Programación para el análisis de datos

Federico Pousa
fpousa@udesa.edu.ar

Hoy:
Números, vectores, matrices

Numpy+Scipy

Numpy

- Libreria para Python
- Arreglos multidimensionales y Matrices
- Funciones matemáticas hiper-optimizadas
- Buenas conexiones con lenguajes de más bajo nivel (C,C++,Fortran)
- NumPy — NumPy

Cómo instalamos Numpy

- pip install numpy
- conda install numpy

Arrays

Los arrays son el tipo de datos más importante que provee NumPy. En su versión más básica representan vectores pero pueden tener más dimensiones y representar matrices y tensores en general.

El tipo se llama *ndarray* pero también se lo conoce en la librería simplemente como *array*.

En su versión más simple podemos pensar que no es más que una lista de python pero:

1. A diferencia de las listas en python, solo pueden tener un tipo de datos adentro.
2. Existen un montón de operaciones matemáticas definidas y optimizadas para trabajar con este tipo de datos.

Al ser una clase de python, además de métodos tiene atributos, algunos de ellos son:

- shape: Indica las dimensiones del array.
- ndim: Indica la cantidad de dimensiones del array.
- size: Indica la cantidad total de elementos en el array.
- dtype: Indica el tipo de datos de los elementos del array.
- data: Contiene todo los valores del array.

Arrays

```
In [1]: import numpy as np

In [2]: an_array = np.arange(20)

In [3]: an_array
Out[3]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
               17, 18, 19])

In [4]: an_array.shape
Out[4]: (20,)

In [5]: an_array.ndim
Out[5]: 1

In [6]: an_array.size
Out[6]: 20

In [7]: an_array.dtype
Out[7]: dtype('int64')

In [8]: an_array.data
Out[8]: <memory at 0x7fa52d9dcc40>
```

Matrices

Las matrices no son más que array con 2 dimensiones. Si bien existe un tipo específico para matrices en numpy cayó en desuso (y obsolescencia).

```
In [9]: a_matrix = np.arange(16).reshape(4,4)

In [10]: a_matrix
Out[10]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11],
                [12, 13, 14, 15]])

In [11]: a_matrix.shape
Out[11]: (4, 4)

In [12]: a_matrix.ndim
Out[12]: 2

In [13]: a_matrix.size
Out[13]: 16

In [14]: a_matrix.dtype
Out[14]: dtype('int64')

In [15]: a_matrix.data
Out[15]: <memory at 0x7fa52d6c4c70>
```

Arrays reshape

Las dimensiones de un array se pueden manipular como se vió en el ejemplo de creación de la matriz.

Algunos métodos que modifican las dimensiones:

- reshape: Devuelve un nuevo array con las dimensiones indicadas cómo parámetro. Si alguno de los parámetros es igual a -1, se calculan las dimensiones para que sea factible el cambio.
- resize: el mismo efecto que “reshape” pero modifica el array en vez de devolver uno nuevo.
- T: sirve para transponer una matriz.
- ravel: “aplana” el array devolviendo todo en una sola dimensión.

Arrays reshape

```
In [25]: a_matrix
Out[25]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11],
                [12, 13, 14, 15]])

In [26]: a_matrix.reshape(8,2)
Out[26]: array([[ 0,  1],
                [ 2,  3],
                [ 4,  5],
                [ 6,  7],
                [ 8,  9],
                [10, 11],
                [12, 13],
                [14, 15]])

In [27]: a_matrix.shape
Out[27]: (4, 4)

In [28]: a_matrix.reshape(8,-1)
Out[28]: array([[ 0,  1],
                [ 2,  3],
                [ 4,  5],
                [ 6,  7],
                [ 8,  9],
                [10, 11],
                [12, 13],
                [14, 15]])
```

```
In [29]: a_matrix.resize(8,2)

In [30]: a_matrix.shape
Out[30]: (8, 2)

In [31]: a_matrix.T
Out[31]: array([[ 0,  2,  4,  6,  8, 10, 12, 14],
                [ 1,  3,  5,  7,  9, 11, 13, 15]])

In [32]: a_matrix.ravel()
Out[32]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
```

Armando arrays

Si queremos crear nuevos arrays, existen diversas maneras.

Por un lado, tenemos el constructor de la clase que admite como parámetro listas de valores.

