IA PUCP - Diplomado de Desarrollo de Aplicaciones de Inteligencia Artificial
Python para Ciencia de Datos

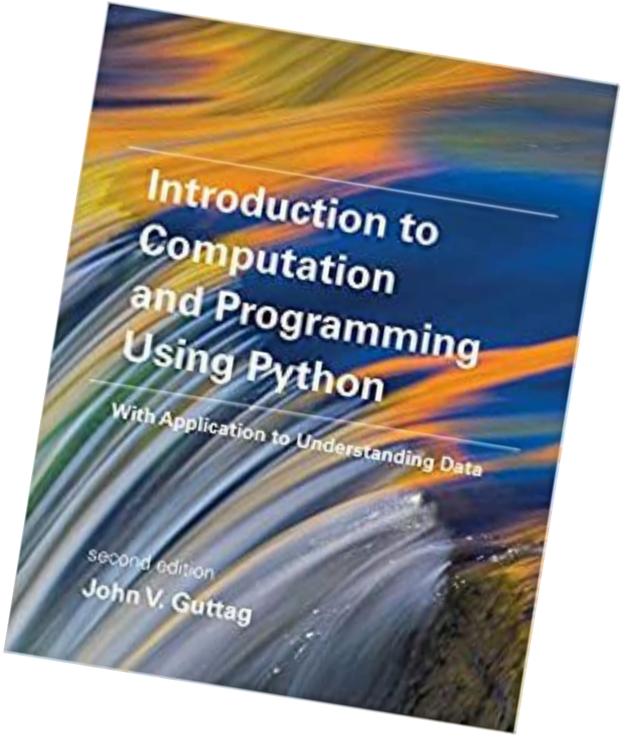
Introducción a Numpy

Contenido

- Programación Orientada a Objetos en Python
- Vectores en Numpy
- Matrices en Numpy
- Funcionalidades de Numpy

Ver más...

Guttag, John. ***Introduction to Computation and Programming Using Python: With Application to Understanding Data Second Edition.*** MIT Press, 2016. ISBN: 9780262529624.

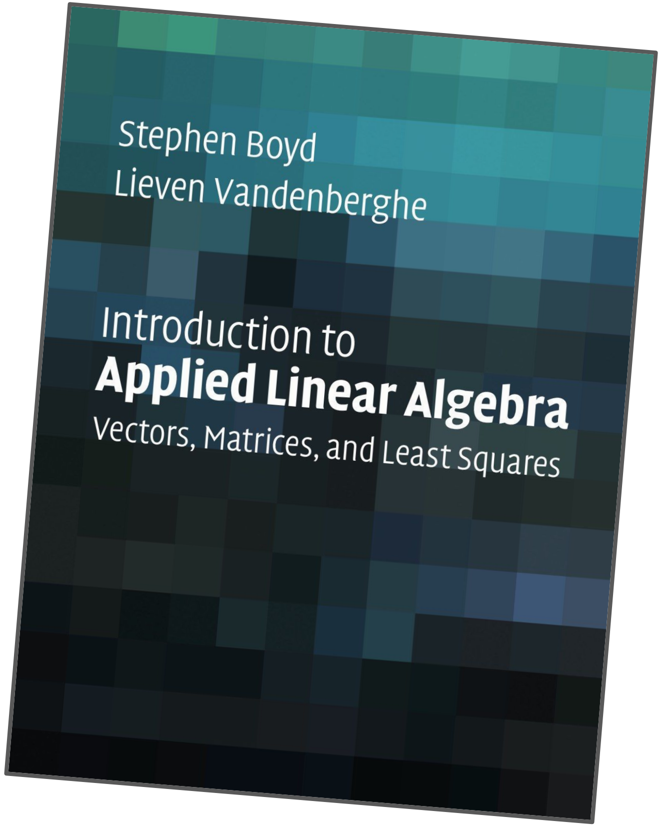


Ver más en

Boyd, S., & Vandenberghe, L. (2018). **Introduction to applied linear algebra: vectors, matrices, and least squares**. Cambridge university press.

PDF Gratuito en
http://vmls-book.stanford.edu/

Los slides están basados en
http://vmls-book.stanford.edu/vmls-slides.pdf



Objetos

- Python soporta varios tipos de data

```
1234   3.14159    "Hola"    [1, 5 ,7 ,11, 13]
{"LIM": "Lima", "MIA": "Miami"}
```

- cada uno de estos es un objeto que tiene:
    - tipo
    - representación de datos interna (primitivos o compuestos)
    - lista de procedimientos para interacción con el objeto

Basado en: Ana Bell, Eric Grimson, and John Guttag. 6.0001 Introduction to Computer Science and Programming in Python. Fall 2016. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu. License: Creative Commons BY-NC-SA. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/MIT6_0001F16_Lec1.pdf

Objetos

- Un objeto es una instancia de un tipo (podemos usar `type` para saber cuál)
  - 1234 es una instancia de `int`
  - "hello" es una instancia de `str`

Basado en: Ana Bell, Eric Grimson, and John Guttag. 6.0001 Introduction to Computer Science and Programming in Python. Fall 2016. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu. License: Creative Commons BY-NC-SA.
https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/MIT6_0001F16_Lec1.pdf

Programación Orientada a Objetos (1/2)

- Todo en Python es un objeto (y tiene un tipo)
- Se puede crear nuevos objetos de un tipo
- Se puede manipular objetos

Basado en: Ana Bell, Eric Grimson, and John Guttag. 6.0001 Introduction to Computer Science and Programming in Python. Fall 2016. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu. License: Creative Commons BY-NC-SA.
https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/MIT6_0001F16_Lec1.pdf

Programación Orientada a Objetos (2/2)

- Se puede destruir objetos
  - Explícitamente con el keyword `del`
  - Implícitamente “olvidando” al objeto
  - Python tiene “garbage collection”

Basado en: Ana Bell, Eric Grimson, and John Guttag. 6.0001 Introduction to Computer Science and Programming in Python. Fall 2016. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu. License: Creative Commons BY-NC-SA. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/MIT6_0001F16_Lec1.pdf

¿Qué son los objetos?

