Python en Ámbitos Científicos

Facundo Batista & Manuel Carlevaro

7 de noviembre de 2024

VERSIÓN PRELIMINAR

Título: Python en Ámbitos Científicos
Autores: Facundo Batista & Manuel Carlevaro
ISBN-13 (versión electrónica): ???-?-????-????-?

Primera Edición (versión preliminar)
Escrito con XeLaTeX.

Licencia: Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0)
Lugar: Olivos y La Plata, Buenos Aires, Argentina
Año: 2024
Web: http://pyciencia.taniquetil.com.ar/

10 9 8 7 6 5 4 3 2 1

VERSIÓN PRELIMINAR

Prefacio

Cuando la Comunidad Científica notó que las computadoras podían facilitar su trabajo, sus integrantes se convirtieron posiblemente en los adoptantes tempranos más entusiastas de la tecnología. Aunque el uso y la potencia de las computadoras crece continuamente en el ámbito científico, poco se ha avanzado en la formación y desarrollo de habilidades en Desarrollo de Software.

El cálculo científico requiere realizar combinaciones de múltiples tareas de diversa clase. Por ejemplo, es necesario registrar automáticamente datos de un experimento y visualizarlos, realizar cálculos numéricos o simbólicos, ordenar, clasificar, simular, etc. Muchas veces podemos utilizar paquetes de software que realizan esas tareas por nosotros, pero muchas otras ocurre que nadie ha implementado un determinado cómputo en la forma que necesitamos, o simplemente queremos probar ideas nuevas. Cualquiera sea la naturaleza de nuestra actividad en la ciencia o en la tecnología, no resulta infrecuente la necesidad de interactuar con computadoras a través de programas propios.

El uso de Python en aplicaciones científicas ha aumentado sostenidamente en los últimos años, sin embargo es difícil encontrar libros o manuales en castellano de Python que no estén pensados para programadores.

Este libro nace con la idea primaria de acercar Python al mundo científico, en un libro pensado para científiques, a partir de nuestra experiencia en el dictado del curso "Herramientas Computacionales para Científicos"que ofrecemos en la Universidad Nacional de La Plata y la Universidad Tecnológica Nacional, desde 2007. De la misma manera, la elección del castellano como idioma de escritura es un factor crítico, porque aunque sabemos que el inglés es una herramienta fundamental tanto para programar como para hacer ciencia, estamos convencidos que no debería ser una barrera de entrada.

Más allá de las secciones básicas de un libro (índice, bibliografía, etc.), el libro tiene tres grandes partes. La primera habla de Python, tanto de forma introductoria como también sobre otros temas que son fundamentales y algunas bibliotecas importantes. La segunda trata algunas herramientas fundamentales que son base para el trabajo en el resto del libro. Finalmente la tercera parte muestra cómo abordar temas científicos básicos utilizando Python, de forma teórica y práctica.

En todos los casos este libro esquiva la pretensión de ser una referencia absoluta, sino que tiene el propósito de allanar el camino de les científiques para dar los primeros pasos en el lenguaje y solucionar los problemas básicos (pero no por eso menos importantes) de la ciencia y la tecnología.

Olivos y La Plata, Buenos Aires, Argentina,

Facundo Batista & Manuel Carlevaro

VERSIÓN PRELIMINAR

Índice general

VERSIÓN PRELIMINAR

Parte I
Python

Esta primera parte comprende varios capítulos orientados a proveer la información necesaria de Python para poder entender el resto del libro.

Se sugiere a los lectores que no tienen experiencia previa en el uso de Python, un recorrido secuencial a través de los capítulos que componen esta Parte.

Parte II
Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos sobre herramientas fundamentales que serán aprovechados en la Parte III al abordar temas de aplicaciones específicas.

1 | Numpy

NumPy es una biblioteca muy utilizada en Python para trabajar con vectores multidimensionales (arreglos, matrices, etc.) que ofrece también una gran colección de funciones matemáticas de alto nivel para operar con ellos.

El punto de entrada a NumPy es la estructura `array`, que nos permite manejar matrices de cualquier dimensión (incluso de una). Estos arreglos, a diferencia de las listas integradas en el lenguaje, son homogéneos: todos sus elementos deben ser del mismo tipo (lo cual es clave en la eficiencia y potencia de cálculo de esta biblioteca).

Módulo	Versión
NumPy	1.26.4

Código disponible

Podemos crear un arreglo utilizando un iterable como fuente:

Cell 01

```
import numpy as np
```

Cell 03

```
np.array([0, 1, 2, 3, 4, 5])
```

```
array([0, 1, 2, 3, 4, 5])
```

Es común importar el módulo principal de NumPy llamándolo `np`: es más corto y conciso, y no confunde porque todo el mundo está acostumbrado a eso.