Por otro lado, existen diversas funciones que crean arrays:

- arange: similar al “range” de python. Permite crear una array a partir de una secuencia de valores indicando principio, fin e intervalo.
- linspace: sirve para crear un array equiespaciado entre un número inicial y uno final, indicando la cantidad de elementos que se quieren.
- zeros: devuelve un array lleno de ceros, del tamaño indicado por parámetro.
- ones: devuelve un array lleno de unos, del tamaño indicado por parámetro.
- Muchas otras funciones que devuelven arrays particulares como por ejemplo “un array de 8x2 con todos elementos sampleados de una distribución normal con media 0 y varianza 1”.

Armando arrays

```
In [24]: np.arange(10, 30, 2)

Out[24]: array([10, 12, 14, 16, 18, 20, 22, 24, 26, 28])

In [25]: np.linspace(0.1, 0.5, 4)

Out[25]: array([0.1       , 0.23333333, 0.36666667, 0.5       ])

In [26]: np.logspace(0.1, 0.5, 4)

Out[26]: array([1.25892541, 1.7113283 , 2.32630507, 3.16227766])

In [27]: np.zeros((3,2))

Out[27]: array([[0., 0.],
                [0., 0.],
                [0., 0.]])

In [28]: np.ones((2,3))

Out[28]: array([[1., 1., 1.],
                [1., 1., 1.]])

In [29]: np.random.standard_normal((8,2))

Out[29]: array([[ 1.05197874,  1.01449052],
                [ 0.39963202, -0.42015446],
                [ 0.32597474, -0.57581449],
                [-0.36708382, -1.06729137],
                [-0.84317929,  0.369063  ],
                [ 0.6221361 , -1.30865083],
                [-0.54057614, -0.66797521],
                [ 0.29426669,  0.5861678 ]])
```

Operaciones

Existen muchísimas operaciones definidas para arrays.

Sobrecarga de operadores básicos como sumas y restas de vectores, potencias y raíces posición a posición, etc.

Luego existen una gran variedad de funciones matemáticas a aplicar a cada posición de un vector como funciones trigonométricas, funciones de redondeo, etc.

También existen las operaciones clásicas entre vectores como el producto punto o el producto cruz.

En el caso de las matrices también existe la multiplicación clásica entre matrices (con el símbolo @) o lo que es el producto punto de matrices.

Operaciones

```
In [30]: array_1 = np.random.uniform(0,1,10)
         array_2 = np.random.uniform(0,1,10)

In [31]: array_1

Out[31]: array([0.45670503, 0.28890966, 0.95386456, 0.48863157, 0.76332832,
                0.58262065, 0.31682771, 0.60800377, 0.57455479, 0.34095776])

In [32]: array_2

Out[32]: array([0.21174988, 0.17007836, 0.77240404, 0.66337446, 0.16840258,
                0.32498179, 0.48468262, 0.02138897, 0.75439121, 0.6671835 ])

In [33]: array_1 + array_2

Out[33]: array([0.66845491, 0.45898802, 1.7262686 , 1.15200604, 0.93173091,
                0.90760244, 0.80151033, 0.62939274, 1.328946  , 1.00814126])

In [34]: array_1 - array_2

Out[34]: array([ 0.24495515,  0.1188313 ,  0.18146052, -0.17474289,  0.59492574,
                 0.25763886, -0.16785491,  0.5866148 , -0.17983643, -0.32622574])

In [35]: array_1**3

Out[35]: array([0.0952593 , 0.02411494, 0.86788091, 0.11666607, 0.44476861,
                0.19776873, 0.0318031 , 0.22475989, 0.18966812, 0.03963709])

In [36]: np.sqrt(array_2)

Out[36]: array([0.46016288, 0.41240558, 0.8788652 , 0.81447803, 0.41036884,
                0.57007174, 0.69619151, 0.14624969, 0.86855697, 0.81681301])
```