- Los objetos son una abstracción sobre la data que:
  - Tiene una representación interna
  - Tiene una interfaz para interactuar con otros objetos
    - A través de métodos (funciones/procedimientos)
    - Define comportamiento pero oculta la implementación

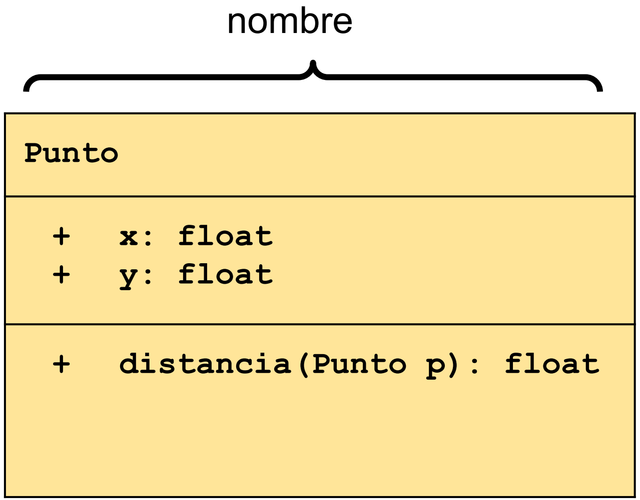
Ventajas de la Programación Orientada a Objetos (1/2)

- Empaquetar la data junto con las funciones que la procesan a través de interfaces bien definidas
- Desarrollo divide-y-vencerás
  - Implementar y probar el comportamiento de cada clase de manera separada
  - Mayor modularidad que reduce la complejidad

Ventajas de la Programación Orientada a Objetos (2/2)

- Las clases facilitan la reutilización de código
    - Muchos módulos de Python definen nuevas clases
    - Cada clase tiene un ‘entorno’ aislado
    - La herencia permite que subclases redefinan o extiendan comportamientos

Creando nuestros propios “tipos” o clases

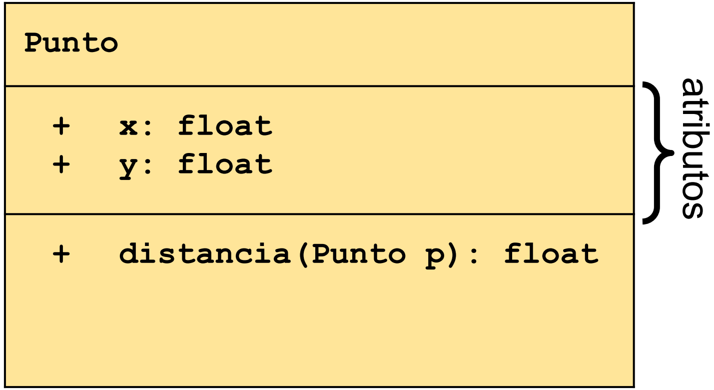



```
class Punto():
    def __init__(self, x,y)
        self.x = x
        self.y = y
    def distancia(self, p):
        x_diff_2 = (self.x-p.x)**2
        y_diff_2 = (self.y -p.y)**2
        return (x_diff_2+y_diff_2)**0.5

c = Punto(3,4)
origen = Punto(0,0)
print(c.x, origen.x)
```

Basado en: Ana Bell, Eric Grimson, and John Guttag. 6.0001 Introduction to Computer Science and Programming in Python. Fall 2016. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu. License: Creative Commons BY-NC-SA. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/MIT6_0001F16_Lec1.pdf

Empaquetando Data: Atributos




```
class Punto():
    def __init__(self, x,y)
        self.x = x
        self.y = y
    def distancia(self, p):
        x_diff_2 = (self.x-p.x)**2
        y_diff_2 = (self.y -p.y)**2
        return (x_diff_2+y_diff_2)**0.5

c = Punto(3,4)
origen = Punto(0,0)
print(c.x, origen.x)
```

Basado en: Ana Bell, Eric Grimson, and John Guttag. 6.0001 Introduction to Computer Science and Programming in Python. Fall 2016. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu. License: Creative Commons BY-NC-SA.
https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/MIT6_0001F16_Lec1.pdf

Empaquetando Data: Atributos

init es el constructor, es lo que se ejecutará al “instanciar” la clase

Punto
+ x: float + y: float
+ distancia(Punto p): float

atributos

```
class Punto():
    def __init__(self, x,y)
        self.x = x
        self.y = y
    def distancia(self, p):
        x_diff_2 = (self.x-p.x)**2
        y_diff_2 = (self.y -p.y)**2
        return (x_diff_2+y_diff_2)**0.5

c = Punto(3,4)
origen = Punto(0,0)
print(c.x, origen.x)
```