NumPy nos ofrece formas rápidas de crear arreglos de distintos tamaños, por ejemplo con los números secuenciales como el conocido `range`, pero también inicializado con ceros y con unos (lo cual es muy normal, ya que son la identidad aditiva y multiplicativa respectivamente):

Cell 04

```
np.arange(6)
```

```
array([0, 1, 2, 3, 4, 5])
```

CELL 05

```
np.ones(4)
```

```
array([1., 1., 1., 1.])
```

CELL 06

```
np.zeros(10)
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

En este capítulo vamos a ver otras funciones que posee NumPy, como estas que acabamos de mostrar, pero siempre es interesante explorar y tener a mano la Documentación de Referencia [2].

Vemos que por default NumPy crea todos los arreglos con `float`s. Como decíamos antes, todos los elementos tienen que ser del mismo tipo, pero tenemos control sobre cual tipo es ese; por ejemplo, si vamos a trabajar con enteros:

CELL 07

```
np.ones(4, dtype=np.int32)
```

```
array([1, 1, 1, 1], dtype=int32)
```

1.1. Forma de trabajo

Hagamos una comparación entre las listas integradas y este nuevo tipo de dato, para empezar a notar las diferencias. Supongamos que queremos trabajar sobre un millón de números al mismo tiempo (pero por simplificación para el libro, usemos sólo 10... para el caso es lo mismo). Entonces tenemos nuestro millón de números:

CELL 08

```
clasica = list(range(10))
clasica
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Esto, en numpy sería:

CELL 09

```
nueva = np.arange(10)
nueva
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Ahora, supongamos que queremos los cuadrados de esos números. Como vimos en la subsección **??**, podemos hacer una list comprehension. Ingenuamente, intentamos lo mismo para ambas estructuras.

Cell 10

```
[x ** 2 for x in clasica]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Cell 11

```
np.array([x ** 2 for x in nueva])
```

```
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

Y acá es donde descubrimos el inmenso poder de NumPy: nos permite procesar los arreglos directamente, manejando simultaneamente todos sus elementos internos.

En el ejemplo que acabamos de mostrar, Python tiene que recorrer el iterable, ir recibiendo uno por uno los elementos, elevar ese elemento al cuadrado, e irlo agregando a una lista. Con diez elementos no pasa nada, pero recordemos, nuestro ejemplo imaginario tiene un millón de números.

Numpy, repetimos, nos permite procesar todos los elementos del arreglo al mismo tiempo:

Cell 12

```
nueva
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Cell 13

```
nueva ** 2
```

```
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

Acá tenemos en Python solamente una llamada a una función que estará implementada en C, Fortran o algún lenguaje de bajo nivel, y fuertemente optimizada. Y se realiza todo el procesamiento necesario solamente con esa llamada a función, obteniendo efectivamente performances comparables con esos lenguajes de más bajo nivel. De cualquier manera, la idea de usar NumPy es justamente despreocuparnos sobre cómo está implementada y optimizada, y enfocarnos en usar la biblioteca correctamente, de la misma manera que al usar Python en sí por ejemplo nos despreocupamos sobre la administración dinámica que hace de la memoria.

Como ejercicio, comparemos las performances de los dos casos de nuestro ejemplo anterior.

CELL 14

```
def f():
    clasica = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    cuads = [x ** 2 for x in clasica]

def g():
    nueva = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
    cuads = nueva ** 2

print("Método clásico:")
%timeit f()
print("Usando numpy:")
%timeit g()
```

```
Método clásico:
1.73 µs ± 9.7 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
Usando numpy:
1.79 µs ± 17.5 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

¡Epa! Vemos que es más lento en NumPy, ¿cómo puede ser eso? Simple, porque no tiene sentido usar NumPy para conjuntos pequeños de datos, realmente estamos moviendo toda una infrastructura para procesar pocos casos. Tengamos en cuenta que `timeit.timeit` mide lo que pedimos un millón de veces por default, así que el número que estamos viendo ahí es en micro-segundos.