Operaciones

```
In [37]: np.sin(array_1)
Out[37]: array([0.44099324, 0.28490724, 0.81565737, 0.46941804, 0.69133011,
                0.55021413, 0.31155373, 0.57123011, 0.54346112, 0.33438987])

In [38]: np.cos(array_2)
Out[38]: array([0.97766464, 0.98557151, 0.71623506, 0.7879188 , 0.98585376,
                0.94765654, 0.88482287, 0.99977126, 0.7286886 , 0.78556756])

In [39]: np.floor(array_1)
Out[39]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

In [40]: np.round(array_2)
Out[40]: array([0., 0., 1., 1., 0., 0., 0., 0., 1., 1.])

In [41]: np.dot(array_1, array_2)
Out[41]: 2.3521325570699623

In [42]: array_1 = np.random.uniform(0,1,3)
         array_2 = np.random.uniform(0,1,3)
         np.cross(array_1, array_2)
Out[42]: array([-0.40659118,  0.05745938,  0.3745824 ])
```

```
In [43]: matrix_1 = np.random.uniform(0,1,(3,3))
         matrix_2 = np.random.uniform(0,1,(3,3))

In [44]: matrix_1
Out[44]: array([[0.34624106, 0.61455139, 0.82392586],
                [0.15250895, 0.97087489, 0.89537385],
                [0.56349779, 0.01300908, 0.86004428]])

In [45]: matrix_2
Out[45]: array([[0.55557185, 0.87734073, 0.61252844],
                [0.78623795, 0.96221107, 0.51875193],
                [0.14698919, 0.97042539, 0.12086917]])

In [46]: matrix_1 * matrix_2
Out[46]: array([[0.19236179, 0.53917096, 0.50467802],
                [0.11990833, 0.93418656, 0.46447692],
                [0.08282809, 0.01262434, 0.10395284]])

In [47]: matrix_1 @ matrix_2
Out[47]: array([[0.79665361, 1.69465811, 0.63046945],
                [0.97967864, 1.9368824 , 0.70528239],
                [0.44970896, 1.34150586, 0.45585975]])
```

Universal functions y performance

En NumPy las *universal functions* son todas aquellas funciones que operan elemento a elemento sobre un array de manera predefinida. Varios de los ejemplos vistos en las slides anteriores caen en este tipo de funciones.

Estas funciones se dicen que son *vectorizadas* y están particularmente optimizadas para hacer muy rápidamente la misma función sobre todas las posiciones de un vector de manera muy rápida.

Se podría perfectamente obtener el mismo resultado iterando el array y aplicando la función requerida, pero tomaría mucho más tiempo de cómputo.

Universal functions y performance

Ejercicio:

Comparar el tiempo de cómputo necesario para aplicar la función seno sobre un array cuando se realiza mediante una universal function nativa de NumPy vs si se hace iterando sobre el array posición a posición y aplicando la función seno de la librería “math”.

Realizar la comparación para arrays de 100 millones de elementos.

Universal functions y performance

In [48]:
```
import time
import math

def compute_sin_native(array):
    result = []
    for element in array:
        result.append(math.sin(element))
    return result

def compute_sin_numpy(array):
    result = np.sin(array)
    return result
```

In [49]:
```
an_array = np.arange(100000000)
start_time_native = time.time()
result_native = compute_sin_native(an_array)
end_time_native = time.time()
start_time_numpy = time.time()
result_numpy = compute_sin_numpy(an_array)
end_time_numpy = time.time()
print('Time without numpy:', end_time_native - start_time_native)
print('Time with numpy:', end_time_numpy - start_time_numpy)
print(np.allclose(result_native, result_numpy))
```

```
Time without numpy: 16.849889755249023
Time with numpy: 1.2601311206817627
True
```

Broadcasting

Otra operación que suele resultar muy útil (y eficiente) es el *broadcasting*. Mediante esta propiedad de NumPy podemos operar entre arrays de diferentes dimensiones de manera *razonable*.

La idea en general es que en algunas ocasiones tenemos dos arrays de diferentes dimensiones que, igualmente, queremos que operen de manera conjunta.