Empaquetando Data: Atributos

```
Punto
-----------------------------------
+   x: float
+   y: float
-----------------------------------
+   distancia(Punto p): float
```

atributos

```python
class Punto():
    def __init__(self, x,y)
        self.x = x
        self.y = y
    def distancia(self, p):
        x_diff_2 = (self.x-p.x)**2
        y_diff_2 = (self.y -p.y)**2
        return (x_diff_2+y_diff_2)**0.5

c = Punto(3,4)
origen = Punto(0,0)
print(c.x, origen.x)
```

Aquí estamos instanciando la clase Punto

Empaquetando Data: Atributos

Se pasa de manera automática

```
Punto
----------------
+   x: float
+   y: float
----------------
+   distancia(Punto p): float
```

atributos

```
class Punto():
    def __init__(self, x,y)
        self.x = x
        self.y = y
    def distancia(self, p):
        x_diff_2 = (self.x-p.x)**2
        y_diff_2 = (self.y -p.y)**2
        return (x_diff_2+y_diff_2)**0.5

c = Punto(3,4)
origen = Punto(0,0)
print(c.x, origen.x)
```

Basado en: Ana Bell, Eric Grimson, and John Guttag. 6.0001 Introduction to Computer Science and Programming in Python. Fall 2016. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu. License: Creative Commons BY-NC-SA. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/MIT6_0001F16_Lec1.pdf

Empaquetando Data: Atributos

Punto
+ x: float + y: float
+ distancia(Punto p): float

atributos

```
class Punto():
    def __init__(self, x,y)
        self.x = x
        self.y = y
    def distancia(self, p):
        x_diff_2 = (self.x-p.x)**2
        y_diff_2 = (self.y -p.y)**2
        return (x_diff_2+y_diff_2)**0.5

c = Punto(3,4)
origen = Punto(0,0)
print(c.x, origen.x)
```

El operador punto permite acceder a los atributos

Basado en: Ana Bell, Eric Grimson, and John Guttag. 6.0001 Introduction to Computer Science and Programming in Python. Fall 2016. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu. License: Creative Commons BY-NC-SA. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/MIT6_0001F16_Lec1.pdf

Empaquetando Funcionalidad: Métodos

```
Punto
------------------------------
+   x: float
+   y: float
------------------------------
+   distancia(Punto p): float
```

métodos

```python
class Punto():
    def __init__(self, x,y)
        self.x = x
        self.y = y
    def distancia(self, p):
        x_diff_2 = (self.x-p.x)**2
        y_diff_2 = (self.y -p.y)**2
        return (x_diff_2+y_diff_2)**0.5

c = Punto(3,4)
origen = Punto(0,0)
print(c.distancia(origen))
```

Empaquetando Funcionalidad: Métodos

Aquí declaramos el método distancia

```
Punto
---------------------------------
+   x: float
+   y: float
---------------------------------
+   distancia(Punto p): float
```

métodos

```python
class Punto():
    def __init__(self, x,y):
        self.x = x
        self.y = y
    def distancia(self, p):
        x_diff_2 = (self.x-p.x)**2
        y_diff_2 = (self.y -p.y)**2
        return (x_diff_2+y_diff_2)**0.5

c = Punto(3,4)
origen = Punto(0,0)
print(c.distancia(origen))
```

Empaquetando Funcionalidad: Métodos

```
Punto
---------------------------------
+   x: float
+   y: float
---------------------------------
+   distancia(Punto p): float
```

métodos

```python
class Punto():
    def __init__(self, x,y)
        self.x = x
        self.y = y
    def distancia(self, p):
        x_diff_2 = (self.x-p.x)**2
        y_diff_2 = (self.y -p.y)**2
        return (x_diff_2+y_diff_2)**0.5

c = Punto(3,4)
origen = Punto(0,0)
print(c.distancia(origen))
```

Aquí llamamos al método distancia

Vectores

- Un vector es una lista ordenada de números escritos como

$$\begin{bmatrix} -1.1 \\ 0.0 \\ 3.6 \\ -7.2 \end{bmatrix} \quad \text{or} \quad \begin{pmatrix} -1.1 \\ 0.0 \\ 3.6 \\ -7.2 \end{pmatrix}$$

o `(-1.1, 0, 3.6, -7.2)`

Vectores

- Un vector es una lista ordenada de números escritos como

$$\begin{bmatrix} -1.1 \\ 0.0 \\ 3.6 \\ -7.2 \end{bmatrix} \quad \text{or} \quad \begin{pmatrix} \boxed{-1.1} \\ 0.0 \\ 3.6 \\ -7.2 \end{pmatrix}$$

o `(-1.1, 0, 3.6, -7.2)`

Los números en la lista son elementos (entradas, coeficientes, componentes)

Vectores

- Un vector es una lista ordenada de números escritos como

$$\begin{bmatrix} -1.1 \\ 0.0 \\ 3.6 \\ -7.2 \end{bmatrix} \quad \text{or} \quad \begin{pmatrix} -1.1 \\ 0.0 \\ 3.6 \\ -7.2 \end{pmatrix}$$

o `(-1.1, 0, 3.6, -7.2)`

El número de elementos es el **tamaño** (dimensión, longitud) del vector

Vectores

- Un vector es una lista ordenada de números escritos como

$$\begin{bmatrix} -1.1 \\ 0.0 \\ \boxed{3.6} \\ -7.2 \end{bmatrix} \quad \text{or} \quad \begin{pmatrix} -1.1 \\ 0.0 \\ \boxed{3.6} \\ -7.2 \end{pmatrix}$$