Veamos un ejemplo más real, haciendo lo mismo pero con un millón de números:

CELL 15

```
def f():
    clasica = range(1_000_000)
    cuads = [x ** 2 for x in clasica]

def g():
    nueva = np.arange(1_000_000)
    cuads = nueva ** 2

print("Método clásico:")
%timeit f()
print("Usando numpy:")
%timeit g()
```

```
Método clásico:
190 ms ± 2.34 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
Usando numpy:
1.97 ms ± 65.2 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Como trabajar sobre un millón de números obviamente tarda mucho más, le decimos a `timeit.timeit` que mida sólo una determinada cantidad de veces, y luego dividimos por esa cantidad, así que ahora estamos viendo segundos.

Y encontramos que para este caso más real, NumPy es más de cien veces más rápido.

Entonces, es responsabilidad nuestra saber cuando utilizar esta biblioteca. Por un lado, no vale la pena para algunos pocos datos, pero especialmente tenemos que tener el cuidado de NO caer en la tentación de procesar los arreglos "a mano en Python", sino siempre utilizar las herramientas

correctas de NumPy.

¿Tenemos que sumar los números usando el `sum` de Python? No. ¿Sacar el máximo usando `max`? Tampoco. ¿Y si queremos aplicar alguna función matemática? De nuevo, usar las herramientas de NumPy:

CELL 16

```
arr = np.arange(10)
arr
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

CELL 17

```
arr.max()
```

```
9
```

CELL 18

```
arr.sum()
```

```
45
```

CELL 19

```
np.sin(arr)
```

```
array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ,
       -0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849])
```

1.2. Multidimensionalidad

Hasta ahora sólo armamos un arreglo de una sola dimensión, pero como decíamos arriba, NumPy permite que sean de varias dimensiones.

Incluso esas dimensiones pueden no ser iguales. Armemos por ejemplo una matriz de dos dimensiones que nos quede como una "tabla rectangular":

CELL 20

```
np.ones((2, 5))
```

```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

No estamos limitados en cantidad de dimensiones, armemos un arreglo de tres dimensiones: un paralelepípedo de 2x2x5:

Cell 21

```
np.ones((2, 2, 5))
```

```
array([[[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]],

       [[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]]])
```

También, podemos cambiarle la forma. Armemos la misma matriz pero con números secuenciales:

Cell 22

```
lineal = np.arange(20)
lineal
```

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

Cell 23

```
lineal.shape
```

```
(20,)
```

Cell 24

```
matriz = lineal.reshape((2, 2, 5))
matriz
```

```
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9]],

       [[10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]]])
```

Cell 25

```
matriz.shape
```

```
(2, 2, 5)
```

En el ejemplo vemos como podemos preguntar la forma de un arreglo: en el primer caso es de largo 20 en la única dimensión que tiene, mientras que ya la matriz tiene largos 2, 2 y 5 en cada una de sus dimensiones.

También podemos preguntar la cantidad de dimensiones, y el tamaño del arreglo (que es sencillamente la cantidad de elementos que contiene, no confundir con los otros atributos recién mencionados):

Cell 26

```
lineal.ndim
```

```
1
```

Cell 27

```
matriz.ndim
```

```
3
```

Cell 28

```
lineal.size
```

```
20
```

Cell 29

```
matriz.size
```

```
20
```

Para acceder a los elementos adentro de los arreglos, NumPy nos permite explotar la natural multidimensionalidad de los mismos, y utilizaremos una sintaxis muy similar a la que estábamos acostumbrados con las secuencias de Python, pero con este gran detalle: podemos expresar al mismo tiempo el índice de cada dimensión, separándolos por coma.

En el caso de los arreglos de una sola dimensión nos queda exactamente como siempre:

Cell 30

```
lineal[4]
```

```
4
```

Cell 31

```
lineal[:4]
```

```
array([0, 1, 2, 3])
```

Es en el caso de los arreglos de más de una dimensión donde podemos explotar esto, si es necesario. En el siguiente ejemplo accedemos a la matriz con un sólo índice, que nos dará la "submatriz" correspondiente:

Cell 32

```
matriz.shape
```

```
(2, 2, 5)
```

Cell 33

```
subm = matriz[0]
subm.shape
```

```
(2, 5)
```

Cell 34

```
subm
```

```
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

Si queremos acceder a la primer submatriz, y luego a un subarreglo de la misma, y de ahí a un elemento, podríamos hacer lo que estamos acostumbrados en Python, o utilizar la forma de NumPy (obviamente, el segundo caso es más rápido, porque no vamos y venimos de NumPy, sino que se procesa todo en una misma llamada):

Cell 35

```
matriz[1][0][3]
```

```
13
```

Cell 36

```
matriz[1,0,3]
```

```
13
```

El verdadero potencial de esta forma se expresa cuando trabajamos con slices, ya que nos permite trabajar con varias dimensiones al mismo tiempo (como recién, separando cada dimensión con coma, ¡pero con slices!). En este caso lo complicado es ajustar el ojo para leer lo que está dentro de los corchetes, porque cuando en Python clásico tenemos algo como `[2:5]`, separamos el contenido con los dos puntos, leyendo el 2 primero, luego el 3, y entendiendo que es un "desde-hasta".

