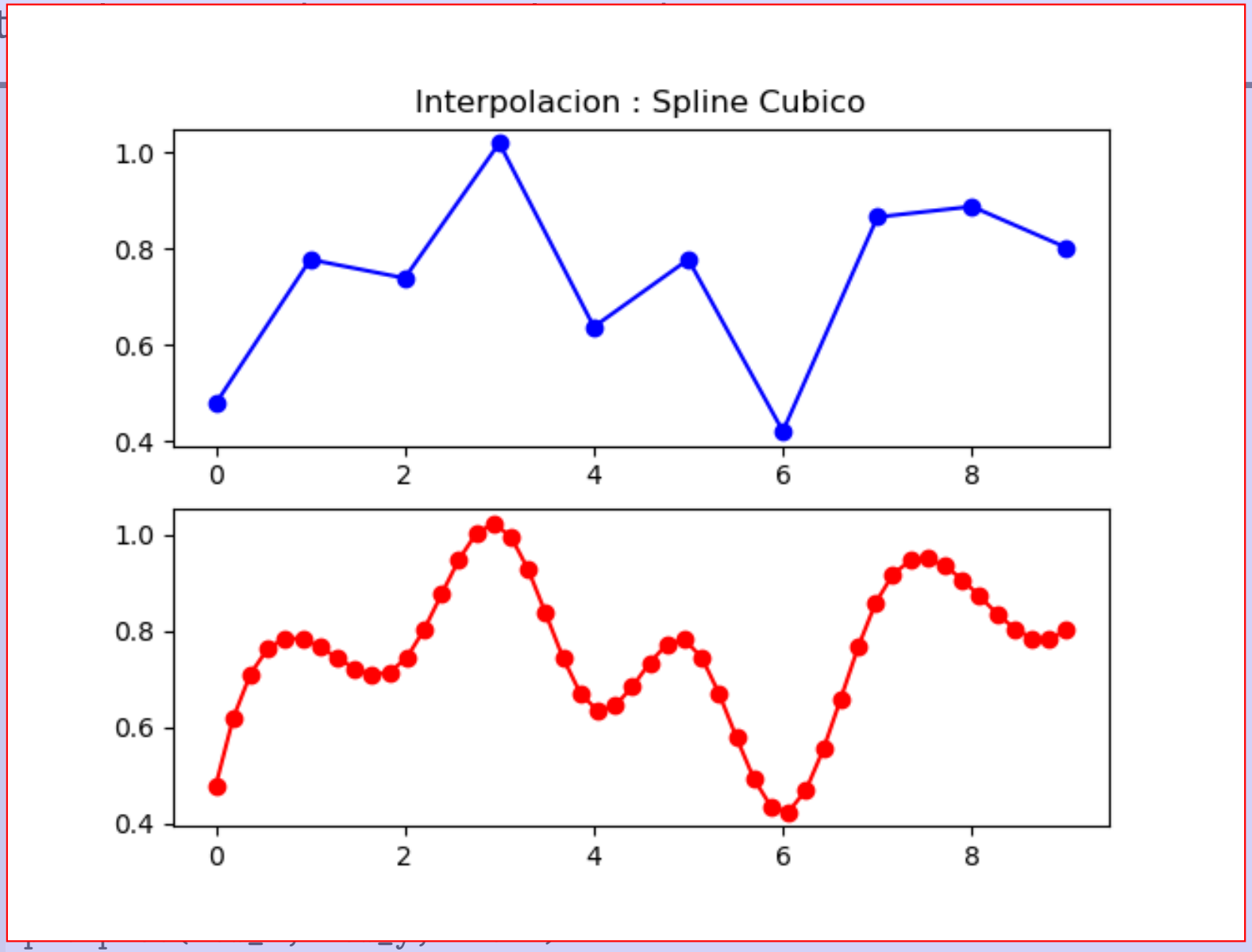

```
19 plt.legend()
20 plt.show()
```

# *Paseo por Scipy :* ***scipy.interpolate***

Interpolacion mediante un spline cubico :

```
1  import numpy as np
2  import scipy as sp
3  import matplotlib.pyplot as plt
4
5  # Generamos data aleatoria
6  y = (np.random.random(10) - 0.5).cumsum()
7  x = np.arange(y.size)
8
9  # Hacemos una interpolacion mediante un spline cubico
10 new_length = 50
11 new_x = np.linspace(x.min(), x.max(), new_length)
12 new_y = sp.interpolate.interp1d(x, y, kind='cubic')(new_x)
13
14 # graficamos los resultados
15 plt.figure()
16 plt.subplot(2,1,1)
17 plt.plot(x, y, 'bo-')
18 plt.title('Interpolacion : Spline Cubico')
19 plt.subplot(2,1,2)
20 plt.plot(new_x, new_y, 'ro-')
21 plt.show()
```

Int




```
21 plt.show()
```

# *Paseo por Scipy :* ***scipy.optimize***

El sub-modulo **scipy.optimize** proporciona una gama de algoritmos populares para minimización de funciones multidimensionales (con o sin restricciones adicionales), ajuste de datos por mínimos cuadrados y resolución de ecuaciones multidimensionales (búsqueda de raíces). En esta parte del curso se ofrecerá una descripción general de las opciones más importantes disponibles, pero debe tenerse en cuenta que la mejor elección de algoritmo dependerá de la función individual que se analice. Para una función arbitraria, no hay garantía de que un método particular converja en el mínimo deseado (o raíz, etc.), o que, si lo hace, converja rápidamente. Algunos algoritmos se adaptan mejor a ciertas funciones que otros, y cuanto más sepa sobre su función, mejor. **SciPy** puede configurarse para emitir un mensaje de advertencia cuando falla un algoritmo en particular, y este mensaje generalmente puede ayudar a analizar el problema.

Las rutinas de optimización de **SciPy** para minimizar una función de una o más variables $f(x_1, x_2, \ldots, x_n)$. La técnica para determinar el máximo, es determinar el mínimo de $-f(x_1, x_2, \ldots, x_n)$. Algunos de los algoritmos de minimización solo requieren que la función sea evaluada; otros requieren su primera derivada con respecto a cada una de las variables en una matriz conocida como el jacobiano:

$$j(f) = \left( \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial x_3}, \ldots, \frac{\partial f}{\partial x_n} \right) \tag{6}$$

Algunos algoritmos intentarán estimar numéricamente el jacobiano si no puede ser proporcionado como una función separada.

Además, algunos algoritmos de optimización sofisticados requieren información sobre las segundas derivadas de la función, una matriz simétrica de valores llamada Hessian:

$$(Hf)_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j} \tag{7}$$

Así como el jacobiano representa el gradiente local de una función de varias variables, el La hessiano representa la curvatura local.

El algoritmo general para la minimización de funciones escalares multivariadas es **scipy.optimize.minimize**, que toma dos argumentos obligatorios:

```
1 minimize(fun, x0, ...)
```

El primer argumento es la función a minimizar, el segundo es un punto que representa la estimación inicial para que el algoritmo de minimizacion inicie.

Como ejemplo estudiaremos la función de Himmelblau.Una función bidimensional simple con algunas características incómodas que la convierten en una buena función de prueba para algoritmos de optimización.Su regla de correspondencia es :

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2 \tag{8}$$

La region $[-5, 5]$ contiene un maximo local :
$f(-0.270845, -0.923039) = 181.617$ (aunque la función sube abruptamente fuera de esta región). Hay cuatro mínimos:

$$\begin{aligned} f(3, 2) &= 0 \\ f(-2.805118, 3.131312) &= 0 \\ f(-3.779310, -3.283186) &= 0 \\ f(3.584428, -1.848126) &= 0. \end{aligned}$$

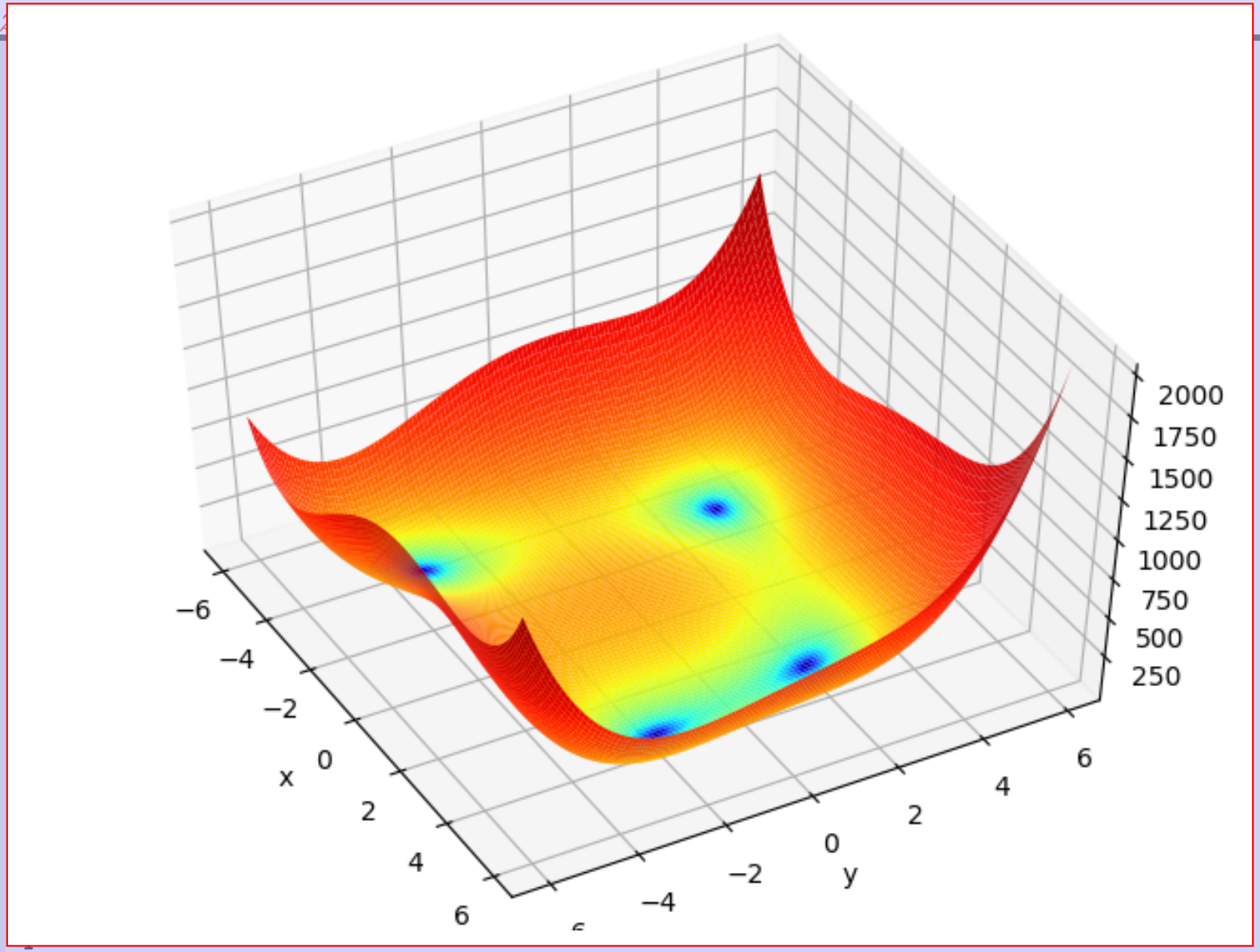
y cuatro puntos silla.

# *Paseo por Scipy :* ***scipy.optimize***

## *Minimización sin restricciones*

```
1  from mpl_toolkits.mplot3d import Axes3D
2  from matplotlib import cm
3  from matplotlib.colors import LogNorm
4  import matplotlib.pyplot as plt
5  import numpy as np
6
7  fig = plt.figure()
8  ax = Axes3D(fig, azim = -29, elev = 49)
9  X = np.arange(-6, 6, 0.1)
10 Y = np.arange(-6, 6, 0.1)
11 X, Y = np.meshgrid(X, Y)
12 Z = (X*X+Y-11)**2 + (X+Y*Y-7)**2
13 ax.plot_surface(X, Y, Z, rstride = 1, cstride = 1, norm =
   ↪  LogNorm(), cmap = cm.jet)
14
15 plt.xlabel("x")
16 plt.ylabel("y")
17
18 plt.savefig("Himmelblau function.png")
19
20 plt.show()
```