4 elementos

o `(-1.1, 0, 3.6, -7.2)`

Este vector tiene tamaño 4, y la 3ra entrada es 3.6

Vectores

- Un vector es una lista ordenada de números escritos como

$$\begin{bmatrix} -1.1 \\ 0.0 \\ 3.6 \\ -7.2 \end{bmatrix} \quad \text{or} \quad \begin{pmatrix} \boxed{-1.1} \\ 0.0 \\ 3.6 \\ -7.2 \end{pmatrix}$$

o `(-1.1, 0, 3.6, -7.2)`

Los números son llamados escalares

Vectores

- Dos vectores a y b del mismo tamaño son iguales si $\mathbf{a_i = b_i}$ para todo $\mathbf{i}$ esto lo denotamos como $\mathbf{a=b}$

Numpy

Si bien las listas en Python permiten almacenar secuencias ordenadas de números, Numpy provee objetos más apropiados. Para poder usarlo tenemos que importar la librería

```python
import numpy
```

o más convenientemente

```python
import numpy as np
```

usaremos esta versión en el curso

Vectores en Numpy

- El objeto base para representar vectores en Numpy es el Array (arreglo)

```
x = np.array([-1.1, 0, 3.6, -7.2])
```

Pasamos los datos como una lista

Vectores en Numpy

- El objeto base para representar vectores en Numpy es el Array (arreglo)

```
x = np.array([-1.1, 0, 3.6, -7.2])

print(type(x))
```

Vectores en Numpy

- El objeto base para representar vectores en Numpy es el Array (arreglo)

```
x = np.array([-1.1, 0, 3.6, -7.2])

print(type(x))
```

```
numpy.ndarray
```

El tipo de dato es nd-array que significa arreglo n-dimensional. En 1 dimensión es un vector, en 2 dimensiones será una matriz

Vectores en Numpy

- El objeto base para representar vectores en Numpy es el Array (arreglo)

```
x = np.array((-1.1, 0, 3.6, -7.2))

print(type(x))
```

```
numpy.ndarray
```

También podemos usar una tupla

Vectores en Numpy

- El objeto base para representar vectores en Numpy es el Array (arreglo)

```
x = np.array([-1.1, 0, 3.6, -7.2])

print(x.shape)
```

```
(4,)
```

Podemos ver el atributo .shape para saber el tamaño de un vector (devuelve una tupla)

Vectores en Numpy

- El objeto base para representar vectores en Numpy es el Array (arreglo)

```
x = np.array([-1.1, 0, 3.6, -7.2])

print(len(x))
```

```
4
```

Podemos usar la función `len` también en el caso de los vectores, devuelve un entero

Vectores en Numpy

- El objeto base para representar vectores en Numpy es el Array (arreglo)

```
x = np.array([-1.1, 0, 3.6, -7.2])

print(x[2])
```

```
3.6
```

Para obtener la 3ra entrada usamos el índice 2, porque se indexa desde 0.

Ceros, unos y vectores unitarios

- Creando un vector $0_{n=5}$

```
n = 5
np.zeros(n)
```

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

- Creando un vector $1_{n=5}$

```
n = 5
np.ones(n)
```

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

Sparsity

- Un vector es esparzo (sparse) si muchas de sus entradas son 0
- Puede ser almacenado y manipulado eficientemente en un computadora
- **nnz(x)** es el número de entradas que son diferentes de 0

```
a = np.zeros(100)
a[10] = 2
np.count_nonzero(a)
```

```
1
```

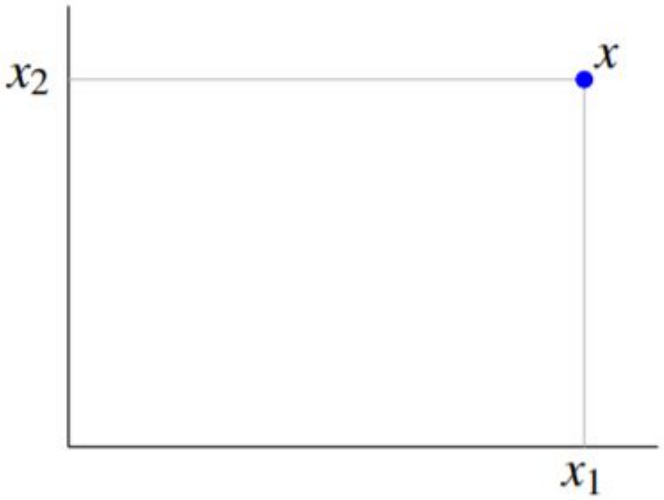
En numpy usamos la función `np.count_nonzero()` para medir *sparcity*.

Ubicación o desplazamiento en 2d o 3d

- Un vector 2-dimensional puede tener múltiples interpretaciones, dos representaciones comunes son la **ubicación** y **desplazamiento** en 2d

Ubicación o desplazamiento en 2d o 3d

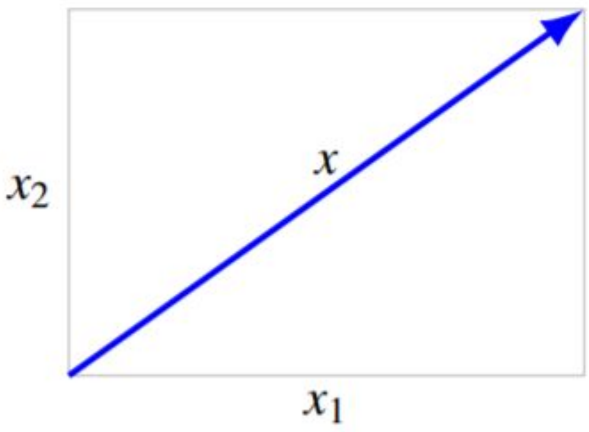
- Un vector 2-dimensional puede representar la ubicación o desplazamiento en 2d