El caso más simple es si queremos sumar un único número a todo un array con única dimensión.

```
In [21]: an_array = np.array(range(2,100))

In [22]: a_number = 1

In [23]: an_array_of_ones = np.array([1]*an_array.shape[0])

In [24]: an_array + a_number

Out[24]: array([  3,   4,   5,   6,   7,   8,   9,  10,  11,  12,  13,  14,  15,
                 16,  17,  18,  19,  20,  21,  22,  23,  24,  25,  26,  27,  28,
                 29,  30,  31,  32,  33,  34,  35,  36,  37,  38,  39,  40,  41,
                 42,  43,  44,  45,  46,  47,  48,  49,  50,  51,  52,  53,  54,
                 55,  56,  57,  58,  59,  60,  61,  62,  63,  64,  65,  66,  67,
                 68,  69,  70,  71,  72,  73,  74,  75,  76,  77,  78,  79,  80,
                 81,  82,  83,  84,  85,  86,  87,  88,  89,  90,  91,  92,  93,
                 94,  95,  96,  97,  98,  99, 100])

In [25]: an_array + an_array_of_ones

Out[25]: array([  3,   4,   5,   6,   7,   8,   9,  10,  11,  12,  13,  14,  15,
                 16,  17,  18,  19,  20,  21,  22,  23,  24,  25,  26,  27,  28,
                 29,  30,  31,  32,  33,  34,  35,  36,  37,  38,  39,  40,  41,
                 42,  43,  44,  45,  46,  47,  48,  49,  50,  51,  52,  53,  54,
                 55,  56,  57,  58,  59,  60,  61,  62,  63,  64,  65,  66,  67,
                 68,  69,  70,  71,  72,  73,  74,  75,  76,  77,  78,  79,  80,
                 81,  82,  83,  84,  85,  86,  87,  88,  89,  90,  91,  92,  93,
                 94,  95,  96,  97,  98,  99, 100])

In [26]: for i in range(an_array.shape[0]):
             an_array[i] += 1
         an_array

Out[26]: array([  3,   4,   5,   6,   7,   8,   9,  10,  11,  12,  13,  14,  15,
                 16,  17,  18,  19,  20,  21,  22,  23,  24,  25,  26,  27,  28,
                 29,  30,  31,  32,  33,  34,  35,  36,  37,  38,  39,  40,  41,
                 42,  43,  44,  45,  46,  47,  48,  49,  50,  51,  52,  53,  54,
                 55,  56,  57,  58,  59,  60,  61,  62,  63,  64,  65,  66,  67,
                 68,  69,  70,  71,  72,  73,  74,  75,  76,  77,  78,  79,  80,
                 81,  82,  83,  84,  85,  86,  87,  88,  89,  90,  91,  92,  93,
                 94,  95,  96,  97,  98,  99, 100])
```

Broadcasting

La regla de broadcasting va mirando cada una de las dimensiones de derecha a izquierda.
Dos dimensiones son compatibles si son iguales o si alguna de ellas es 1 (en cuyo caso se va a repetir el array tantas veces como sea necesario para ser igual a la otra dimensión).

En caso de que un array tenga menos dimensiones que el otro, se rellenan las dimensiones faltantes con unos.

https://numpy.org/doc/stable/user/basics.broadcasting.html

```
In [27]: from numpy import array, argmin, sqrt, sum
         observation = array([111.0, 188.0])
         codes = array([[102.0, 203.0],
                        [132.0, 193.0],
                        [45.0, 155.0],
                        [57.0, 173.0]])
         diff = codes - observation    # the broadcast happens here
         dist = sqrt(sum(diff**2,axis=-1))
         argmin(dist)
Out[27]: 0

In [28]: observation.shape
Out[28]: (2,)

In [29]: codes.shape
Out[29]: (4, 2)

In [30]: diff.shape
Out[30]: (4, 2)

In [31]: observation.resize(1,2)

In [33]: observation.shape
Out[33]: (1, 2)

In [34]: diff = codes - observation

In [35]: observation.resize(2,1)

In [36]: observation.shape
Out[36]: (2, 1)

In [37]: diff = codes - observation
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [37], in <cell line: 1>()
----> 1 diff = codes - observation

ValueError: operands could not be broadcast together with shapes (4,2) (2,1)
```

Accediendo y recorriendo

Los arrays en NumPy se pueden acceder a posiciones particulares o mediante *slicing* de maneras parecidas a las listas nativas.

- Con [i] se puede acceder a la posición *i* de un array unidimensional.
- Con [start : end : step] se puede hacer slicing desde la posición *start* hasta la *end* tomando cada *step* posiciones. Algunos de estos se pueden omitir y ser interpretados de manera default. También se pueden utilizar números negativos con semántica análoga a lo que sucede en las listas.
- Con [eje1 , eje2 , … , ejen] se puede acceder a arrays multidimensionales indicando una regla independiente por cada dimensión.
- Con un *for* clásico, se puede iterar sobre el primer eje del array.