Pero con NumPy la separación realmente es "por dimensión". Entonces si vemos algo como `matriz[:,1:,2]` lo tenemos que separar por las comas, entender que para la primer dimensión tenemos `:` (todo el contenido en esa dimensión, que en el caso del ejemplo serían las dos matrices de 2 x 5 que se muestran), luego para la segunda dimensión tenemos `1:` (todo el contenido luego desde el primer item, que para el ejemplo seria exceptuar la primer linea de cada grupo), y finalmente para la tercer dimensión tenemos `2` (el elemento en la posición dos, en el caso del ejemplo nos quedamos con el 7 y con el 17).

Veamos progresivamente esto, y al final el detalle de cómo luego de recortar las distintas dimensiones de la matriz original, el resultado queda con su propia "forma":

Cell 37

```
matriz
```

```
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9]],

       [[10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]]])
```

Cell 38

```
matriz[:]
```

```
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9]],

       [[10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]]])
```

Cell 39

```
matriz[:, 1:]
```

```
array([[[ 5,  6,  7,  8,  9]],

       [[15, 16, 17, 18, 19]]])
```

Cell 40

```
matriz[:, 1:, 2]
```

```
array([[ 7],
       [17]])
```

Cell 41

```
matriz[:, 1:, 2].shape
```

```
(2, 1)
```

Cuando hacemos *slicing* de arreglos de Numpy tenemos que tener un gran detalle en cuenta: a diferencia del mismo procedimiento en las listas integradas de Python, donde efectivamente tenemos una copia de la lista original, en el caso de NumPy lo que obtenemos es una "vista" del arreglo original.

Esta vista es afectada si modificamos el arreglo original, y viceversa. Esto es algo que hace NumPy en pos de la velocidad de procesamiento, ya que evita estar copiando objetos en memoria todo el tiempo.

Veamos este comportamiento. Primero un ejemplo usando las listas integradas en Python, para resaltar el efecto "copia" de las mismas:

Cell 42

```
l1 = [1, 2, 3, 4]
l2 = l1[:2]
l2
```

```
[1, 2]
```

Cell 43

```
l1[0] = 7
l2
```

```
[1, 2]
```

Cell 44

```
l2[1] = 9
l1
```

```
[7, 2, 3, 4]
```

Y ahora el caso de los arreglos de NumPy, con el comportamiento de las vistas:

Cell 45

```
a1 = np.array([1, 2, 3, 4])
a2 = a1[:2]
a2
```

```
array([1, 2])
```

Cell 46

```
a1[0] = 7
a2
```

```
array([7, 2])
```

Cell 47

```
a2[1] = 9
a1
```

```
array([7, 9, 3, 4])
```

1.3. Indización avanzada

NumPy ofrece una funcionalidad aún más interesante que lo que recién vimos sobre indizar con varias dimensiones al mismo tiempo: ¡podemos también indizar usando otros arreglos! Esto explota en distintas funcionalidades que vemos en esta sección.

Si el arreglo-índice tiene números, esos números indicarán la posición de los elementos que queremos del arreglo que estamos indizando.

En el siguiente ejemplo usamos esta funcionalidad para sacar los cuadrados de las posiciones 2, 3, 15 y 7:

CELL 48

```
cuadrados = np.arange(20) ** 2
cuadrados
```

```
array([  0,   1,   4,   9,  16,  25,  36,  49,  64,  81, 100, 121, 144,
       169, 196, 225, 256, 289, 324, 361])
```

CELL 49

```
índice = [2, 3, 15, 7]
cuadrados[índice]
```

```
array([  4,   9, 225,  49])
```

Vemos que no hace falta que el índice sea un arreglo de NumPy, pero obviamente eso también es soportado.

Tengamos en cuenta que los índices pueden ser repetidos (lo cual es útil en estadística para hacer muestreos eficientemente, por ejemplo):

CELL 50

```
cuadrados[[0, 1, 2, 3, 2, 1, 0]]
```

```
array([0, 1, 4, 9, 4, 1, 0])
```

Por otro lado, si el arreglo-índice está compuesto por booleanos, los mismos indicarán qué elementos elegimos del arreglo que estamos indizando (en este caso el índice tiene que tener el mismo largo que el arreglo indizado).

CELL 51

```
nros = np.array([0, 1, 2, 3, 4])
índice = [True, False, False, True, False]
nros[índice]
```

```
array([0, 3])
```

Esto es muy útil para elegir elementos de un arreglo en función de si cumplen alguna condición (lo cual en muchos contextos se denomina *máscara*).

Veamos un ejemplo donde tenemos muchos números y queremos calcular el logaritmo sólo de los positivos (claro que lo podríamos hacer con una *list comprehension* eliminando a los negativos y luego calculando el logaritmo de cada uno, pero recordemos que la idea es siempre quedarnos adentro de NumPy, para aprovechar al máximo la potencia de esta biblioteca).