Un vector $x = (x_1, x_2)$ puede tener la interpretación de ubicación, si fuera así, las operaciones vectoriales que realicemos cobrarán significados en esa dirección

Ubicación o desplazamiento en 2d o 3d

- Un vector 2-dimensional puede representar la ubicación o desplazamiento en 2d



Otra interpretación común para un vector $x = (x_1, x_2)$ es el desplazamiento

Otras representaciones/interpretaciones que pueden tener los vectores

- color: (R,G,B)
- cantidades de n diferentes recursos. Por ej. lista de materiales
- portafolio: las entradas indican cuántas acciones en n bienes
- flujo de capital: x_i es el pago en el periodo i
- audio: x_i es la presión acústica en el tiempo i (las muestras están espaciadas 1/44100 segundos)
- características: x_i es el valor de la i-ésima característica o atributo de una entidad
- cantidad de palabras: x_i es la cantidad de veces que la palabra i aparece en un documento

Vectores de cantidad de palabras

- Un documento corto

> **Word** count vectors are used **in** computer based **document** analysis. Each entry of the **word** count vector is the **number** of times the associated dictionary **word** appears **in** the **document**.

- Un diccionario pequeño (izquierda) y un vector de cantidad de palabras (derecha)

$$\begin{array}{l} \text{word} \\ \text{in} \\ \text{number} \\ \text{horse} \\ \text{the} \\ \text{document} \end{array} \qquad \begin{bmatrix} 3 \\ 2 \\ 1 \\ 0 \\ 4 \\ 2 \end{bmatrix}$$

- Los diccionarios usados en la práctica con mucho más grandes

Adición de vectores

- Los vectores n-dimensionales `a` y `b` pueden ser sumados, con la suma denotada por `a + b`
- Para obtener la suma, adicionar las entradas correspondientes:

$$\begin{bmatrix} 0 \\ 7 \\ 3 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 9 \\ 3 \end{bmatrix}$$

- La resta es similar

Adición de vectores en Numpy

```
a = np.array([1,2,3,4,5,6])
b = np.array([9,8,7,6,5,4])
a+b
```

```
array([10, 10, 10, 10, 10, 10])
```

```
a = np.array([1,2,3,4,5,6])
b = np.array([9,8,7,6,5,4])
a-b
```

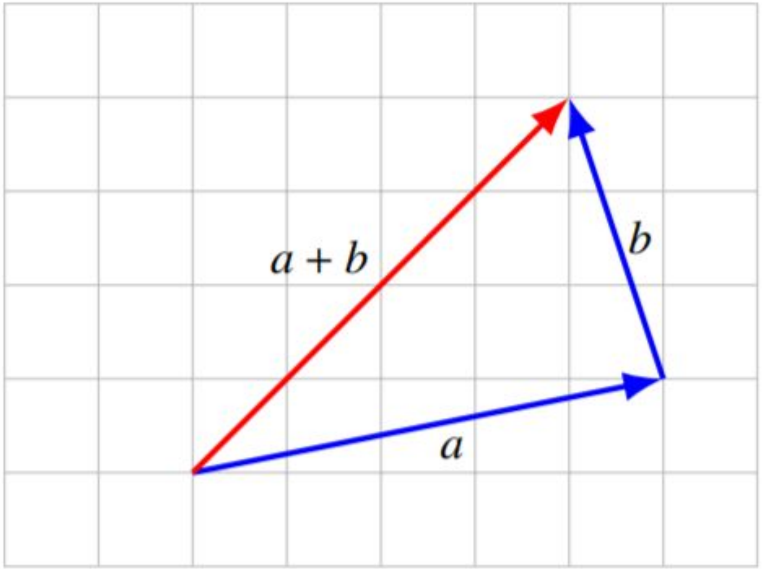
```
array([-8, -6, -4, -2,  0,  2])
```

Propiedades de la adición de vectores

- conmutatividad: `a + b = b +a`
- asociatividad: `(a+b)+c = a+(b+c)`
  ambos podrían ser reescritos como `a+b+c`
- `a+0 = 0+a = a`
- `a-a = 0`

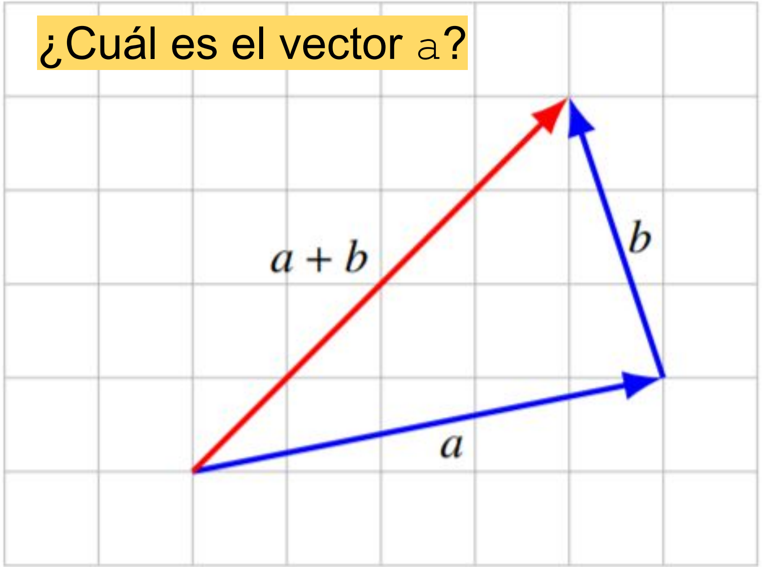
Sumando desplazamientos

Si los vectores 2-dimensionales **a** y **b** son desplazamientos, **a+b** es el desplazamiento total