Minimi



# *Paseo por Scipy :* ***scipy.optimize***

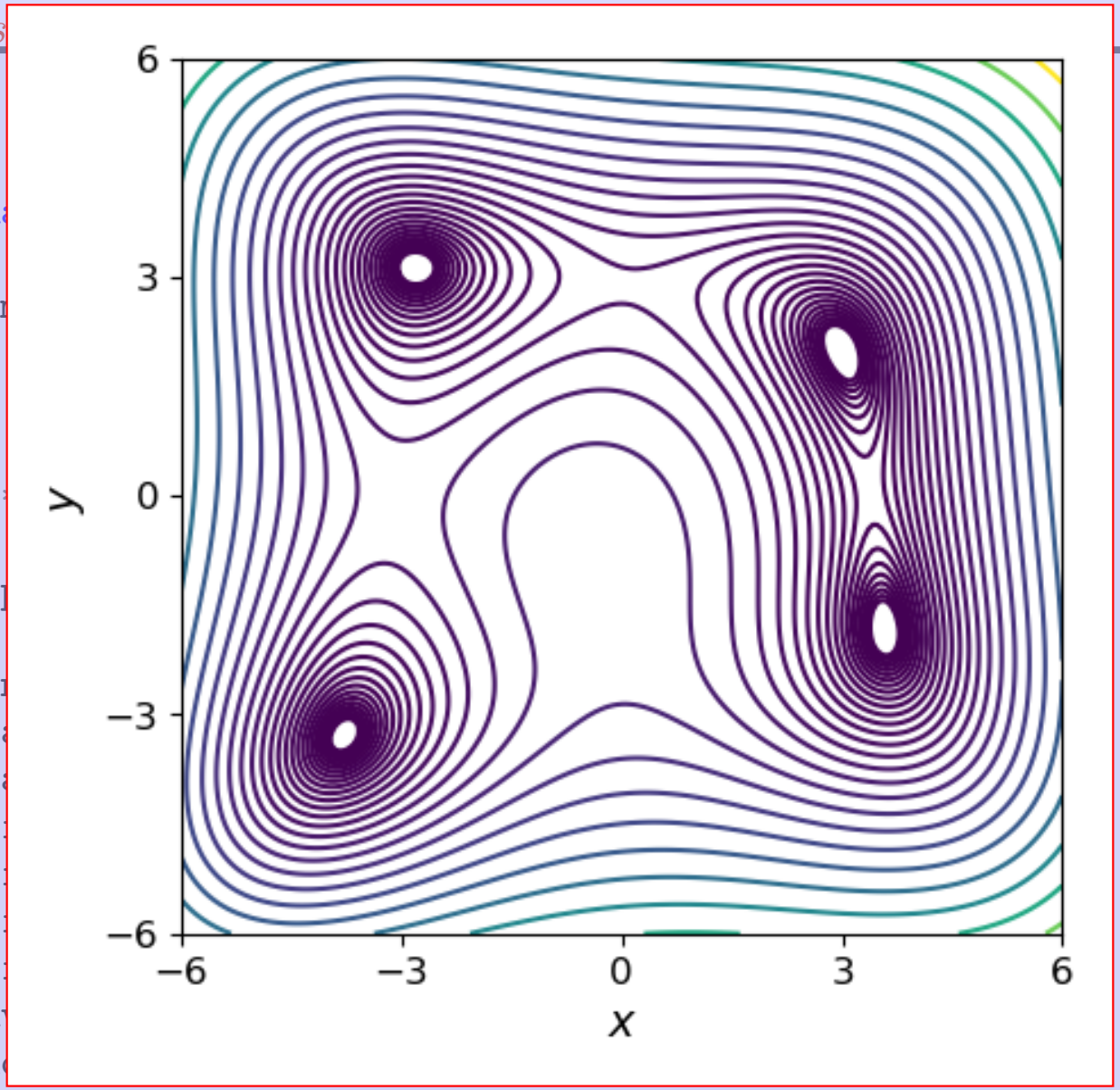
## *Minimización sin restricciones*

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from matplotlib import rcParams
4
5  rcParams['font.size'] = 12
6
7  npts = 201
8  x, y = np.mgrid[-6:6:npts*1j, -6:6:npts*1j]
9  z = (x**2 + y - 11)**2 + (x + y**2 - 7)**2
10
11 fig = plt.figure(figsize=(5, 5))
12 levels = np.logspace(0.3, 3.5, 15)
13 plt.contour(x, y, z, levels, cmap="viridis")
14 plt.xlabel(r"$x$", fontsize=14)
15 plt.ylabel(r"$y$", fontsize=14)
16 plt.xticks([-6, -3, 0, 3, 6])
17 plt.yticks([-6, -3, 0, 3, 6])
18 plt.xlim([-6, 6])
19 plt.ylim([-6, 6])
20 plt.savefig("Himmelblau_contour.png", bbox_inches="tight")
21 plt.show()
```

# *Paseo por Scipy : scipy.optimize*

*Minimización s*

```
1  import
2  import
3  from ma
4
5  rcPara
6
7  npts =
8  x, y =
9  z = (x
10
11 fig = p
12 levels
13 plt.co
14 plt.xla
15 plt.yla
16 plt.xt
17 plt.yt
18 plt.xl
19 plt.yl
20 plt.sa
21 plt.sh
```



# *Paseo por Scipy :* ***scipy.optimize***

## *Minimización sin restricciones*

La funcion se puede definir de manera usual :

```
1 def f(X):
2 x, y = X
3 return (x**2 + y - 11)**2 + (x + y**2 - 7)**2
```

Para buscar el minimo, llamamos a a la funcion **minimiza** con punto inicial $(0, 0)$

```
1 from scipy.optimize import minimize
2 minimize(f, (0,0))
```

La funcion **minimize** retorna un objeto parecido a un diccionario con informacion acerca del proceso de minimizacion que realizo.Si la minimizacion es exitosa, el minimo aparecera como `x` en este objeto.Podemos conluir que $f(3, 2) = 0$.

Obteniendo como salida de **minimize(f, (0,0))** :

```
fun: 1.3782261326630835e-13
hess_inv: array([[ 0.01578229, -0.0094806 ],
[-0.0094806 ,  0.03494937]])
jac: array([-3.95019832e-06, -1.19075540e-06])
message: 'Optimization terminated successfully.'
nfev: 64
nit: 10
njev: 16
status: 0
success: True
x: array([2.99999994, 1.99999999])
```

# *Paseo por Scipy :* ***scipy.optimize***

*Minimización sin restricciones*

| | |
|---|---|
| `success` | Un valor booleano que indica si la minimización fue o no exitosa |
| `x` | Si tiene exito , la solucion es el punto donde la funcion tiene un minimo. Si el algoritmo no tuvo exito, indica el punto donde dejo de iterar. |
| `fun` | Si tuvo exito, aca se almacena el valor minimo de la funcion. |
| `message` | Una cadena que describe el resultado del proceso de minimizacion |
| `jac` | El valor del jacobiano: si la minimización tiene éxito los valores en esta matriz debe estar cerca de cero. |
| `hess,hess_inv` | El hessiano y su inversa (si fue usada). |
| `nfev,njev,nhev` | El numero de evaluaciones de la funcion, del jacobiano y del hessiano. |

# *Paseo por Scipy :* ***scipy.optimize***

*Algunos algoritmos usado por minimize*

| Metodo | Descripcion |
|---|---|
| BFGS | El algoritmo Broyden-Fletcher-Goldfarb-Shanno (BFGS), es el que se usa por defecto para problemas sin restricciones (limites). |
| Nelder-Mead | Algoritmo de Nelder-Mead, también conocido como el simplex de descenso o método ameba. No se necesitan derivadas. |
| CG | Metodo de gradiente conjugado. |
| Powell | Metodo de Powell. |
| dogleg | Algoritmo de la region de confianza de la pata de perro (minimizacion sin restricciones) . Se requiere del jacobiano y hessiano (el cual debe ser definido positivo ). |
| TNC | Algoritmo de Newton truncado para minimización con cotas. |

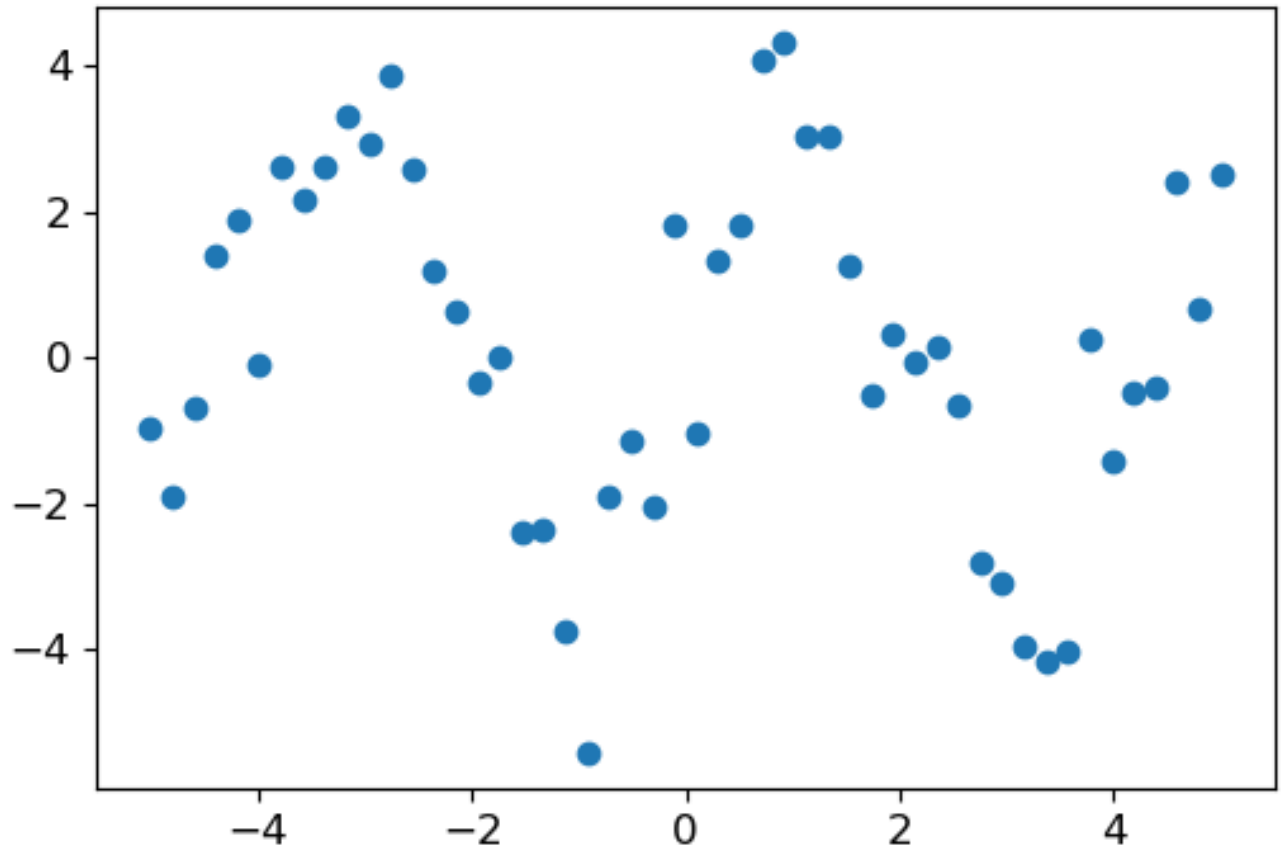
# *Paseo por Scipy :* ***scipy.optimize***

*Algunos algoritmos usado por minimize*

| Metodo | Descripcion |
|---|---|
| l-bfgs-b | Minimización restringida acotada con el algoritmo L-BFGS-B |
| slsqp | Método de "programación de mínimos cuadrados secuenciales" para minimizar con cotas y restricciones de igualdad y desigualdad. |
| cobyla | Método de “optimización restringida por aproximación lineal” para minimización con restricciones . |

# *Paseo por Scipy : **Ajuste de curvas (Fitting)***

```python
import numpy as np
from scipy import optimize

# Definimos una semilla
np.random.seed(0)

x_data = np.linspace(-5, 5, num=50)
y_data = 2.9 * np.sin(1.5 * x_data) + np.random.normal(size=50)

# Graficamos la data generada
import matplotlib.pyplot as plt
plt.figure(figsize=(6, 4))
plt.scatter(x_data, y_data)
```

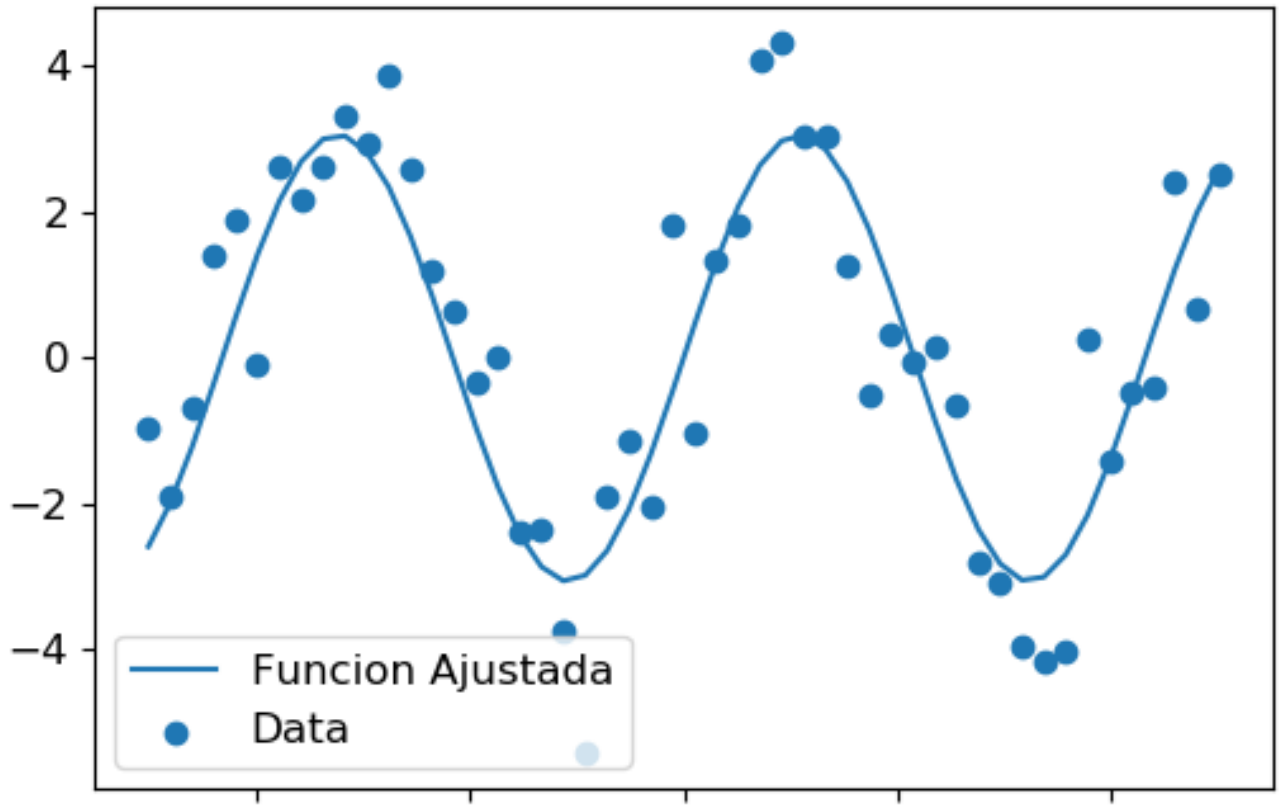

4
2
0
−2
−4
−4
−2
0
2
4

# *Paseo por Scipy : **Ajuste de curvas (Fitting)***

```
# definimos una funcion de ajuste (modelo)
def test_func(x, a, b):
return a * np.sin(b * x)