Accediendo y recorriendo

```
In [50]: an_array = np.arange(0,40,2)

In [51]: an_array

Out[51]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32,
                34, 36, 38])

In [52]: an_array[6]

Out[52]: 12

In [53]: an_array[2:18:2]

Out[53]: array([ 4,  8, 12, 16, 20, 24, 28, 32])

In [54]: an_array[:18:]

Out[54]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32,
                34])

In [55]: an_array[:-1:]

Out[55]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32,
                34, 36])

In [56]: an_array[::]

Out[56]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32,
                34, 36, 38])

In [57]: an_array[::-1]

Out[57]: array([38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14, 12, 10,  8,  6,
                4,  2,  0])

In [58]: for element in an_array[:5]:
             print(element)

         0
         2
         4
         6
         8
```

Accediendo y recorriendo

```
In [59]: a_matrix = np.random.standard_normal((4,4))

In [60]: a_matrix

Out[60]: array([[-0.38718895, -1.39387653,  0.1837009 , -1.80026133],
                [-1.23972156, -1.53819126, -0.8442662 ,  0.22694993],
                [-0.25084176, -0.05411051, -1.1535745 , -0.30982147],
                [ 0.49917184,  0.65297332, -3.04488895,  0.09003538]])

In [61]: a_matrix[:,2]

Out[61]: array([ 0.1837009 , -0.8442662 , -1.1535745 , -3.04488895])

In [62]: a_matrix[:,1:3]

Out[62]: array([[-1.39387653,  0.1837009 ],
                [-1.53819126, -0.8442662 ],
                [-0.05411051, -1.1535745 ],
                [ 0.65297332, -3.04488895]])

In [63]: a_matrix[-1,:]

Out[63]: array([ 0.49917184,  0.65297332, -3.04488895,  0.09003538])

In [64]: for row in a_matrix:
             print(row)

[-0.38718895 -1.39387653  0.1837009  -1.80026133]
[-1.23972156 -1.53819126 -0.8442662   0.22694993]
[-0.25084176 -0.05411051 -1.1535745  -0.30982147]
[ 0.49917184  0.65297332 -3.04488895  0.09003538]
```

Tipos de asignación

Cuando se asigna una variable a partir de otra (an_object = another_object) puede suceder que la información sea creada de dos formas distintas:

- Por copia: se crea una copia de toda la información que está en *another_object* y se asigna esa nueva información a la variable *an_object*. Si una variable cambia, la otra no se entera.
- Por referencia: no se crea nueva información, sino que la variable *an_object* apunta al “mismo lugar de la memoria” que *another_object*. Si una variable cambia, la otra cambia.

En Python nativo algunas asignaciones son por copia y algunas son por referencia (¿Cuáles?), pero en NumPy el default es por referencia a menos que se pida explícitamente una copia.

- *an_array = another_array*: genera una nueva referencia al mismo array.
- *an_array = another_array.*view(): genera una *vista* o *shallow copy*. Se crea un objeto nuevo, pero los datos del array están referenciados.
- *an_array = another_array.copy()*: genera un objeto completamente nuevo con toda la información copiada.

Tipos de asignación

Referencia

```
In [65]: another_array = np.arange(10)

In [66]: another_array
Out[66]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [67]: an_array = another_array

In [68]: an_array[3] = 8

In [69]: another_array
Out[69]: array([0, 1, 2, 8, 4, 5, 6, 7, 8, 9])
```

Vista

```
In [70]: another_array = np.arange(10)

In [71]: another_array
Out[71]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [72]: an_array = another_array.view()

In [73]: an_array = an_array.reshape(5,2)

In [74]: an_array
Out[74]: array([[0, 1],
                [2, 3],
                [4, 5],
                [6, 7],
                [8, 9]])

In [75]: another_array
Out[75]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [76]: another_array[3] = 8

In [77]: an_array
Out[77]: array([[0, 1],
                [2, 8],
                [4, 5],
                [6, 7],
                [8, 9]])
```

Copia

```
In [78]: another_array = np.arange(10)

In [79]: an_array = another_array.copy()

In [80]: an_array[3] = 8

In [81]: an_array
Out[81]: array([0, 1, 2, 8, 4, 5, 6, 7, 8, 9])

In [82]: another_array
Out[82]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

SciPy

SciPy es una librería que trabaja directamente sobre NumPy y la extiende con un montón de algoritmos útiles en diferentes áreas.

- Visualizaciones
- Optimizaciones
- Polinomios, Integraciones, Interpolaciones, Transformadas de Fourier
- Algebra Lineal
- Señales
- **Estadistica**

¿Cómo la instalamos?