Sumando desplazamientos

Si los vectores 2-dimensionales **a** y **b** son desplazamientos, **a+b** es el desplazamiento total



A) (5, 1)

B) (-1, 3)

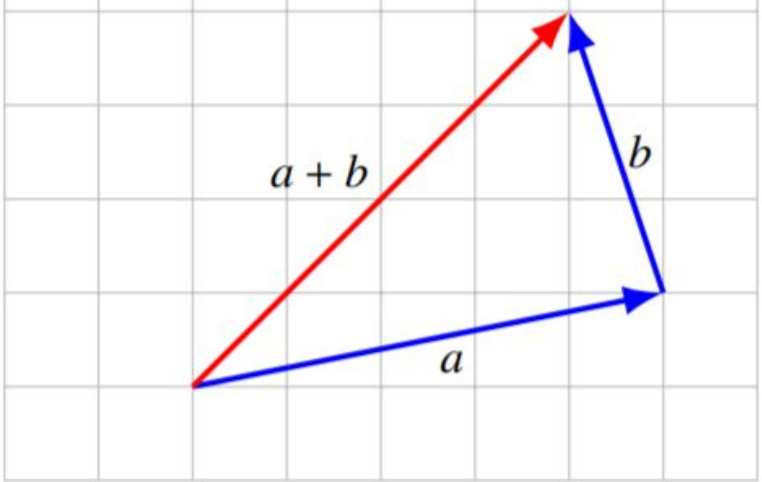
C) (4, 4)

D) (2, 2)

Sumando desplazamientos

Si los vectores 2-dimensionales **a** y **b** son desplazamientos, **a+b** es el desplazamiento total

¿Cuál es el vector a?



A) (5, 1)

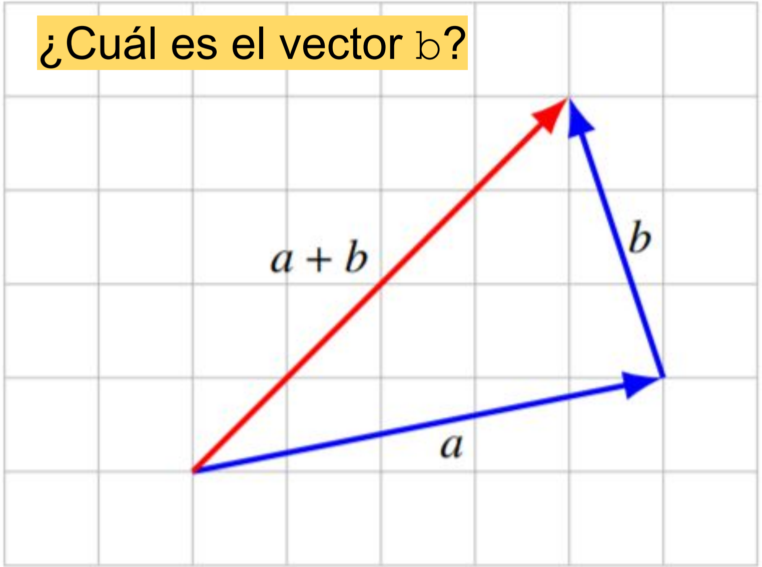
B) (-1, 3)

C) (4, 4)

D) (2, 2)

Sumando desplazamientos

Si los vectores 2-dimensionales **a** y **b** son desplazamientos, **a+b** es el desplazamiento total



A) (5, 1)

B) (-1, 3)

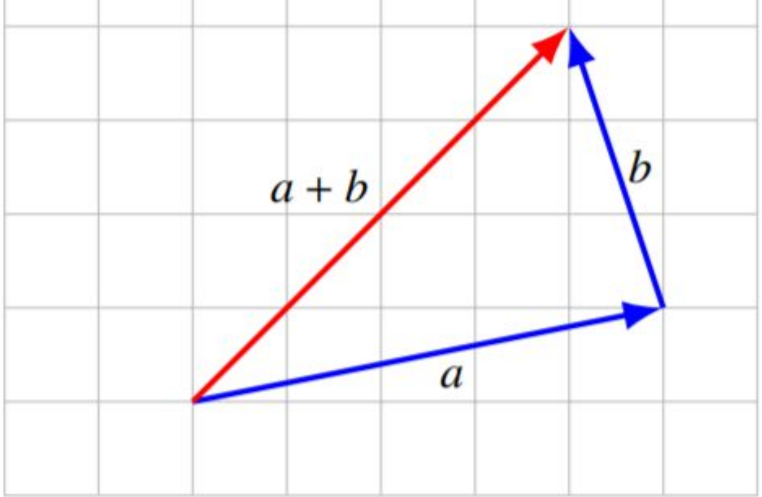
C) (4, 4)

D) (2, 2)

Sumando desplazamientos

Si los vectores 2-dimensionales **a** y **b** son desplazamientos, **a+b** es el desplazamiento total

¿Cuál es el vector b?