# scipy realiza el ajuste
params, params_covariance = optimize.curve_fit(test_func,
↪  x_data, y_data, p0=[2, 2])

# Imprimirmos los parametros a y b del modelo
print(params)
```

# *Paseo por Scipy : **Ajuste de curvas (Fitting)***

```
# Visualizamos los resultados.
plt.figure(figsize=(6, 4))
plt.scatter(x_data, y_data, label='Data')
plt.plot(x_data, test_func(x_data, params[0], params[1]),
label='Funcion Ajustada')

plt.legend(loc='best')

plt.show()
```


4
2
0
−2
−4
−4
−2
0
2
4
Funcion Ajustada
Data

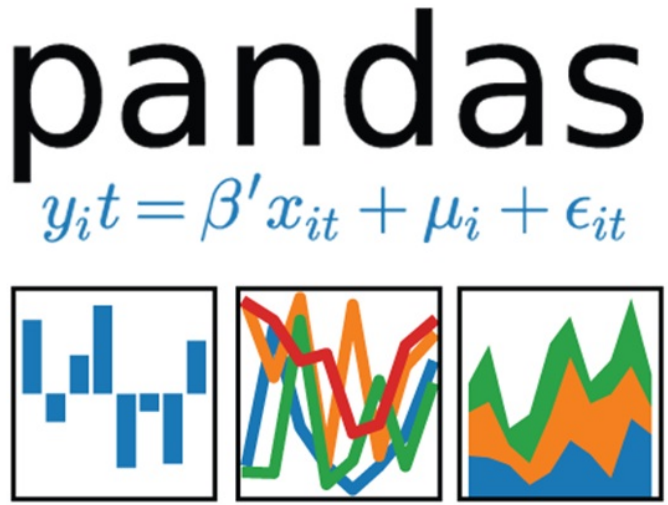
Numpy

Matplotlib

Scipy

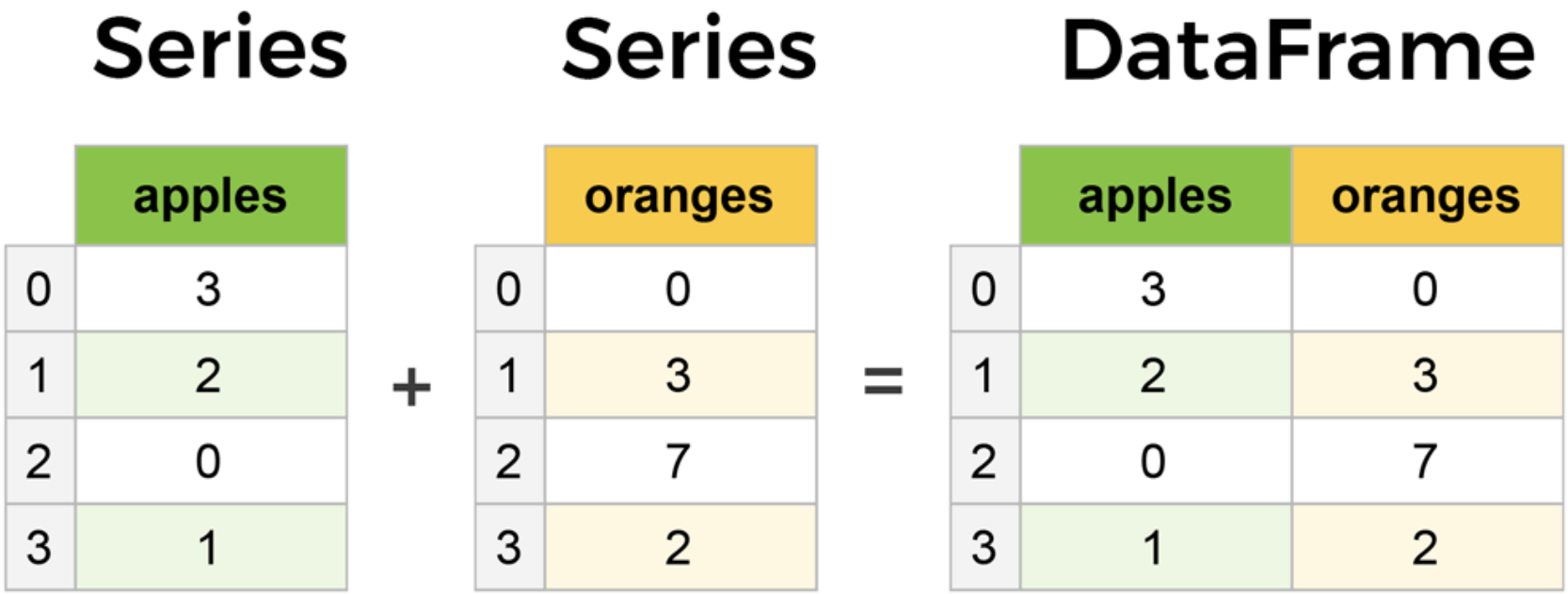
Pandas

# Pandas



# *Introducción a las estructuras de datos de Pandas*

Pandas es una biblioteca de Python de código abierto para el análisis de datos. Le da a Python la capacidad de trabajar con datos similares a una hoja de cálculo para cargar, manipular, alinear, fusionar, etc. datos de forma rápida y eficiente . Para darle a Python estas características mejoradas, Pandas presenta dos nuevos tipos de datos : Series y DataFrame.



# *Introducción a las estructuras de datos de Pandas*

Pandas introduce dos objetos clave en Python, el dato de tipo series y el dato de tipo DataFrames, siendo este último el más versátil y el que permite el desarrollo y analisis de una mayor numero de problemas, pero el dato de tipo *DataFrames* puede considerarse como la union de varios datos de de tipo series .

Una serie es una secuencia de datos, como una lista en Python o como una matriz unidimensional de NumPy. Y, al igual que el dato de tipo *ndarray*, una serie tiene un solo tipo de dato, pero la indexación con una serie es diferente.

Con NumPy no hay mucho control sobre los índices de fila y columna; pero con un dato de tipo series, cada elemento de la serie debe tener un índice, nombre y clave únicos, como quiera que lo piense.

Un DataFrame puede considerarse como varios objetos de tipo series, con un índice común, unidos en un solo objeto tabular. Este objeto se asemeja a un objeto bidimensional de tipo ndarray, pero no es lo mismo. No todas las columnas deben ser del mismo tipo de datos.

# *Introducción a las estructuras de datos de Pandas*

Para empezar con pandas es necesario que el punto inicial sea manejar las dos estructuras de pandas :

1. Series
2. DataFrame

Si bien no son una solución universal para cada problema, proporcionan una base solida y fácil de usar para la mayoría de las aplicaciones.

Un objeto `Series` es similar a una matriz unidimensional que contiene un vector de datos (numpy.ndarray) y un vector asociado de etiquetas para los datos, llamado `Index`. El ejemplo mas simple es el que se forma a partir de un vector de datos

```
1  from pandas import Series, DataFrame
2  import pandas as pd
3
4  obj = Series([4, 7, -5, 3])
5
6  obj.values
7
8  obj.index
```

A menudo será deseable crear un objeto de tipo `Series` con un índice que identifique cada punto de datos:

```
obj2 = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

obj2.index

obj2['a']

obj2['d'] = 6

obj2[['c', 'a', 'd']]
```

Las operaciones de matrices de tipo numpy, como el filtrado con una matriz booleana, multiplicacion por un escalar o al aplicar funciones matematicas, se preserva el enlace que existe entre el `Index:value`

```
obj2

obj2[obj2 > 0]

obj2 * 2

np.exp(obj2)
```

## *Series*

Otra forma de ver a los datos de tipo `Series` es como un diccionario ordenado de longitud fija, ya que es un mapeo de los valores por medio de los indices. Pueden sustituir en muchas funciones a datos de tipo diccionario.

```
'b' in obj2

'e' in obj2
```

Si se tiene datos almacenados en un dicccionario se puede crear un objeto de tipo `Series` a partir del diccionario:

```
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

obj3 = Series(sdata)

obj3
```

Cuando se pasa un diccionario, el indice del objeto `Series` resultante tendra a las claves del diccionario de manera ordenada

```
1  states = ['California', 'Ohio', 'Oregon', 'Texas']
2
3  obj4 = Series(sdata, index=states)
4
5  obj4
```

En este caso se encontraron 3 valores en `sdata` y se colocaron en las posiciones apropiadas, pero desde que no se encontreo informacion previa sobre **california**, el valor para esta llave es de `NaN` (NOt a number)que es como considera pandas a los valores faltantes.

Las funciones `isnull` y `notnull` se pueden usar para detectar valores faltantes.

```
pd.isnull(obj4)

pd.notnull(obj4)
```

Los datos de tipo `Series` también poseen estos métodos :

```
obj4.isnull()
```

Para las operaciones aritméticas, los datos de tipo `Series` se alinean de manera automática teniendo en cuenta los `index` para hacer las respectivas operaciones :

```
obj3 + obj4
```

Tanto el dato de tipo `Series` como su respectivo `index` tienen un atributo llamado `name` que se integra con otras áreas clave de pandas:

```
obj4.name = 'population'

obj4.index.name = 'state'
```

El `index` de un dato de tipo `Series` puede cambiarse por asignacion :

```
obj = Series([4, 7, -5, 3])

obj

obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
```

Un tipo de dato `DataFrame` representa una estructura tabular, tipo hoja de calculo que contiene un colección de columnas, cada una de las cuales puede ser un tipo de datos básico diferente (numérico,cadena,booleano,etc).El `DataFrame` tiene un indice para filas y columnas, puede ser pensado como un diccionario de datos de tipo `Series` (uno para todos compartiendo el mismo indice).En comparación con otros datos de este tipo (como los dataframe de R), las operaciones orientadas a filas y orientadas a columnas se tratan aproximadamente de forma simetrica.

Hay muchas formas para construir un `DataFrame` , aunque una de las mas comunes es la de un diccionario de listas de igual longitud o de un vector numpy

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

El `DataFrame` resultante tendrá su `index` (indice) asignado automáticamente como con los datos de tipo `Series` y las columnas se colocan de manera ordenada.

Si se especifica una secuencia para las columnas, esta secuencia es como se construira el `DataFrame`.

```
DataFrame(data, columns=['year', 'state', 'pop'])
```

Como con los datos de tipo `Series` si se pasa una columna de la que no se tenga informacion, esta aparecera con valores `NaN`

```
frame2 = DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
index=['one', 'two', 'three', 'four', 'five'])

frame2

frame2.columns
```

Una columna en un `DataFrame` se puede recuperar como un dato de tipo `Series`, ya sea por una notacion tipo diccionario o como un atributo

```
1  frame2['state']
2
3  frame2.year
```

Note que el dato de tipo `Series` que retorna tiene el mismo `index` que el `DataFrame` y su atributo `name` a sido configurado apropiadamente.

Las filas tambien se pueden recuperar por su posición o por su nombre mediante un par de métodos : `loc`

```
frame2.loc['three']
```

Las columnas se pueden modificar por asignación. Por ejemplo , la columna vacia `debt` del `DataFrame` se le podria asignar 7 un escalar o un vector.

```
frame2

frame2['debt'] = 16.5

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

## *DataFrame*

Al asignar listas o matrices a una columnas, la longitud del campo valor debe coincidir con la longitud del `DataFrame` . Si se asigna un dato de tipo `Series` en su lugar se ajustara exactamente al `index` del `DataFrame`, insertando `NaN`'s a los valores faltantes

```
1  val = Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
2
3
4  frame2['debt'] = val
5
6  frame2
```

Asignar una columna que no existe creara una nueva columna, la palabra reservada `del` eliminara columnas como lo hace con un diccionario.

```
1  frame2['eastern'] = frame2.state == 'Ohio'
```

## *DataFrame*

Otra forma comun de almacenar data es anidando diccionarios

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

pop['Nevada']

pop['Nevada'][2001]
```

Si se crea un `DataFrame` con el diccionario (de diccionarios) `pop` interpretara las claves (llaves o keys) del diccionario mas externo como las columnas y el diccionario mas interno como los indices de las filas .

```
frame3 = DataFrame(pop)

#porsupuesto, siempre se puede
#transponer el resultado
frame3.T
```

Las claves en los diccionarios internos están juntos y ordenados para formar el `index` del resultado. Esto no es cierto si se especifica un `index` explicito.

```
DataFrame(pop, index=[2001, 2002, 2003])
```

Los diccionarios de `Series` se tratan de la misma manera.

```
pdata = {'Ohio': frame3['Ohio'][:-1],'Nevada': frame3['Nevada'][:2]}

DataFrame(pdata)
```

Si el `Index` y las columnas de un `DataFrame` tienen seteado el atributo `name`, estos se mantendrán en el `DataFrame`.

```
frame3.index.name = 'year'; frame3.columns.name = 'state'

frame3
```

Al igual que con los datos de tipo `Series`, el atributos `values` retorna la data contenida en un numpy array 2D.

```
1  frame3.values
```

Si las columnas del `DataFrame` son de diferentes tipos, entonces el tipo de la matriz de valores sera `objeto`.

```
1  frame2.values
```

# *Procesamiento con Pandas*

## *DataFrame : ¿Quién se come los alimentos que cultivamos? (1961-2013)*

La hoja de datos de balance de alimentos (`FAO.csv`) esta relativamente completa.Algunos países que ya no existen , como checoslovaquia,se eliminaron de la base de datos. Se mantuvieron países que se formaron recientemente, como sudan del sur, a pesar de que no tienen los datos completos desde 1961.

```
1  # cargamos el archivo csv
2  data = pd.read_csv("FAO.csv",encoding ='latin-1')
3
4  # sacamos un poco de informacion de la data
5  print(type(data))
6  data.shape
7  data.ndim
8  data.head()
9  data.tail()
10 data.dtypes
11 data['Y2013'].describe()
12 data['Area'].describe()
13 data.describe()
```

**Seleccionando y manipulando :** Hay dos opciones principales para desarrollar actividades de selección e indexación en pandas.

1. Seleccionar la data en el eje de las filas (`iloc`).
2. Seleccionar data por etiquetas o por sentencias condicionales (`loc`)

El indexador `iloc` para un `DataFrame` se usa para indexación basada en la posición. La sintaxis de `iloc` es `data.olic(seleccion_Fila,Seleccion_Columna).Iloc` en pandas se usa para filas y columnas por numero, en el orden que aparecen en el `DataFrame`

# *Procesamiento con Pandas*

## *DataFrame : ¿Quién se come los alimentos que cultivamos? (1961-2013)*

```
# Seleccion simple en un dataframe
# Filas:
data.iloc[0] # Primera fila
data.iloc[1] # Segunda fila
data.iloc[-1] # Ultima fila del DataFrame
# Columnas:
data.iloc[:,0] # primera columna del DataFrame
data.iloc[:,1] # Segunda columna del DataFrame
data.iloc[:,-1] # ultima columna del DataFrame
```

**Se pueden seleccionar múltiples columnas y filas juntas usando el indexador `iloc`.**

```
# primeras cinco filas del DataFrame
data.iloc[0:5]
# primeras dos columnas del DataFrame con todas sus filas
data.iloc[:, 0:2]
# 1era, 4ta, 7ma, 25ava fila + 1era 6ta 7ma columnas.
data.iloc[[0,3,6,24], [0,5,6]]
# primeras cinco filas y 5ta, 6ta, 7ma columnas del DataFrame
data.iloc[0:5, 5:8]
```

Hay dos cosas a tener en cuenta al usar `iloc`:

1. Tenga en cuenta que `.iloc` devuelve un dato de tipo `Series` cuando se selecciona una fila, y un `DataFrame` cuando se seleccionan varias filas, o si se selecciona una columna en su totalidad.