```
pip install scipy
```

Paquetes de SciPy

SciPy Organization

SciPy is organized into subpackages covering different scientific computing domains. These are summarized in the following table:

Subpackage	Description
`cluster`	Clustering algorithms
`constants`	Physical and mathematical constants
`fftpack`	Fast Fourier Transform routines
`integrate`	Integration and ordinary differential equation solvers
`interpolate`	Interpolation and smoothing splines
`io`	Input and Output
`linalg`	Linear algebra
`ndimage`	N-dimensional image processing
`odr`	Orthogonal distance regression
`optimize`	Optimization and root-finding routines
`signal`	Signal processing
`sparse`	Sparse matrices and associated routines
`spatial`	Spatial data structures and algorithms
`special`	Special functions
`stats`	Statistical distributions and functions

SciPy: Un pantallazo

Algebra lineal: https://docs.scipy.org/doc/scipy-1.8.0/html-scipyorg/reference/linalg.html

Linear Algebra (**scipy.linalg**)

- scipy.linalg vs numpy.linalg
- numpy.matrix vs 2-D numpy.ndarray
- Basic routines
    - Finding the inverse
    - Solving a linear system
    - Finding the determinant
    - Computing norms
    - Solving linear least-squares problems and pseudo-inverses
    - Generalized inverse
- Decompositions
    - Eigenvalues and eigenvectors
    - Singular value decomposition
    - LU decomposition
    - Cholesky decomposition
    - QR decomposition
    - Schur decomposition
    - Interpolative decomposition
- Matrix functions
    - Exponential and logarithm functions
    - Trigonometric functions
    - Hyperbolic trigonometric functions
    - Arbitrary function
- Special matrices

SciPy: Un pantallazo

Integración: https://docs.scipy.org/doc/scipy-1.8.0/html-scipyorg/reference/integrate.html

Integration (**scipy.integrate**)

- General integration (**quad**)
- General multiple integration (**dblquad**, **tplquad**, **nquad**)
- Gaussian quadrature
- Romberg Integration
- Integrating using Samples
- Faster integration using low-level callback functions
- Ordinary differential equations (**solve_ivp**)
    - Solving a system with a banded Jacobian matrix
    - References

SciPy: Un pantallazo

Optimización: https://docs.scipy.org/doc/scipy-1.8.0/html-scipyorg/reference/optimize.html

Optimization (**scipy.optimize**)

- Unconstrained minimization of multivariate scalar functions (**minimize**)
  - Nelder-Mead Simplex algorithm (**method='Nelder-Mead'**)
  - Broyden-Fletcher-Goldfarb-Shanno algorithm (**method='BFGS'**)
  - Newton-Conjugate-Gradient algorithm (**method='Newton-CG'**)

  Trust-Region Newton-Conjugate-Gradient Algorithm (**method='trust-ncg'**)
- Constrained minimization of multivariate scalar functions (**minimize**)
  - Sequential Least SQuares Programming (SLSQP) Algorithm (**method='SLSQP'**)
- Global optimization
- Least-squares minimization (**least_squares**)
- Univariate function minimizers (**minimize_scalar**)
- Custom minimizers
- Root finding

SciPy: Un pantallazo

Interpolación: https://docs.scipy.org/doc/scipy-1.8.0/html-scipyorg/reference/interpolate.html

Interpolation (**scipy.interpolate**)

- 1-D interpolation (**interp1d**)
- Multivariate data interpolation (**griddata**)
- Spline interpolation
  - Spline interpolation in 1-D: Procedural (interpolate.splXXX)
  - Spline interpolation in 1-d: Object-oriented (**UnivariateSpline**)
  - 2-D spline representation: Procedural (**bisplrep**)
  - 2-D spline representation: Object-oriented (**BivariateSpline**)
- Using radial basis functions for smoothing/interpolation
  - 1-D Example
  - 2-D Example

A hacer la práctica!
(hay más de 100 ejercicios)