A) (5, 1)

B) (-1, 3)

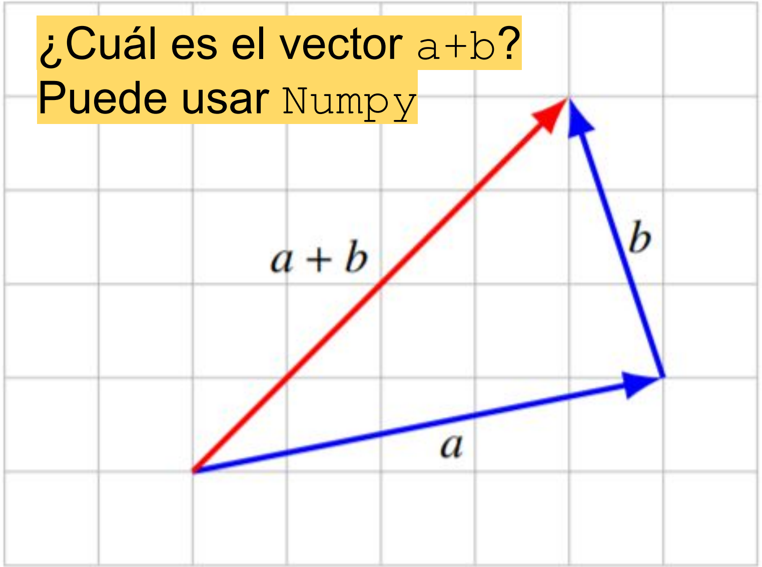
C) (4, 4)

D) (2, 2)

Sumando desplazamientos

Si los vectores 2-dimensionales **a** y **b** son desplazamientos, **a+b** es el desplazamiento total

¿Cuál es el vector `a+b`?
Puede usar `Numpy`



A) (5, 1)

B) (-1, 3)

C) (4, 4)

D) (2, 2)

Sumando desplazamientos

Si los vectores 2-dimensionales **a** y **b** son desplazamientos, **a+b** es el desplazamiento total

```
a = np.array([5,1])

b = np.array([-1,3])

print(a+b)
```

```
array([4,4])
```

A) (5, 1)

B) (-1, 3)

C) (4, 4)

D) (2, 2)

Sumando desplazamientos

Si los vectores 2-dimensionales **a** y **b** son desplazamientos, **a+b** es el desplazamiento total

Cuando la interpretación que damos a los vectores es el desplazamiento, la operación de suma se interpreta como desplazamiento total.

A) (5, 1)

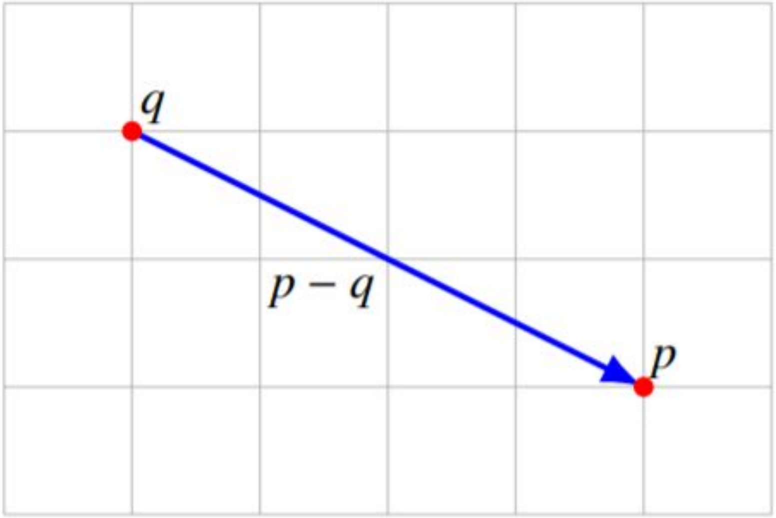
B) (-1, 3)

C) (4, 4)

D) (2, 2)

Sumando desplazamientos

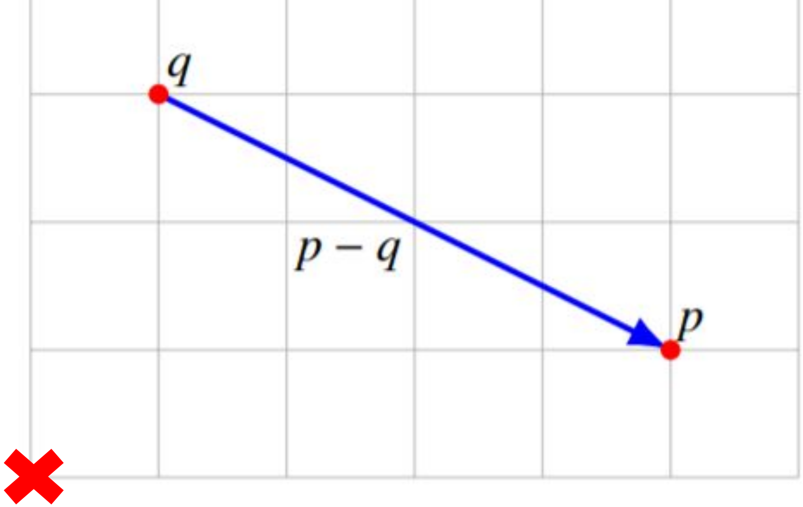
el desplazamiento del punto **q** al punto **p** es **p-q**



Sumando desplazamientos

el desplazamiento del punto **q** al punto **p** es **p-q**

¿Cuál es el vector p?