```
1  print(type(data.iloc[-1]))
2  print(type(data.iloc[:,0]))
3  print(type(data.iloc[[0,3,6,24], [0,5,6]] ))
```

2. Al seleccionar varias filas y varias columnas de esta manera, recordar que en la selección que hacemos ,por ejemplo `[1:5]`, las filas / columnas seleccionadas se ejecutan desde el primer numero (1) hasta uno menos el segundo numero (5-1)

```
# cargar la data y visualizar el explorador de variables
dataUK = pd.read_csv('uk-500.csv')


dataUK.set_index("last_name", inplace=True)
# visualizar el explorador de variables
```

**Seleccionando data usando** `loc`: El indexador `loc` se puede usar con `DataFrames` en dos diferentes casos :

1. Seleccionando filas por etiqueta / indice .
2. Seleccionando filas con una búsqueda condicional (booleana).

La sintaxis del indexador `loc` tiene la misma sintaxis que `iloc` : `loc(Seleccion_Fila,Seleccion_Columna)`.

**Indexacion usando `loc` basado en etiquetas / indices** :Ahora con el conjunto de indices, podemos directamente seleccionar filas para diferentes valores de `last_name` usando `loc['etiqueta']`- ya sea uno solo o varios

```
dataUK.loc['Andrade']
dataUK.loc[['Andrade','Veness']]
```

Tambien podemos seleccionar algunas columnas y volver a cambiar el `set_index`

```
dataUK.loc[['Andrade', 'Veness'], 'city':'email']
dataUK.loc['Andrade':'Veness', ['first_name', 'address', 'city']]
# seleccionamos la fila con id = 487
data.loc[487]
```

**Indexado booleano/logico usando** `loc` : Las selecciones condiciones con matrices booleanas usando `loc[seleccion]` es el método mas común usado con este tipo de datos estructurados, ya que se parece mucho a una base de datos relacional.

```
1  dataUK.loc[dataUK['first_name'] == 'Antonio']
2
3  dataUK.loc[dataUK['first_name'] == 'Erasmo',['company_name','email','phone1' ]]
4
5  # devuelve un tipo de dato Series
6  dataUK.loc[dataUK['first_name'] == 'Antonio','email']
7
8  #  devuelve un tipo de dato DataFrame
9  dataUK.loc[dataUK['first_name'] == 'Antonio',['email']]
```

# *Procesamiento con Pandas*

## *DataFrame : Ciudadanos (UK)*

```
1   # selecciona filas con primer nombre Antonio
2   # y todas las columnas entre 'city' e 'email'
3   dataUK.loc[dataUK['first_name'] == 'Antonio', 'city':'email']
4
5
6   # selecciona filas donde la columna 'email' termina con 'hotmail.com'
7   # e incluye todas las columnas
8   dataUK.loc[dataUK['email'].str.endswith("hotmail.com")]
9
10
11  # selecciona filas con 'last_name' igual a algunos valores
12  # e incluye todas las columnas
13  dataUK.loc[dataUK['first_name'].isin(['France', 'Tyisha', 'Eric'])]
14
15  # selecciona filas con 'first_name' igual a Antonio y
16  # direcciones de correo en gmail
17  dataUK.loc[dataUK['email'].str.endswith("gmail.com") &
18  (dataUK['first_name'] == 'Antonio')]
```

# *Resumiendo información y haciendo estadística descriptiva con pandas*

```
df = DataFrame([[1.4, np.nan], [7.1, -4.5],
[np.nan, np.nan], [0.75, -1.3]],
index=['a', 'b', 'c', 'd'],
columns=['one', 'two'])

df.sum()

df.sum(axis=1)

df.mean(axis=1, skipna=False)

df.idxmax()

df.cumsum()

df.describe()
```

# *DataSet : Titanic*

## *Primeros pasos*

1. Importar las librerías a usar.
2. Cargar la base de datos.
3. Visualización simple de la data.

```
1  import pandas as pd
2  import os
3  os.chdir("Directorio_de_trabajo")
4  data = pd.read_csv("titanic.csv")
5  data.head()
```

# *DataSet : Titanic*

## *Primeros pasos*

Algunas propiedades del conjunto de datos.

```
print(data.shape)
print(data.count())
```

De los slides anteriores podemos notar que la columna **Cabin** tiene algunos valores **NaN**, estas son observaciones faltantes de la información.

# *DataSet : Titanic*

## *Primeros pasos*

Otra forma de contar los valores nulos, es contarlos por columna, ya que con la función **data.count** lo que hacemos es contar los valores no nulos.Esto sale de iterar sobre la lista de columnas y aplicando el a cada uno el método **isnull()** para luego obtener la suma con **sum()**

```
1  # Obtenemos los nombres delas columnas como una lista
2  col_names = data.columns.tolist()
3
4  # iteramos sobre la lista
5  for column in col_names:
6    print("Valores nulos en <{0}>: {1}".format(column,data[column].isnull().sum()))
```

# *DataSet : Titanic*

## *Primeros pasos*

Una operación común con colecciones de datos es la de estandarizarlos o darles algún formato en particular. Tomemos como ejemplo la columna **Sex**, que tiene como unicos valores *female* y *male*. Veamos como reemplazar estos valores por *F* y *M* .Para ello haremos uso de un diccionario para la *traducción* y de una función de tipo *lambda*.

```
1  # creamos un diccionario con los valores originales
2  #y con los valores a reemplazar
3  d = {"male":"M" , "female":"F"}
4
5  # Utilizamos una funcion lambda para hacer el reemplazo
6  data["Sex"] = data["Sex"].apply(lambda x:d[x])
7
8
9  # Podemos verificar el cambio
10 data["Sex"].head()
```

# *DataSet : Titanic*

## *Acceso a columnas y resumen estadistico*

Para acceder a las columnas hacemos uso de

`data["nombre_de_la_columna"]`

lo cual se vuelve tedioso cuando quieres hacer varias rondas de análisis, afortunadamente hay otra forma de acceder .

```
1  data.Age
2
3  # tambien podemos obtener informacion de los principales
4  # indicadores estadisticos sobre el dataset
5  data.describe()
```

# *DataSet : Titanic*

## *Algunos resultados primarios*

POdemos ver del resultado anterior que el minimo para la variables **Fare** es 0. *Esto quiere decir que hubo personas que viajaron gratis*

```
1  data[data.Fare ==0]
```

Tambien, haciendo uso de las agrupaciones o tablas de referencia cruzada podemos notar que el numero mayor de sobrevivientes fueron las mujeres.

```
1  pd.crosstab(data.Survived , data.Sex)
```

# *DataSet : Titanic*

*Algunos resultados primarios*

Agrupaciones por varias columnas :

¿Cuantos hombres y mujeres sobrevivieron por clase?

```
1   pclass_gender_survival_count_df = data.groupby(["Pclass" ,"Sex"])["Survived"].sum()
```

# *DataSet : Titanic*

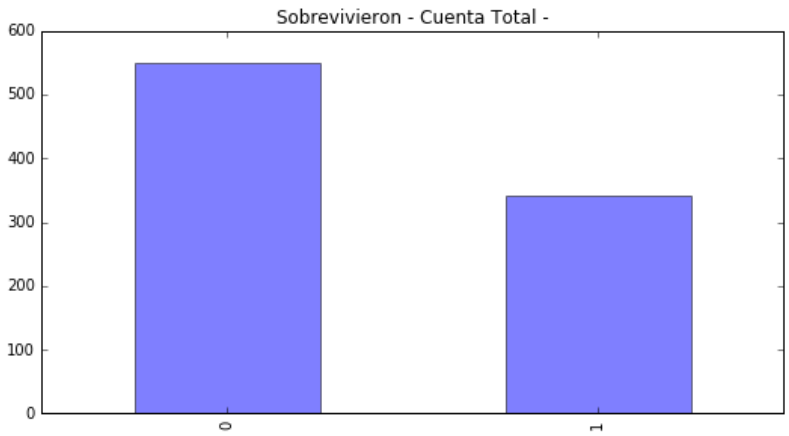
## *Aspectos básicos de visualización*

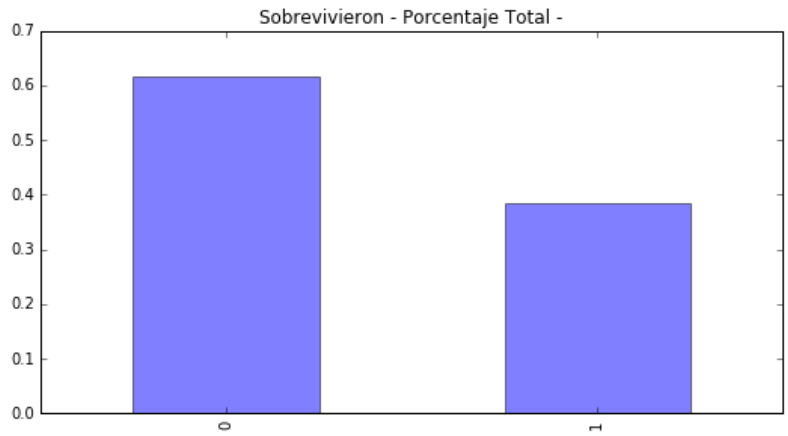
Para ello usamos **matplotlib**, lo primero es determinar el tamaño del gráfico y luego usar la función **subplot2grid** para poder tener los gráficos uno al lado del otro.

```
1  import matplotlib.pyplot as plt
2  # Creamos un figura de 30x10 pixeles
3  fig = plt.figure(figsize = (30,10))
4  # Deseamos un ventana (plot) al lado de la otra
5  # para esto pensamos en una grilla (celda)
6  plt.subplot2grid((2,3),(0,0))
7  data.Survived.value_counts().plot(kind = 'bar',alpha = 0.5)
8  plt.title("Sobrevivieron - Cuenta Total -")
9  # Usando porcentajes
10 plt.subplot2grid((2,3),(0,1))
11 data.Survived.value_counts(normalize = True).\
12 plot(kind = 'bar',alpha = 0.5)
13 plt.title("Sobrevivieron - Porcentaje Total -")
14 plt.savefig('TotPorcent',bbox_inches = 'tight')
15 plt.show()
```

# *DataSet : Titanic*

## *Aspectos básicos de visualización*




La gráfica de la izquierda muestra los sobrevivientes en numero mientras que la de la derecha los muestra en porcentajes . Menos del 40% del conjunto de datos sobrevivió.

# *DataSet : Titanic*

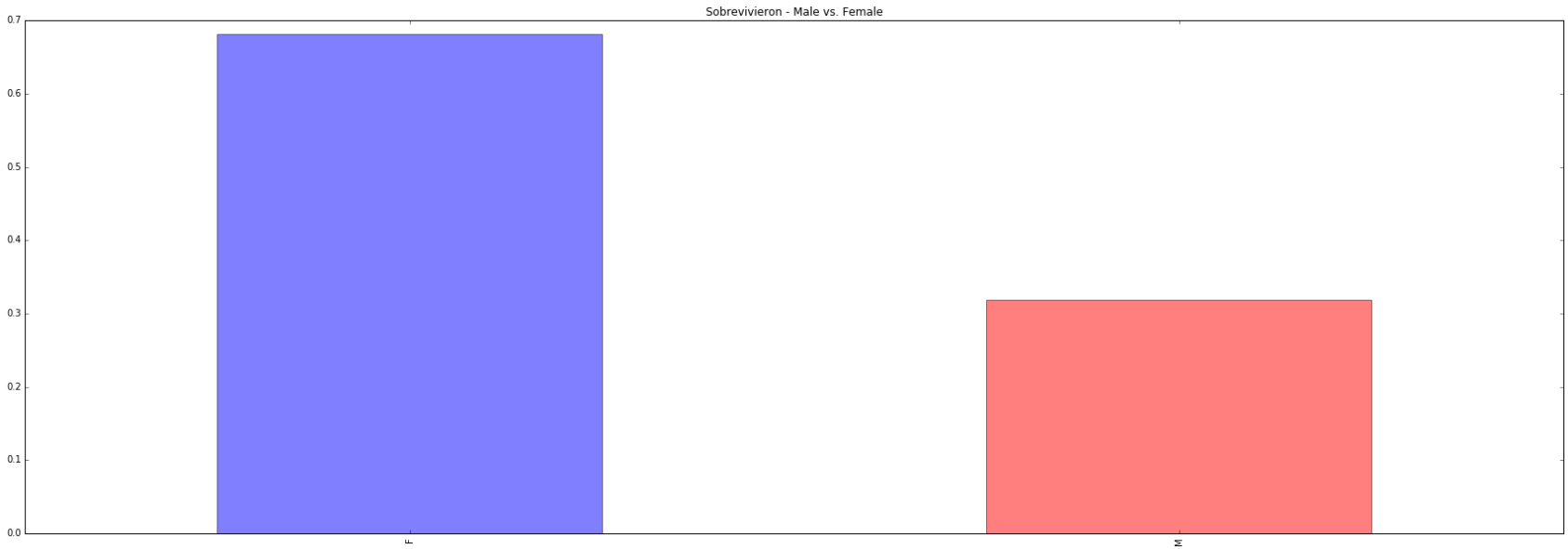
*Aspectos básicos de visualización*

## ¿Quienes sobrevivieron más, los hombres o las mujeres ?