CELL 52

```
nros = np.random.randint(-10, 10, 20)
nros
```

```
array([ -2,   3,   9,  -4,  -1,  -1,   0,   3,  -6,   0,   6,   2,   2,
         3,   4,  -6,   2,   6,  -5, -10])
```

CELL 53

```
cuales_positivos = nros > 0
cuales_positivos
```

```
array([False,  True,  True, False, False, False, False,  True, False,
       False,  True,  True,  True,  True,  True, False,  True,  True,
       False, False])
```

CELL 54

```
positivos = nros[cuales_positivos]
positivos
```

```
array([3, 9, 3, 6, 2, 2, 3, 4, 2, 6])
```

CELL 55

```
np.log(positivos)
```

```
array([1.09861229, 2.19722458, 1.09861229, 1.79175947, 0.69314718,
       0.69314718, 1.09861229, 1.38629436, 0.69314718, 1.79175947])
```

NumPy también nos da mecanismos para trabajar con arreglos correlacionados (esto es, dos o más arreglos donde cada elemento de un arreglo tiene una relación con el elemento de la misma posición en los otros arreglos).

Por ejemplo, podemos tener unas mediciones para graficar, con los valores del eje `x` en un arreglo y los correspondientes del eje `y` en otro. Por algún motivo necesitamos ordenar los puntos del eje `x`, pero obviamente debemos mantener la relación con los valores correspondientes del eje `y`.

Para ello usaremos la función `np.argsort`, que nos devuelve un arreglo-índice que ordenaría el arreglo indicado, y luego usamos ese índice con ambos arreglos `x` e `y`:

CELL 56

```
x = np.array([7, 3, 5, 9, 0, 6, 4, 1, 2, 8])
y = np.array([-8,  1, -3, -2,  3,  6,  5, -3,  2,  9])
índice = np.argsort(x)
índice  # este nos dejaría a X ordenado
```

```
array([4, 7, 8, 1, 6, 2, 5, 0, 9, 3])
```


VERSIÓN PRELIMINAR

Cell 57

```
x[índice]
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Cell 58

```
y[índice]
```

```
array([ 3, -3,  2,  1,  5, -3,  6, -8,  9, -2])
```

Vemos que al indizar x obtenemos el arreglo ordenado, y al indizar y obtenemos los valores correspondientes según el nuevo orden de x (se sigue manteniendo la relación entre los elementos de x e y para cada posición).

En todos estos casos de indización avanzada tenemos que prestar atención al hecho de que aunque arrancamos con un arreglo y terminamos en un arreglo, no hay relación entre estos dos (a diferencia de cuando hacíamos slicing, que teníamos una vista). En definitiva no es más que una forma expresiva y eficiente de indizar el arreglo por posición varias veces. Es por esto que si modificamos el arreglo que tenemos como resultado no estaremos modificando el arreglo original.

1.4. Broadcasting

El término *broadcasting* (que podríamos traducir como "transmisión") describe como NumPy trata durante las operaciones aritméticas a los arreglos que tienen distintas formas.

Porque cuando los arreglos tienen la misma forma es sencillo, los elementos mapean uno a uno y la operación se realiza entre los correspondientes:

Cell 59

```
x = np.array([0, 1, 2, 3, 4])
y = np.array([7, 6, 5, 4, 3])
x + y
```

```
array([7, 7, 7, 7, 7])
```

Cell 60

```
x * y
```

```
array([ 0,  6, 10, 12, 12])
```

Las reglas se complican (y el concepto de broadcasting aparece) cuando los arreglos **no** tienen el mismo tamaño.

Sin embargo, no cualquier combinación funciona. Cuando NumPy opera sobre dos arreglos compara la cantidad de elementos en cada dimensión (arrancando por las últimas), y en cada caso se tiene que cumplir una de dos condiciones:

- ambas dimensiones tienen la misma cantidad de elementos: en este caso se opera entre

cada elemento de esa dimensión

- alguna de las dimensiones tiene sólo un elemento: en este caso se "estira" esa dimensión (se repite ese único elemento para igualar la otra dimensión)

El ejemplo más sencillo de esto es cuando operamos aritméticamente entre un arreglo y un escalar:

CELL 61

```
x = np.array([0, 1, 2, 3, 4])
x * 2
```

```
array([0, 2, 4, 6, 8])
```

La operación que realizó allí es equivalente a haber repetido ese escalar la cantidad suficiente de veces, similar a lo que mostramos a continuación:

CELL 62