```
1  import matplotlib.pyplot as plt
2
3  # Creamos un figura de 30x10 pixeles
4  fig = plt.figure(figsize = (30,10))
5  data.Sex[data.Survived==1].value_counts(normalize = True).plot(kind='bar',alpha=0.5)
6  plt.title("Sobrevivieron - Male vs. Female")
7  # plt.savefig('Sobrevivieron.png',bbox_inches = 'tight')
8  plt.show()
```

# *DataSet : Titanic*

## *Aspectos básicos de visualización*



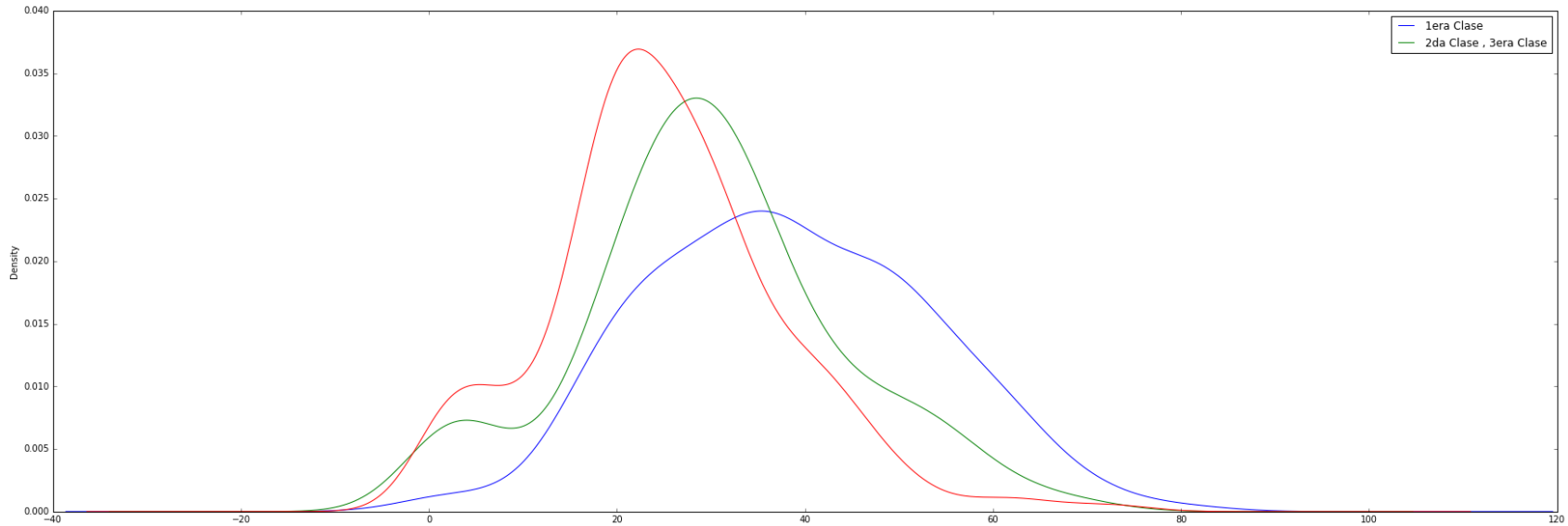

Podemos ver que la mayoría fueron mujeres .

Algo que se podría inferir en relación a edad-economía es que probablemente las personas mas jóvenes tenían menos dinero y por ende compraron los tickets mas baratos. Esto lo podremos ver con una gráfica de densidad, otro tipo (**kind**) de gráfica disponible en **matplotlib**.

```
1  import matplotlib.pyplot as plt
2
3  # Creamos un figura de 30x10 pixeles
4  fig = plt.figure(figsize = (30,10))
5  for t_class in [1,2,3]:
6          data.Age[data.Pclass==t_class].plot(kind='kde')
7  plt.legend(("1era Clase" , "2da Clase , 3era Clase"))
8  plt.savefig('densidad.png',bbox_inches = 'tight')
9  plt.show()
10 plt.close(fig)
```

# *DataSet : Titanic*

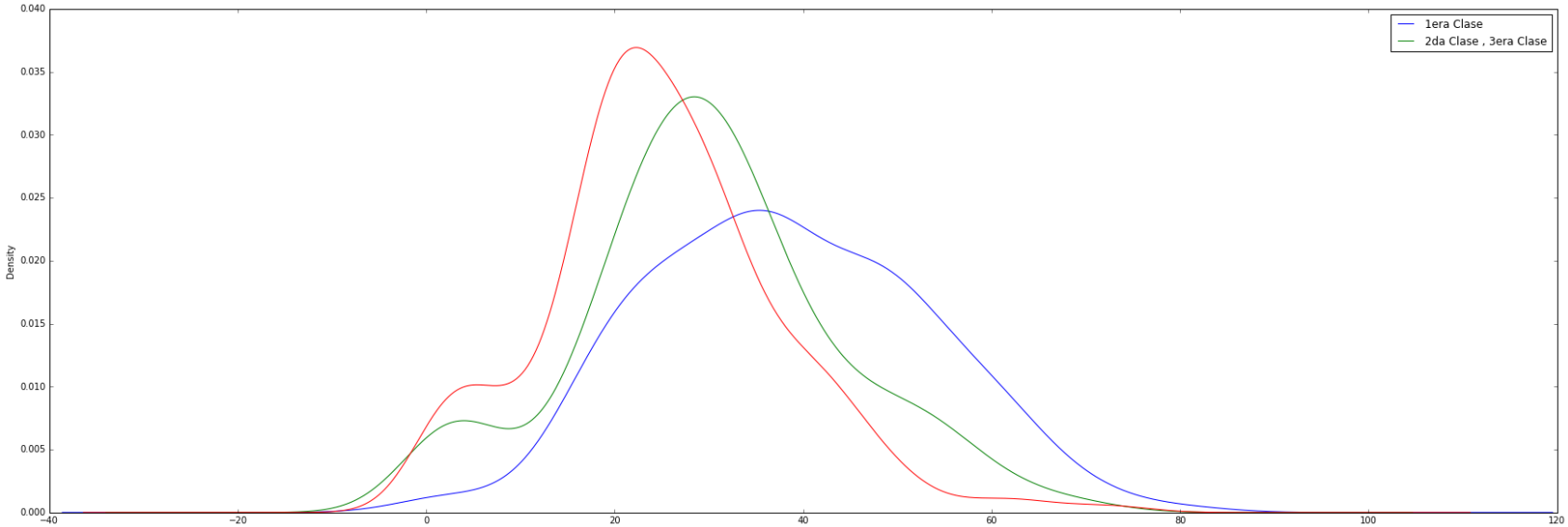
## *Aspectos básicos de visualización*




Si vemos la linea verde (3era Clase ) vemos que el promedio de edad es cerca de los 20 años y en primera clase el promedio de edad es 40, lo que muestra una relación entre edad-economía .

# *DataSet : Titanic*

## *Aspectos básicos de visualización*




Algo interesante de esta gráfica es notar que las lineas de edad no empiezan en cero.

```
1   data[data.Age <1]
```

Afortunadamente todos los bebes de nuestro conjunto de datos sobrevivieron, un posterior analisis seria averiguar si algunos (o sus dos padres) sobrevivieron

# *Análisis de series de tiempo con pandas*

Originalmente desarrollado para series de tiempo financieras, como los precios diarios del mercado de valores, las estructuras de datos robustas y flexibles de los pandas se pueden aplicar a datos de series de tiempo en cualquier dominio, incluidos negocios, ciencia, ingeniería, salud pública y muchos otros.Con estas herramientas, puede organizar, transformar, analizar y visualizar fácilmente sus datos en cualquier nivel de granularidad: examinar detalles durante períodos de tiempo específicos de interés y alejarse para explorar variaciones en diferentes escalas de tiempo, como agregaciones mensuales o anuales, patrones recurrentes y tendencias a largo plazo.

En la definición más amplia, una serie de tiempo es cualquier conjunto de datos donde los valores se miden en diferentes puntos en el tiempo.Muchas series temporales están uniformemente espaciadas a una frecuencia específica, por ejemplo, mediciones climáticas por hora, conteos diarios de visitas al sitio web o totales de ventas mensuales.Las series de tiempo también pueden estar espaciadas de manera irregular y esporádicas, por ejemplo, datos con marca de tiempo en el registro de eventos de un sistema informático o un historial de llamadas de emergencia al 911.

# *Análisis de series de tiempo con pandas*

Al trabajar con una serie temporal de datos de energía, veremos las técnicas como la indexación basada en el tiempo, el remuestreo y las ventanas móviles pueden ayudarnos a explorar las variaciones en la demanda de electricidad y el suministro de energía renovable a lo largo del tiempo. Cubriremos los siguientes temas:

- ➔ El dataset : Open Power Systems Data
- ➔ Estructuras de datos de series de tiempo
- ➔ Indexación basada en el tiempo
- ➔ Visualizar datos de series de tiempo
- ➔ Estacionalidad
- ➔ Frecuencia
- ➔ Resampleo
- ➔ Ventanas moviles
- ➔ Tendencias

# *Análisis de series de tiempo con pandas*

## *El data set: Open Power Systems Data*

En este parte, trabajaremos con series de tiempo diarias de datos del sistema de energía abierto (OPSD) de Alemania, que ha estado expandiendo rápidamente su producción de energía renovable en los últimos años. El conjunto de datos incluye el consumo total de electricidad en todo el país, la producción de energía eólica y la producción de energía solar para 2006-2017. El archivo de datos es `opsd_germany_daily.csv`.

La producción y el consumo de electricidad se reportan como totales diarios en gigavatios-hora (GWh). Las columnas del archivo de datos son:

- ➔ **Date** : La fecha es formato (yyyy-mm-dd)
- ➔ **Consumption** : Consumo de electricidad en Gwh
- ➔ **Wind** : Produccion de energia eolica en Gwh
- ➔ **Solar**: Produccion de energia solar en Gwh
- ➔ **Wind+Solar** : Suma de la producción de energía eólica y solar en GWh

Exploraremos cómo el consumo y la producción de electricidad en Alemania han variado con el tiempo, utilizando herramientas de series temporales de pandas para responder preguntas como:

- ➔ ¿Cuándo es el consumo de electricidad es típicamente más alto y más bajo?
- ➔ ¿Cómo varía la producción de energía eólica y solar con las estaciones del año?
- ➔ ¿Cuáles son las tendencias a largo plazo en el consumo de electricidad, energía solar y energía eólica?
- ➔ ¿Cómo se compara la producción de energía eólica y solar con el consumo de electricidad y cómo ha cambiado esta relación con el tiempo?

Antes de profundizar en los datos de OPSD, vamos a presentar brevemente las principales estructuras de datos de pandas para trabajar con fechas y horas. En pandas, un único punto en el tiempo se representa como un Timestamp. Podemos usar la función `to_datetime()` para crear marcas de tiempo a partir de cadenas en una amplia variedad de formatos de fecha / hora. Importemos pandas y convierta algunas fechas y horas en Timestamps.

```
1  import pandas as pd
2
3  pd.to_datetime('2018-01-15 3:45pm')
4
5  pd.to_datetime('7/8/1952')
```

Como podemos ver, `to_datetime()` infiere automáticamente un formato de fecha / hora basado en la entrada. En el ejemplo anterior, se supone que la fecha ambigua '7/8/1952' es mes / día / año y se interpreta como el 8 de julio de 1952. Alternativamente, podemos usar el parámetro `dayfirst` para indicar a pandas que interpreten la fecha como agosto 7 de 1952.

```
1  pd.to_datetime('7/8/1952', dayfirst=True)
```

Si proporcionamos una lista o array de cadenas como entrada a `to_datetime()`, esta devuelve una secuencia de valores de fecha / hora en un objeto DatetimeIndex, que es la estructura de datos central que potencia gran parte de la funcionalidad de las series temporales en pandas.

```
pd.to_datetime(['2018-01-05', '7/8/1952', 'Oct 10, 1995'])

pd.DatetimeIndex(['2018-01-05', '1952-07-08', '1995-10-10'],
dtype='datetime64[ns]', freq=None)
```

En el `DatetimeIndex` anterior, el tipo de datos `datetime64[ns]` indica que los datos subyacentes se almacenan como enteros de 64 bits, en unidades de nanosegundos(ns).Esta estructura de datos permite que pandas almacene de manera compacta grandes secuencias de valores de fecha / hora y realice eficientemente operaciones vectorizadas utilizando matrices NumPy (datetime64).

Si tratamos con una secuencia de cadenas todas en el mismo formato de fecha / hora, podemos especificarla explícitamente con el parámetro de formato.Para conjuntos de datos muy grandes, esto puede acelerar enormemente el rendimiento de `to_datetime()` en comparación con el comportamiento predeterminado, donde el formato se infiere por separado para cada cadena individual.Se puede usar cualquiera de los formatos de codigo de las funciones `strftime()` y `strptime()` en el módulo `datetime` incorporado en Python. El siguiente ejemplo utiliza los formatos de codigo `%m` (mes numérico), `%d` (día del mes) y `%y` (año de 2 dígitos) para especificar el formato.

```
1  pd.to_datetime(['2/25/10', '8/6/17', '12/15/12'],
2  format='%m/%d/%y')
3
4  pd.DatetimeIndex(['2010-02-25', '2017-08-06', '2012-12-15'],
5  dtype='datetime64[ns]', freq=None)
```

Además de los objetos `Timestamp` y `DatetimeIndex` que representan puntos individuales en el tiempo, pandas también incluye estructuras de datos que representan duraciones (por ejemplo, 125 segundos) y períodos (por ejemplo, el mes de noviembre de 2018). Por ahora solo usaremos `DatetimeIndexes`, la estructura de datos más común para series temporales de pandas.

Para trabajar con datos de series temporales en pandas, utilizamos un `DatetimeIndex` como índice para nuestro DataFrame (o Series). Veamos cómo hacer esto con nuestro conjunto de datos `opsd_germany_daily.csv`. Primero, usamos la función `read_csv()` para leer los datos en un `DataFrame` y luego mostrar su método `shape`.

```
1  opsd_daily = pd.read_csv('opsd_germany_daily.csv')
2  opsd_daily.shape
```

El `DataFrame` tiene 4383 filas, que cubren el período comprendido entre el 1 de enero de 2006 y el 31 de diciembre de 2017.

Para ver cómo se ven los datos, usemos los métodos `head()` y `tail()` para mostrar las primeras filas y las ultimas filas.

```
opsd_daily.head(3)

opsd_daily.tail(3)
```

A continuación, veamos los tipos de datos de cada columna.

```
opsd_daily.dtypes
```

Ahora que la columna `Date` es el tipo de datos correcto, configurémoslo como el índice del `dataframe`.

```
opsd_daily = opsd_daily.set_index('Date')
opsd_daily.head(3)

opsd_daily.index
```

Alternativamente, podemos consolidar los pasos anteriores en una sola línea, utilizando los parámetros `index_col` y `parse_dates` de la función `read_csv()`. Este suele ser un atajo útil.

```
1  opsd_daily = pd.read_csv('opsd_germany_daily.csv',
2  index_col=0, parse_dates=True)
```

Ahora que nuestro índice de `DataFrame` es un `DatetimeIndex`, podemos usar todas las poderosas indexaciones basadas en el tiempo de pandas para discutir y analizar nuestros datos.

Otro aspecto útil de `DatetimeIndex` es que los componentes individuales de fecha / hora están disponibles como atributos `year`, `month`, `day`, etc. Agreguemos algunas columnas más a `opsd_daily`, que contienen el año, el mes y el nombre del día de la semana.

```
1  # Agregamos las columnas year, month, y weekday name
2  opsd_daily['Year'] = opsd_daily.index.year
3  opsd_daily['Month'] = opsd_daily.index.month
4  opsd_daily['Weekday Name'] = opsd_daily.index.weekday_name
5  # mostramos una muestra aleatoria de 5 filas
6  opsd_daily.sample(5, random_state=0)
```

Una de las características más potentes y convenientes de las series temporales de pandas es la indexación basada en el tiempo, que utiliza fechas y horas para organizar y acceder de manera intuitiva a nuestros datos.

Con la indexación basada en el tiempo, podemos usar cadenas con formato de fecha / hora para seleccionar datos en nuestro `DataFrame` con el método `loc` . La indexación funciona de manera similar a la indexación estándar, pero con algunas características adicionales.

Por ejemplo, podemos seleccionar datos para un solo día usando una cadena como '2017-08-10'.

```
opsd_daily.loc['2017-08-10']
```

```
Consumption      1351.49
Wind              100.274
Solar               71.16
Wind+Solar         171.434
Year                  2017
Month                    8
Weekday Name      Thursday
Name: 2017-08-10 00:00:00, dtype: object
```

## *Indexado basado en el tiempo*

También podemos seleccionar una porción de días, como '2014-01-20':'2014-01-22'. Al igual que con la indexación regular, el segmento incluye ambos puntos finales.

```
opsd_daily.loc['2014-01-20':'2014-01-22']
```

Otra característica muy útil de las series temporales de pandas es la indexación de cadenas parciales, donde podemos seleccionar todas las fechas / horas que coinciden parcialmente con una cadena dada. Por ejemplo, podemos seleccionar todo el año 2006 con `opsd_daily.loc['2006']`, o todo el mes de febrero de 2012 con `opsd_daily.loc['2012-02']`.

```
opsd_daily.loc['2012-02']
```

## *Visualizar datos de tipo serie de tiempo*

Con **pandas** y **matplotlib**, podemos visualizar fácilmente nuestros datos de series de tiempo. En los siguientes slides, cubriremos algunos ejemplos y algunas personalizaciones útiles para nuestros gráficos de series de tiempo.

Primero, importemos **matplotlib**.

```
1  import matplotlib.pyplot as plt
```

Utilizaremos el estilo **seaborn** para nuestros plots, y ajustemos el tamaño de figura a una forma adecuada para las series temporales.

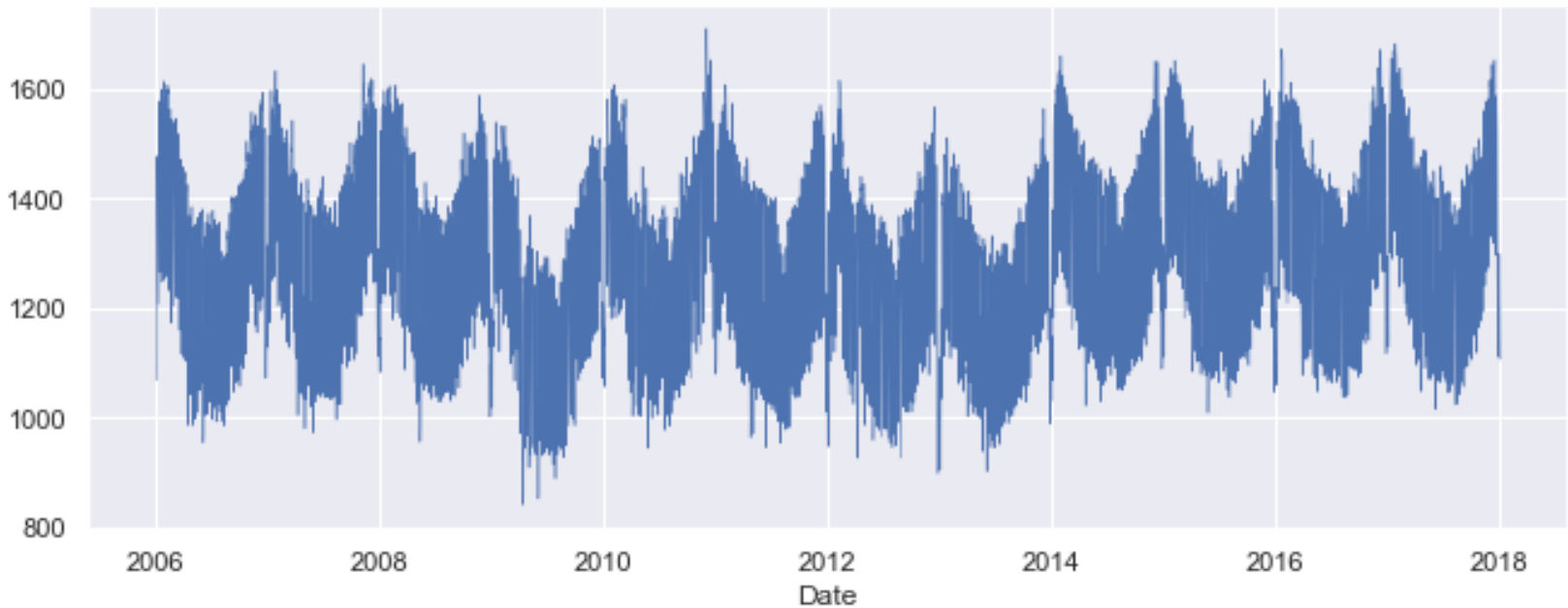
```
1  import seaborn as sns
2  # Usamos seaborn para establecer el tamano de la figura
3  sns.set(rc={'figure.figsize':(11, 4)})
```

# *Análisis de series de tiempo con pandas*

## *Visualizar datos de tipo serie de tiempo*

Creemos un diagrama de la serie de tiempo completa del consumo diario de electricidad de Alemania, utilizando el método `plot()` del `DataFrame`.

```
1  opsd_daily['Consumption'].plot(linewidth=0.5);
```

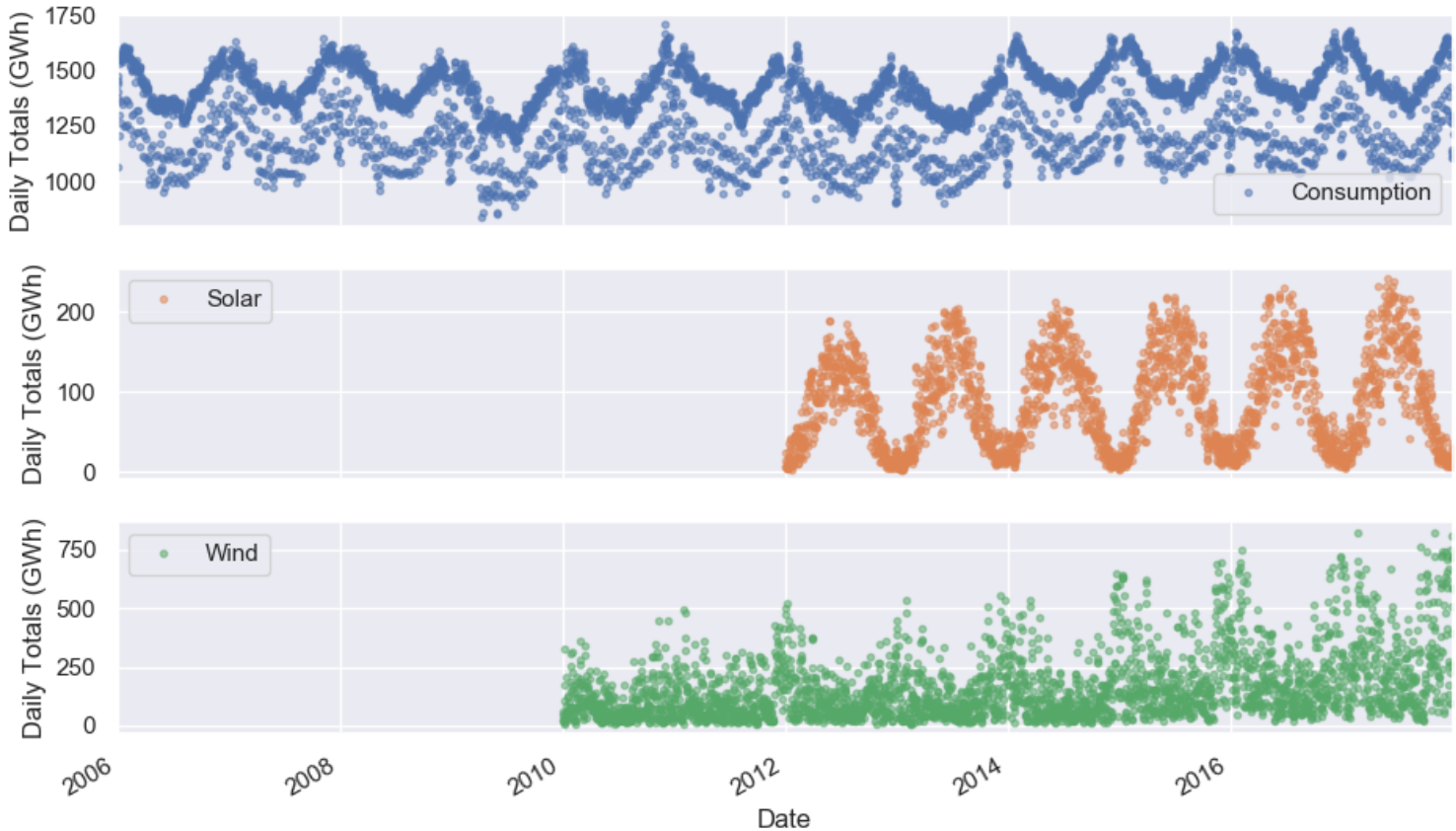


Podemos ver que el método `plot()` ha elegido ubicaciones de ticks bastante buenas (cada dos años) y etiquetas (los años) para el eje x, lo cual es útil. Sin embargo, con tantos puntos, el diagrama anterior está abarrotado y es difícil de leer. Tracemos los datos como puntos, y también veamos las series de tiempo `Solar` y `Wind`.

```
1  cols_plot = ['Consumption', 'Solar', 'Wind']
2  axes = opsd_daily[cols_plot].plot(marker='.', alpha=0.5,
3  linestyle='None', figsize=(11, 9), subplots=True)
4  for ax in axes:
5          ax.set_ylabel('Daily Totals (GWh)')
```

# *Análisis de series de tiempo con pandas*

## *Visualizar datos de tipo serie de tiempo*



Ya podemos ver surgir algunos patrones interesantes:

1. El consumo de electricidad es más alto en invierno, presumiblemente debido a la calefacción eléctrica y al mayor uso de la iluminación, y más bajo en verano.
2. El consumo de electricidad parece dividirse en dos grupos: uno con oscilaciones centradas aproximadamente a 1400 GWh y otro con menos y más puntos de datos dispersos, centrados aproximadamente a 1150 GWh. Podríamos adivinar que estos grupos se corresponden con los días de semana y fines de semana, e investigaremos esto en breve.

3. La producción de energía solar es más alta en verano, cuando la luz solar es más abundante y más baja en invierno.
4. La producción de energía eólica es más alta en invierno, presumiblemente debido a vientos más fuertes y tormentas más frecuentes, y más baja en verano.
5. Parece haber una fuerte tendencia creciente en la producción de energía eólica a lo largo de los años.

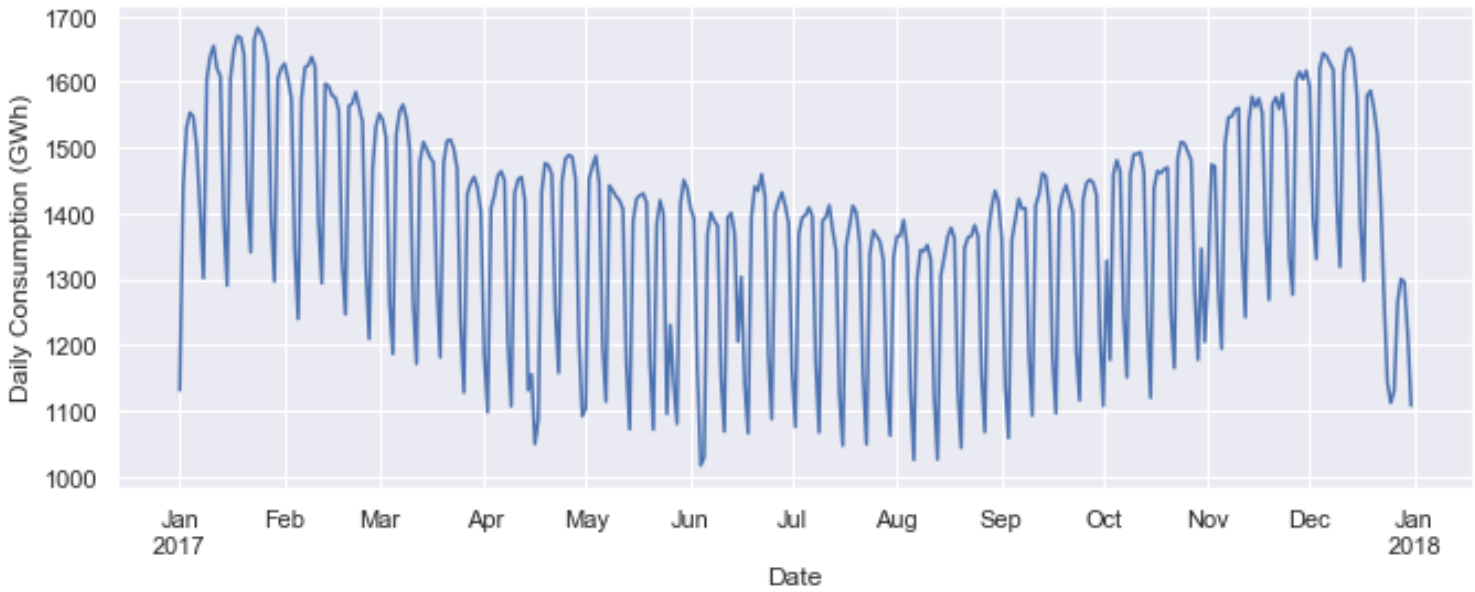
Las tres series temporales exhiben claramente la periodicidad, a menudo denominada estacionalidad en el análisis de series temporales, en el que un patrón se repite una y otra vez a intervalos de tiempo regulares. Las series de tiempo de consumo, solar y viento oscilan entre valores altos y bajos en una escala de tiempo anual, correspondiente a los cambios estacionales en el clima durante el año. Sin embargo, la estacionalidad en general no tiene que corresponder con las estaciones meteorológicas. Por ejemplo, los datos de ventas minoristas a menudo exhiben una estacionalidad anual con mayores ventas en noviembre y diciembre, antes de las vacaciones.

La estacionalidad también puede ocurrir en otras escalas de tiempo. El gráfico anterior sugiere que puede haber cierta estacionalidad semanal en el consumo de electricidad de Alemania, correspondiente a los días de semana y fines de semana. Tracemos la serie temporal en un solo año para investigar más a fondo.