```
x = np.array([0, 1, 2, 3, 4])
x * np.array([2, 2, 2, 2, 2])
```

```
array([0, 2, 4, 6, 8])
```

El *broadcasting*, entonces, provee una forma de vectorizar operaciones de arreglos, de manera que los loops occurran en C en vez de Python. Y hace eso sin realizar copias innecesarias de los datos, lo que normalmente lleva a implementaciones muy eficientes de los algoritmos (hay casos, sin embargo, donde el broadcasting lleva a un uso ineficiente de la memoria y termina ralentizando el cálculo).

Veamos un ejemplo donde broadcasting funciona porque las dimensiones tienen la misma cantidad de elementos.

Supongamos un sensor óptico (pero simplificando, en vez de tener una matriz de 4096x4096, tenemos una matriz de 2x2, cada pixel con los tres valores correspondientes a RGB), Ahora, lo que queremos es aplicar un efecto donde reducimos el rojo muchísimo, dejando el verde y el azul en los valores originales (nuestro filtro tiene una sola dimensión y tres valores).

Prestemos atención al comentario luego de pedir cada forma, porque ahí estamos alineando las dimensiones arrancando por la última, que es como NumPy las compara. Entonces broadcasting funciona porque ambos arreglos tienen la dimensión con la misma cantidad de elementos.

CELL 63

```
foto = np.arange(12).reshape((2, 2, 3))  # 2x2 pixeles, x 3 valores para RGB
foto
```

```
array([[[ 0,  1,  2],
        [ 3,  4,  5]],

       [[ 6,  7,  8],
        [ 9, 10, 11]]])
```

CELL 64

```
filtro = np.array([.01, 1., 1.])  # solamente bajamos rojo muchísimo
filtro
```

```
array([0.01, 1.  , 1.  ])
```

CELL 65

```
# mostramos las formas alineadas a la izquierda
print("foto  : {:>6s}".format(' '.join(str(x) for x in foto.shape)))
print("filtro: {:>6s}".format(' '.join(str(x) for x in filtro.shape)))
```

```
foto  :  2 2 3
filtro:      3
```

CELL 66

```
foto * filtro
```

```
array([[[ 0.  ,  1.  ,  2.  ],
        [ 0.03,  4.  ,  5.  ]],

       [[ 0.06,  7.  ,  8.  ],
        [ 0.09, 10.  , 11.  ]]])
```

Entonces, como la última dimensión tiene tres elementos en cada caso, multiplica los tres valores del filtro por los tres valores para cada uno de los cuatro píxeles de la foto (va haciendo lo mismo en las otras dimensiones, porque "filtro" sólo tenía una.

Como vimos al principio, cuando el escalar se repetía varias veces en un arreglo monodimensional para ser multiplicado por el otro arreglo con la misma forma, acá sucede lo mismo. No es tan fácil de ver, sin embargo, pero podemos pedirle a NumPy que nos muestre cómo quedaría `filtro` si lo brodcasteamos a la forma de la foto:

CELL 67

```
np.broadcast_to(filtro, foto.shape)
```

```
array([[[0.01, 1.  , 1.  ],
        [0.01, 1.  , 1.  ]],

       [[0.01, 1.  , 1.  ],
        [0.01, 1.  , 1.  ]]])
```

Veamos ahora un caso donde ambos arreglos tienen más de una dimensión:

CELL 68

```
nros = np.arange(8).reshape((2, 4))
nros
```

```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

Cell 69

```
nros.shape
```

```
(2, 4)
```

Cell 70

```
factor = np.array([[3], [5]])
factor
```

```
array([[3],
       [5]])
```

Cell 71

```
factor.shape
```

```
(2, 1)
```

En este caso también aplica broadcasting, porque cada par de dimensiones compara bien: arrancando desde atrás (que es lo que hace NumPy) `nros` tiene 4 elementos y `factor` tiene 1 (válido, porque una de las dos tiene un elemento), y luego la anteúltima dimensión en cada caso tiene 2 elementos (válido, porque tienen la misma cantidad).

Cell 72

```
nros * factor
```

```
array([[ 0,  3,  6,  9],
       [20, 25, 30, 35]])
```

Vemos que al operar, la dimensión de `factor` que tenía un elemento se "estira" hasta cubrir los cuatro elementos del `nros`, entonces nos queda que 0, 1, 2 y 3 multiplican cada uno por 3, mientras que 4, 5, 6 y 7 multiplican cada uno por 5.

Cerremos con un ejemplo más complicado:

Cell 73

```
n1 = np.arange(12).reshape((2, 1, 2, 3))
n1
```

```
array([[[[ 0,  1,  2],
         [ 3,  4,  5]]],


       [[[ 6,  7,  8],
         [ 9, 10, 11]]]])
```

CELL 74

```
n2 = np.array([100, 200, 300, 400, 500, 600]).reshape((2, 1, 3))
n2
```

```
array([[[100, 200, 300]],

       [[400, 500, 600]]])
```

En este caso también broadcasting va a funcionar: de atrás para adelante tenemos 3 y 3 (iguales), 2 y 1 (uno es 1), 1 y 2 (uno es 1), y dejamos de comparar porque para un arreglo se nos acabaron las dimensiones.