```
1  ax = opsd_daily.loc['2017', 'Consumption'].plot()
2  ax.set_ylabel('Daily Consumption (GWh)');
```

# *Análisis de series de tiempo con pandas*

## *Visualizar datos de tipo serie de tiempo*



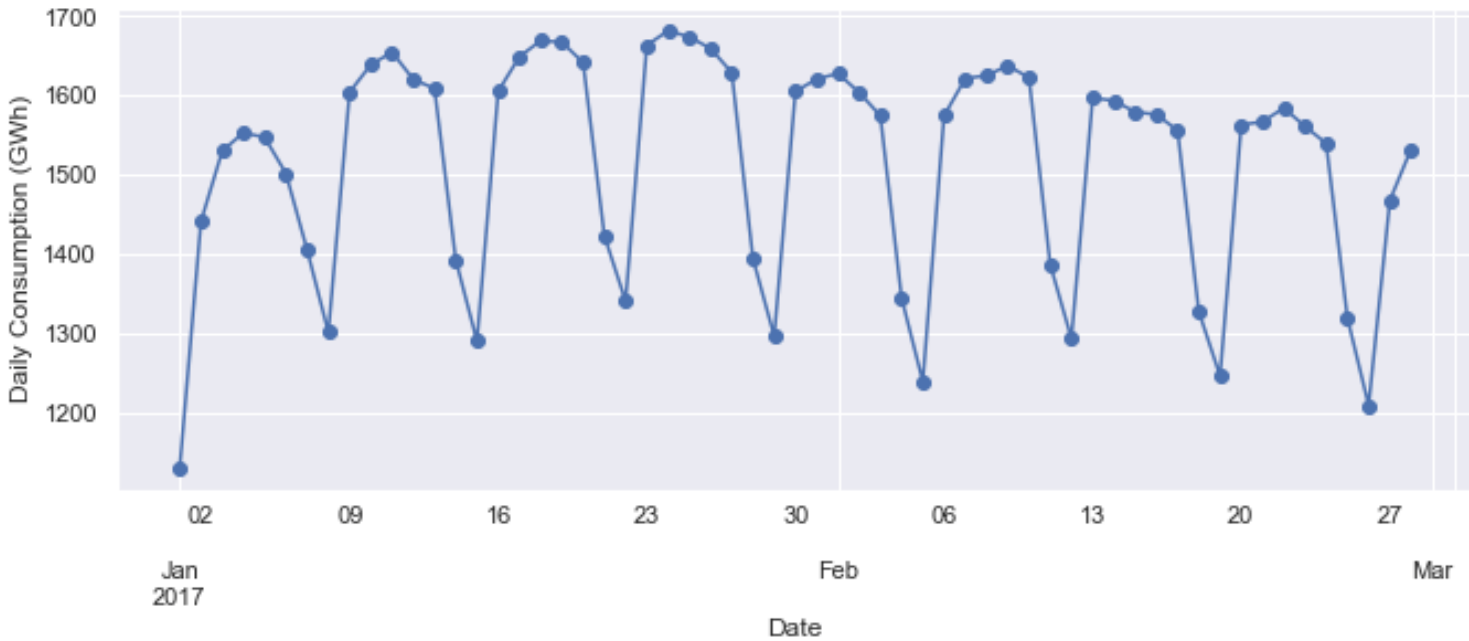

Ahora podemos ver claramente las oscilaciones semanales. Otra característica interesante que se hace evidente en este nivel de granularidad es la disminución drástica del consumo de electricidad a principios de enero y finales de diciembre, durante las vacaciones.

Acerquémonos más y veamos solo enero y febrero.

```
ax = opsd_daily.loc['2017-01':'2017-02', 'Consumption'].
plot(marker='o', linestyle='-')
ax.set_ylabel('Daily Consumption (GWh)');
```

# *Análisis de series de tiempo con pandas*

## *Visualizar datos de tipo serie de tiempo*




Como sospechábamos, el consumo es más alto entre semana y más bajo los fines de semana.

Para visualizar mejor la estacionalidad semanal en el consumo de electricidad en la gráfica anterior, sería bueno tener líneas verticales en una escala de tiempo semanal (en lugar del primer día de cada mes). Podemos personalizar nuestro diagrama con `matplotlib.dates`, así que importaremos ese módulo.

```
1  import matplotlib.dates as mdates
```

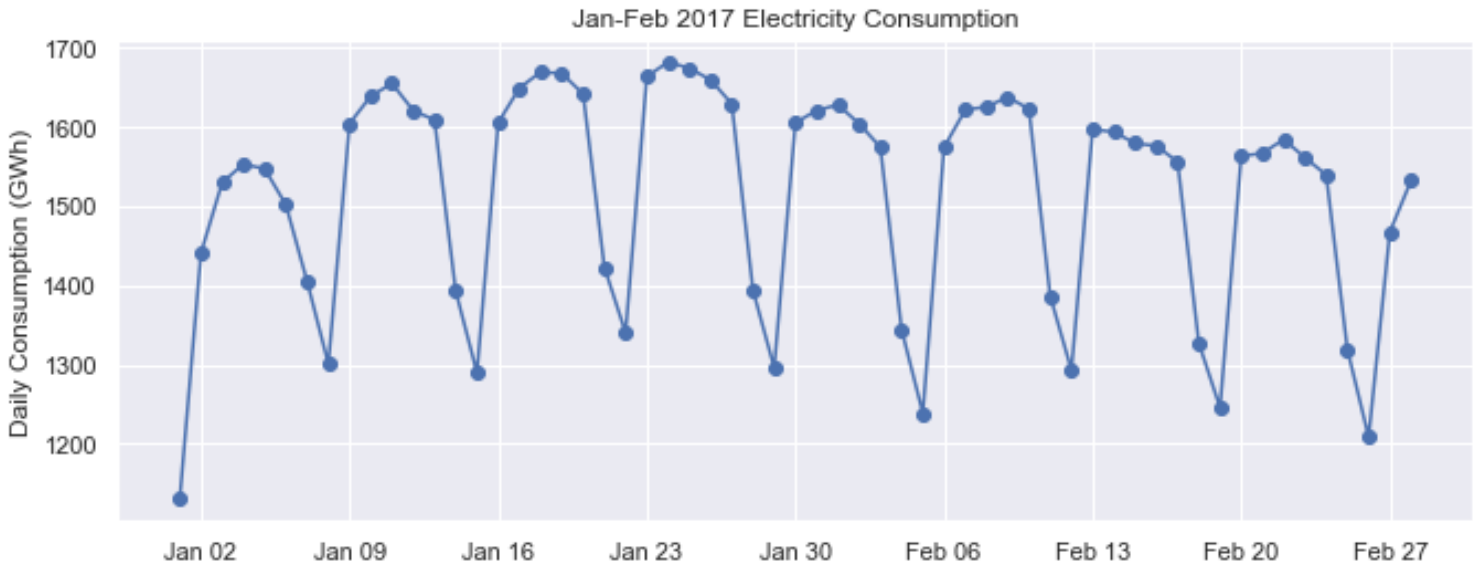
## *Personalizar diagramas de series de tiempo*

Debido a que los ticks de fecha / hora se manejan de manera un poco diferente en `matplotlib.dates` en comparación con el método `plot()` de DataFrame, creemos el gráfico directamente en matplotlib. Luego usamos `mdates.WeekdayLocator()` y `mdates.MONDAY` para establecer los ticks del eje x en el primer lunes de cada semana. También usamos `mdates.DateFormatter()` para mejorar el formato de las etiquetas, utilizando los códigos de formato que vimos anteriormente.

```
fig, ax = plt.subplots()
ax.plot(opsd_daily.loc['2017-01':'2017-02', 'Consumption'],
marker='o', linestyle='-')
ax.set_ylabel('Consumo diario (GWh)')
ax.set_title('Ene-Feb 2017 Consumo de electricidad')
# ticks principales del eje x en intervalos semanales:lunes
ax.xaxis.set_major_locator(mdates.WeekdayLocator(
byweekday=mdates.MONDAY))
# Las etiquetas de marca del eje-X como nombre de mes
# de 3 letras y numero de dia
ax.xaxis.set_major_formatter(mdates.DateFormatter('%b %d'));
```

# *Análisis de series de tiempo con pandas*

## *Personalizar diagramas de series de tiempo*




Ahora tenemos líneas de cuadrícula verticales y etiquetas de marca con un formato agradable cada lunes, por lo que podemos saber fácilmente qué días son entre semana y fines de semana.

En los siguientes slides, exploremos más en profundidad la estacionalidad de nuestros datos con diagramas de caja, utilizando la función `boxplot()` de `seaborn` para agrupar los datos por diferentes períodos de tiempo y mostrar las distribuciones para cada periodo . Primero agruparemos los datos por mes, para visualizar la estacionalidad anual.

```
1  fig, axes = plt.subplots(3, 1, figsize=(11, 10), sharex=True)
2  for name, ax in zip(['Consumption', 'Solar', 'Wind'], axes):
3          sns.boxplot(data=opsd_daily, x='Month', y=name, ax=ax)
4          ax.set_ylabel('GWh')
5          ax.set_title(name)
6  # Elimine la etiqueta automatica del eje x de todos
7  # menos del subplot inferior
8  if ax != axes[-1]:
9          ax.set_xlabel('')
```

Estos box plots confirman la estacionalidad anual que vimos en plots anteriores y proporcionan algunas ideas adicionales:

1. Aunque el consumo de electricidad es generalmente mayor en invierno y menor en verano, la mediana y los dos cuartiles inferiores son menores en diciembre y enero en comparación con noviembre y febrero, probablemente debido a que las empresas cierran durante las vacaciones. Vimos esto en la serie temporal para el año 2017, y el diagrama de caja confirma que este es un patrón consistente a lo largo de los años.
2. Si bien la producción de energía solar y eólica exhibe una estacionalidad anual, las distribuciones de energía eólica tienen muchos más valores atípicos, lo que refleja los efectos de las velocidades extremas ocasionales del viento asociadas con tormentas y otras condiciones climáticas transitorias.

A continuación, agrupemos las series temporales de consumo de electricidad por día de la semana, para explorar la estacionalidad semanal.

```
sns.boxplot(data=opsd_daily, x='Weekday Name', y='Consumption');
```

Como se esperaba, el consumo de electricidad es significativamente mayor entre semana que los fines de semana. Los valores atípicos bajos entre semana son presumiblemente durante las vacaciones.

## *Estacionalidad*

Esta sección ha proporcionado una breve introducción a la estacionalidad de series temporales. Como veremos más adelante, la aplicación de una ventana móvil a los datos también puede ayudar a visualizar la estacionalidad en diferentes escalas de tiempo. Otras técnicas para analizar la estacionalidad incluyen gráficos de autocorrelación, que trazan los coeficientes de correlación de las series de tiempo consigo mismo en diferentes lags de tiempo.

Las series temporales con una fuerte estacionalidad a menudo se pueden representar bien con modelos que descomponen la señal en estacionalidad y una tendencia a largo plazo, y estos modelos se pueden usar para pronosticar valores futuros de la serie temporal.

Un ejemplo más sofisticado es el modelo Prophet de Facebook, que utiliza el ajuste de curvas para descomponer las series de tiempo, teniendo en cuenta la estacionalidad en múltiples escalas de tiempo, efectos de vacaciones, puntos de cambio abruptos y tendencias a largo plazo.

Cuando los puntos de datos de una serie temporal están espaciados uniformemente en el tiempo (por ejemplo, por hora, diario, mensual, etc.) , la serie temporal puede asociarse con una frecuencia en pandas. Por ejemplo, usemos la función `date_range()` para crear una secuencia de fechas espaciadas uniformemente desde 1998-03-10 hasta 1998-03-15 con frecuencia diaria.

```
1  pd.date_range('1998-03-10', '1998-03-15', freq='D')
```

El `DatetimeIndex` resultante tiene un atributo `freq` seteado a un valor `'D'`, que indica una frecuencia diaria. Las frecuencias disponibles en pandas incluyen cada hora (`'H'`), calendario diario (`'D'`), calendario de negocios (`'B'`), semanal (`'W'`), mensual (`'M'`), trimestral (`'Q'`), anual (`'A'`), y muchos otros. Las frecuencias también se pueden especificar como múltiplos de cualquiera de las frecuencias base, por ejemplo, `'5D'` por cada cinco días.

Como otro ejemplo, creemos un rango de fechas con frecuencia por hora, especificando la fecha de inicio y el número de períodos, en lugar de la fecha de inicio y la fecha de finalización.