CELL 75

```
n1 + n2
```

```
array([[[[100, 201, 302],
         [103, 204, 305]],

        [[400, 501, 602],
         [403, 504, 605]]],


       [[[106, 207, 308],
         [109, 210, 311]],

        [[406, 507, 608],
         [409, 510, 611]]]])
```

1.5. Vectores y matrices

Venimos hablando de arreglos, tanto unidimensionales como multidimensionales, que son estructuras genéricas que NumPy nos ofrece para hacer un montón de cosas. Pero nosotros en álgebra lineal tenemos los conceptos de vectores y matrices, que sí, pueden ser considerados arreglos de determinadas dimensiones, pero tienen sus semánticas específicas.

Veamos entonces qué podemos hacer si miramos y usamos los arreglos desde este ángulo.

Arrancamos con la advertencia de que el operador * es para multiplicar los elementos en sí, cuyo uso ya vimos más arriba, y aplican las reglas de broadcasting que también comentamos. Algunos ejemplos:

CELL 76

```
nros = np.array([0, 1, 2, 3, 4])
nros * 3
```

```
array([ 0,  3,  6,  9, 12])
```

CELL 77

```
nros1 = np.array([2, 3, 4])
nros2 = np.array([10, 0, 20])
nros1 * nros2
```

```
array([20,  0, 80])
```

CELL 78

```
mat = np.array([[1, 1, 1], [2, 2, 2], [3, 3, 3]])
nros = np.array([10, 20, 30])
mat * nros
```

```
array([[10, 20, 30],
       [20, 40, 60],
       [30, 60, 90]])
```

Por otro lado, si queremos "multiplicación de matrices", desde Python 3.5 tenemos el operador arroba (@) justamente pensado para esta operación (antes teníamos que usar la función `np.matmul`).

Veamos distintos ejemplos usando esta operación, escalando en complejidad. Si la operación es entre dos vectores, el resultado es la suma de la multiplicación entre sí de los escalares de cada posición (tengamos en cuenta que los dos vectores deben tener el mismo largo)

CELL 79

```
nros1 = np.array([2, 3, 4])
nros2 = np.array([10, 0, 20])
nros1 @ nros2
```

```
100
```

Si multiplicamos una matriz por un vector, la regla es que el largo del vector tiene que coincidir con la cantidad de columnas de la matriz, y el resultado será un vector de largo igual a la cantidad de filas de la matriz, y en cada posición la suma del producto los escalares del vector y esa fila de la matriz:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} a_{11}b_1 + a_{12}b_2 + \cdots + a_{1n}b_n \\ a_{21}b_1 + a_{22}b_2 + \cdots + a_{2n}b_n \\ \vdots \\ a_{m1}b_1 + a_{m2}b_2 + \cdots + a_{mn}b_n \end{bmatrix}$$

Probemos eso en numpy:

CELL 80

```
mat = np.array([[30, 10], [40, 40], [50, 50]])  # 2 columnas
mat
```

```
array([[30, 10],
       [40, 40],
       [50, 50]])
```

CELL 81

```
vec = np.array([1, 2])  # largo 2
vec
```

```
array([1, 2])
```

CELL 82

```
mat @ vec  # el resultado es largo 2 como el vector
```

```
array([ 50, 120, 150])
```

Para el caso bien genérico de multiplicar dos matrices, la regla a tener en cuenta es que el número de columnas de la primer matriz sea igual al número de filas de la segunda, y el resultado será otra matriz (con el número de filas de la primera y la cantidad de columnas de la segunda) donde cada elemento será:

$$c_{ij} = \sum_{r=1}^{n} a_{ir} b_{rj}$$

Entonces en el caso genérico de $\mathbb{AB} = \mathbb{C}$, si $\mathbb{A}$ tiene `m` filas y `n` columnas, $\mathbb{B}$ deberá tener `n` filas (y cualquier cantidad de columnas, digamos `p`); entonces el resultado $\mathbb{C}$ será una matriz de `m` filas y `p` columnas, donde:

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{np} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + \cdots + a_{1n}b_{n1} & \cdots & a_{11}b_{1p} + \cdots + a_{1n}b_{np} \\ \vdots & \ddots & \vdots \\ a_{m1}b_{11} + \cdots + a_{mn}b_{n1} & \cdots & a_{m1}b_{1p} + \cdots + a_{mn}b_{np} \end{bmatrix}$$

En código:

CELL 83

```
mat1 = np.array([[1, 1, 1], [2, 2, 2]])
mat1
```

```
array([[1, 1, 1],
       [2, 2, 2]])
```

CELL 84

```
mat2 = np.array([[30, 10], [40, 40], [50, 50]])
mat2
```

```
array([[30, 10],
       [40, 40],
       [50, 50]])
```

CELL 85

```
mat1 @ mat2
```

```
array([[120, 100],
       [240, 200]])
```

Más allá de todas estas operaciones entre matrices y/o vectores, también hay otros cálculos que generalmente se realizan en álgebra lineal para una matriz, y que por supuesto podemos hacer facilmente en NumPy. Veamos algunos de ellos…

Podemos calcular la determinante:

CELL 86

```
mat1 = np.array([[1, 2], [3, 4]])
np.linalg.det(mat1)
```

```
-2.0000000000000004
```

CELL 87

```
mat2 = np.array([[6, 2, 3], [1, -2, 0], [12, 2, -1]])
np.linalg.det(mat2)
```

```
92.00000000000001
```

Las inversas de esas mismas matrices:

CELL 88

```
np.linalg.inv(mat1)
```

```
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

CELL 89

```
np.linalg.inv(mat2)
```

```
array([[ 0.02173913,  0.08695652,  0.06521739],
       [ 0.01086957, -0.45652174,  0.0326087 ],
       [ 0.2826087 ,  0.13043478, -0.15217391]])
```

O su normas, en toda su diversidad; no vamos a mostrar todas aquí, se puede explorar el item correspondiente en la Referencia de NumPy [3], pero mostremos como ejemplo unas de las más usadas, la Frobenius 2:

CELL 90

```
np.linalg.norm(mat1)
```

```
5.477225575051661
```

CELL 91

```
np.linalg.norm(mat2)
```

```
14.247806848775006
```

E incluso podemos resolver un sistema de ecuaciones lineales con varias incógnitas.

Por ejemplo resolvamos el siguiente sistema con tres ecuaciones y tres incógnitas:

$$\begin{aligned} 3x + y &= -3 \\ 4y - z &= 5 \\ x + 3y - 2z &= -7 \end{aligned}$$

Lo primero es cargar una matriz con las ecuaciones y un vector con los resultados, de manera que toda la resolución sea:

$$\mathbb{A} \cdot \boldsymbol{x} = \boldsymbol{b}$$

En código:

CELL 92

```
A = np.array([[3, 1, 0], [0, 4, -1], [1, 3, -2]])
A
```

```
array([[ 3,  1,  0],
       [ 0,  4, -1],
       [ 1,  3, -2]])
```

CELL 93

```
b = np.array([-3, 5, -7])
b
```

```
array([-3,  5, -7])
```

Luego, NumPy nos calculará $\boldsymbol{x}$:

CELL 94

```
x = np.linalg.solve(A, b)
x
```

```
array([-2.,  3.,  7.])
```

Podemos validar facilmente el resultado:

VERSIÓN PRELIMINAR

Cell 95

```
A @ x
```

```
array([-3.,  5., -7.])
```

Tengamos en cuenta sin embargo que estamos lidiando con punto flotante, y como vimos en el capítulo correspondiente **??** podemos tener errores mínimos en las operaciones que hacen que "comparar por igual" sea engañoso:

Cell 96

```
A @ x == b
```

```
array([False,  True, False])
```

Cell 97

```
(A @ x)[0]
```

```
-3.0000000000000004
```

Cell 98

```
(A @ x)[0] == .3
```

```
False
```

Para seguir con este tema y profundizarlo, les recomendamos el Capítulo de Ecuaciones Algebraicas **??**.

Parte III
Temas específicos

En esta Parte desarrollaremos capítulos que abordan temas específicos de aplicación de herramientas de Python. Cada capítulo está autocontenido, de forma que el lector o lectora puede acceder directamente al tema de interés sin realizar un recorrido secuencial de los capítulos, aunque serán utilizados conceptos y herramientas tratados en la Parte II.

Parte IV
Apéndices

A | Zen de Python

Incluimos aquí las frases traducidas correspondientes al Zen de Python [4].

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora* mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres son una gran idea, ¡tengamos más de esos!

Bibliografía

[1] URL: https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es.

[2] URL: https://numpy.org/doc/stable/reference/.

[3] URL: https://numpy.org/doc/stable/reference/generated/numpy.linalg.norm.html.

[4] Tim Peters. *The Zen of Python*. 19 de ago. de 2004. URL: https://www.python.org/dev/peps/pep-0020/.

VERSIÓN PRELIMINAR