```
pd.date_range('2004-09-20', periods=8, freq='H')
```

Ahora echemos otro vistazo al `DatetimeIndex` de nuestra serie de tiempo `opsd_daily`.

```
opsd_daily.index
```

Podemos ver que no tiene frecuencia (`freq = None`). Esto tiene sentido, ya que el índice se creó a partir de una secuencia de fechas en nuestro archivo CSV, sin especificar explícitamente ninguna frecuencia para la serie temporal.

Si sabemos que nuestros datos deben estar en una frecuencia específica, podemos usar el método `asfreq()` de los DataFrame para asignar una frecuencia. Si falta alguna fecha / hora en los datos, se agregarán nuevas filas para esas fechas / horas, que están vacías (NaN) o se llenan de acuerdo con un método de llenado de datos especificado, como el llenado directo o la interpolación.

Para ver cómo funciona esto, creemos un nuevo DataFrame que contenga solo los datos de Consumo del 3, 6 y 8 de febrero de 2013.

```
1  # Para seleccionar una secuencia arbitraria de valores de
2  # fecha / hora de una serie de tiempo de pandas, necesitamos
3  # usar un DatetimeIndex,en lugar de simplemente una lista
4  # de cadenas de fecha / hora
5  times_sample = pd.to_datetime(['2013-02-03', '2013-02-06', '2013-02-08'])
6  # Seleccionamos las fechas especificadas y solo la columna Consumo
7  consum_sample = opsd_daily.loc[times_sample, ['Consumption']].copy()
8  consum_sample
```

Ahora usamos el método `asfreq()` para convertir el DataFrame a frecuencia diaria, con una columna para datos sin completar y una columna para datos con relleno directo.

```
1  # Convierta los datos a frecuencia diaria, sin rellenar
2  # los datos perdidos (NaN)
3  consum_freq = consum_sample.asfreq('D')
4  # Crea una columna con datos perdidos (NaN)
5  consum_freq['Consumption - Forward Fill'] = consum_sample.
6  asfreq('D', method='ffill')
7  consum_freq
```

En la columna `Consumption`, tenemos los datos originales, con unos valores de NaN para cualquier fecha que faltaba en nuestro dataframe `consum_sample`. En la columna `Consumption - Forward Fill`, las valores perdidos (NaN) se han completado hacia adelante (fordward filled), lo que significa que el último valor se repite a través de las filas faltantes hasta que se encuentra el siguiente valor no perdido. Si está haciendo un análisis de series de tiempo que requiere datos espaciados uniformemente sin faltas, querrá usar `asfreq()` para convertir sus series de tiempo a la frecuencia especificada y llenar las faltas con un método apropiado.

A menudo es útil volver a muestrear nuestros datos de series temporales a una frecuencia más baja o más alta. El muestreo (resampling) a una frecuencia más baja (downsampling) generalmente implica una operación de agregación, por ejemplo, calcular los totales de ventas mensuales a partir de datos diarios. El muestreo a una frecuencia más alta (upsampling) es menos común y a menudo implica la interpolación u otro método de llenado de datos, por ejemplo, la interpolación de datos meteorológicos por hora a intervalos de 10 minutos para ingresar a un modelo científico.

Nos centraremos aquí en la disminución de muestras (downsampling), explorando cómo puede ayudarnos a analizar nuestros datos de OPSD en varias escalas de tiempo. Utilizamos el método `resample()` de DataFrame, que divide el `DatetimeIndex` en intervalos de tiempo y agrupa los datos por intervalo de tiempo. El método `resample()` devuelve un objeto `Resampler`, similar a un objeto `GroupBy` de pandas . Luego podemos aplicar un método de agregación como `mean()`, `median()`, `sum()`, etc., al grupo de datos para cada intervalo de tiempo.

Por ejemplo, remuestreemos los datos a una serie de tiempo semanal.

```
# Espefificamos las columnas que deseamos incluir
#  (por ejemplo excluimos Year, Month, Weekday Name)
data_columns = ['Consumption', 'Wind', 'Solar', 'Wind+Solar']
# remuestreamos a frecuencia semanal, agregando la media
opsd_weekly_mean = opsd_daily[data_columns].resample('W').mean()
opsd_weekly_mean.head(3)
```

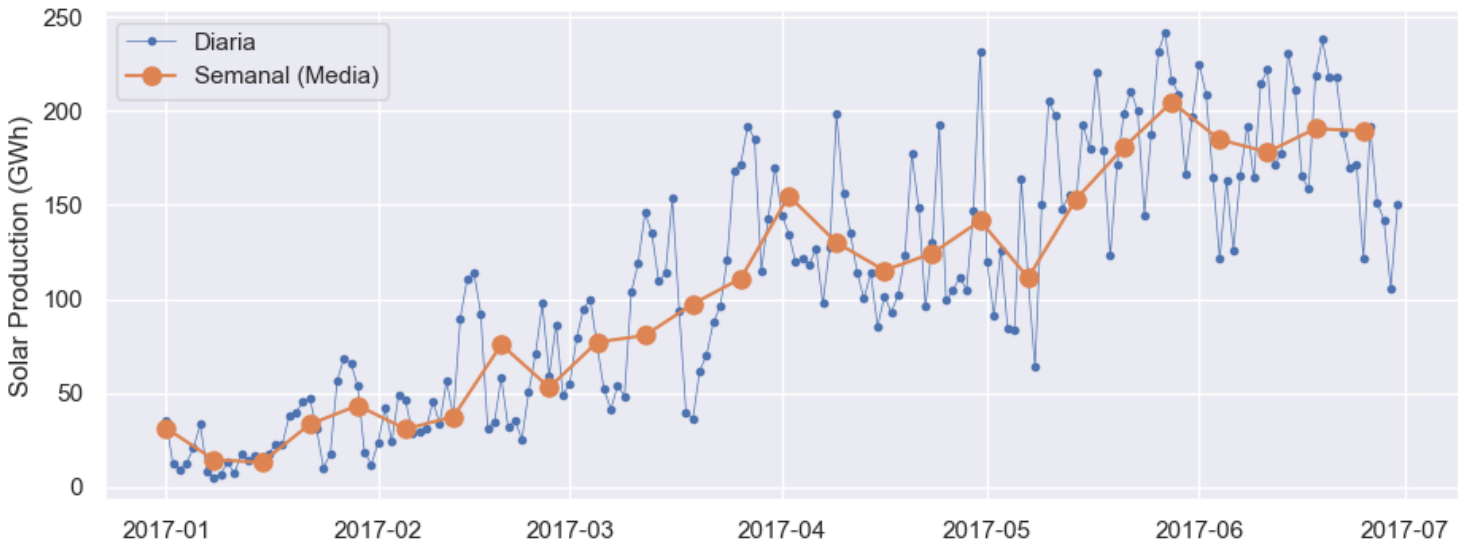
La primera fila de arriba, etiquetada 2006-01-01, contiene la media de todos los datos contenidos en el intervalo de tiempo 2006-01-01 a 2006-01-07. La segunda fila, etiquetada 2006-01-08, contiene los datos medios para el intervalo de tiempo 2006-01-08 a 2006-01-14, y así sucesivamente. De manera predeterminada, cada fila de la serie de tiempo con muestreo downsampled está etiquetada con el extremo izquierdo del intervalo de tiempo.

Grafiquemos las series de tiempo en escalas (resoluciones) diarias y semanales de la variables `Solar` juntas en un solo período de seis meses para compararlas.

```
# inicio,fin del rando de tiempo a analizar
start, end = '2017-01', '2017-06'
# Series de tiempo remuestreada (resampled)
# en escala diaria y semanal
fig, ax = plt.subplots()
ax.plot(opsd_daily.loc[start:end, 'Solar'],
marker='.', linestyle='-', linewidth=0.5, label='Diaria')
ax.plot(opsd_weekly_mean.loc[start:end, 'Solar'],
marker='o', markersize=8, linestyle='-',
label='Semanal (Media)')
ax.set_ylabel('Solar Production (GWh)')
ax.legend();
```

# *Análisis de series de tiempo con pandas*

## *Resampling*




Podemos ver que la serie de tiempo promedio semanal es más suave que la serie de tiempo diaria porque se ha promediado una mayor variabilidad de frecuencia en el muestreo.

Ahora volvamos a muestrear (resample) los datos a la frecuencia mensual, agregando la suma de todos los elementos en lugar de la media. A diferencia de agregar con `mean()`, que establece la salida en NaN para cualquier período con todos los datos faltantes, el comportamiento predeterminado de `sum()` devolverá la salida de 0 como la suma de los datos faltantes. Usamos el parámetro `min_count` para cambiar este comportamiento.

```
# Calcula la suma mensual, establece el valor en NaN para
# cualquier mes que tenga menos de 28 dias de datos
opsd_monthly = opsd_daily[data_columns]. \
resample('M'). \
sum(min_count=28)
opsd_monthly.head(3)
```

Es posible que observe que los datos muestreados mensualmente se etiquetan con el final de cada mes (el extremo derecho del intervalo de tiempo), mientras que los datos remuestreados semanales se etiquetan con el extremo izquierdo izquierdo del intervalo de tiempo. De manera predeterminada, los datos muestreados se etiquetan con el extremo derecho para las frecuencias mensuales, trimestrales y anuales, y con el extremo izquierdo para todas las demás frecuencias. Este comportamiento y varias otras opciones se pueden ajustar utilizando los parámetros listados en la documentación de la función `resample()`.

Ahora exploremos la serie de tiempo mensual trazando el consumo de electricidad como un diagrama lineal, y la producción de energía eólica y solar juntas como un diagrama de área apilada.

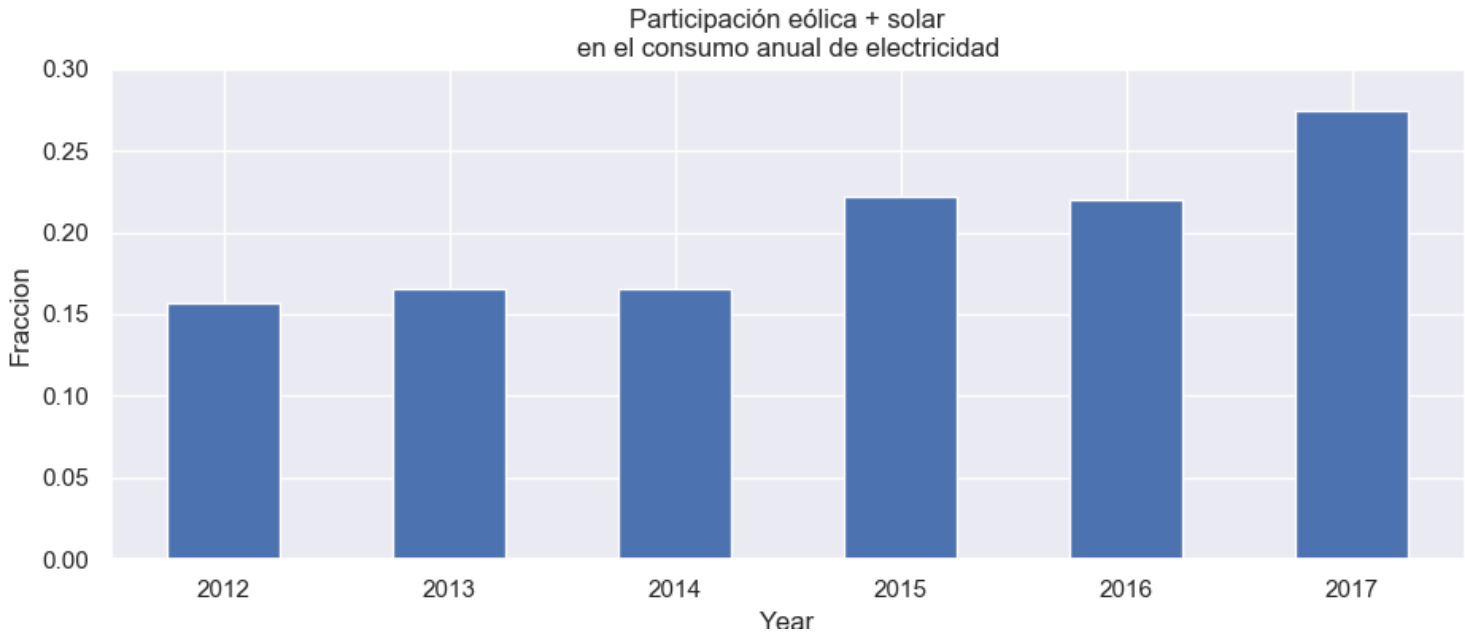
```
# Calcule las sumas anuales, estableciendo el valor en NaN para
# cualquier year que tenga menos de 360 dias de datos.
opsd_annual = opsd_daily[data_columns].resample('A'). \
sum(min_count=360)
# El indice predeterminado del DataFrame muestreado es el ultimo
# dia de cada year ('2006-12-31', '2007-12-31', etc.) para facilitar
# la vida, establezca el indice en el componente year
opsd_annual = opsd_annual.set_index(opsd_annual.index.year)
opsd_annual.index.name = 'Year'
# Calcule la relacion de viento y solar a consumo
opsd_annual['Wind+Solar/Consumption'] = opsd_annual['Wind+Solar'] / \
opsd_annual['Consumption']
opsd_annual.tail(3)
```

Finalmente, grafiquemos la participación eólica + solar del consumo anual de electricidad como un gráfico de barras.

```
# plotear desde 2012 en adelante, porque no hay datos de
# produccion solar en pasados year
ax = opsd_annual.loc[2012:, 'Wind+Solar/Consumption']. \
plot.bar(color='C0')
ax.set_ylabel('Fraccion')
ax.set_ylim(0, 0.3)
Titulo = 'Participacion eolica + solar' \
'\n' \
'en el consumo anual de electricidad'
ax.set_title(Titulo)
plt.xticks(rotation=0);
```

# *Análisis de series de tiempo con pandas*

## *Resampling*




Podemos ver que la producción eólica + solar como parte del consumo anual de electricidad ha aumentado de aproximadamente un 15% en 2012 a aproximadamente un 27% en 2017.