origen de coordenadas

A) (5, 1)

B) (1, 3)

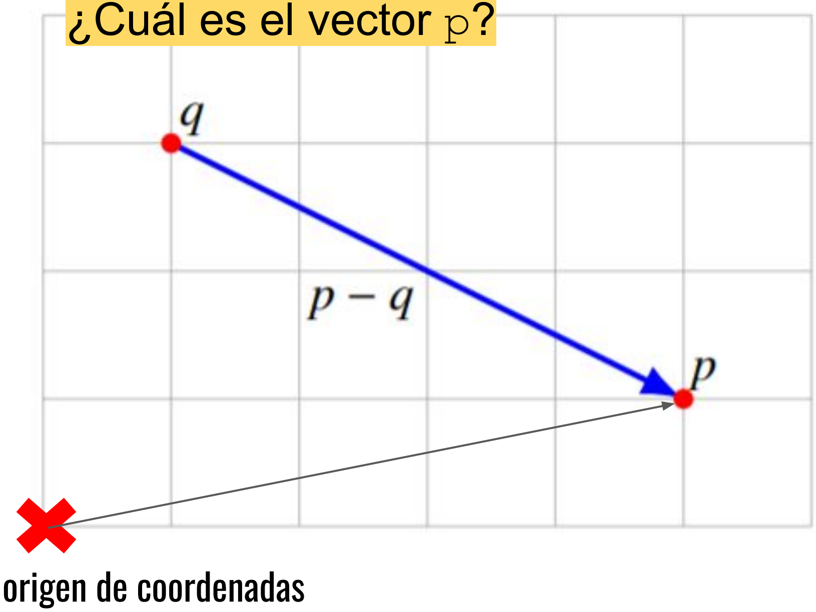
C) (4, -2)

D) (2, 2)

Sumando desplazamientos

el desplazamiento del punto **q** al punto **p** es **p-q**

¿Cuál es el vector $\mathtt{p}$?



A) (5, 1)

B) (1, 3)

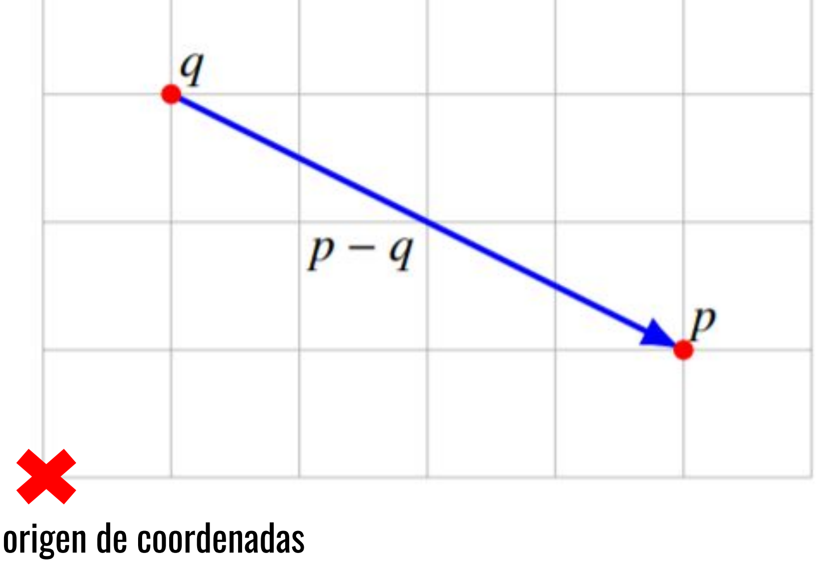
C) (4, -2)

D) (2, 2)

Sumando desplazamientos

el desplazamiento del punto **q** al punto **p** es **p-q**

¿Cuál es el vector q?



A) (5, 1)

B) (1, 3)

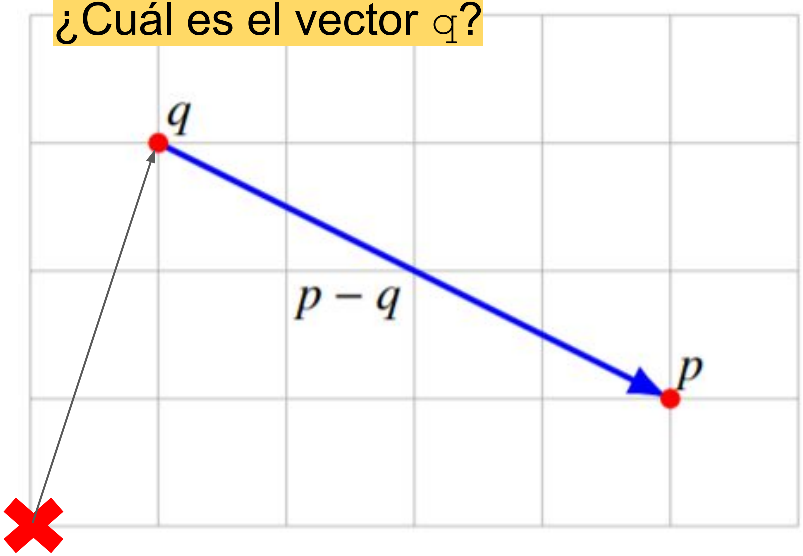
C) (4, -2)

D) (2, 2)

Sumando desplazamientos

el desplazamiento del punto **q** al punto **p** es **p-q**

¿Cuál es el vector q?



origen de coordenadas

A) (5, 1)

B) (1, 3)

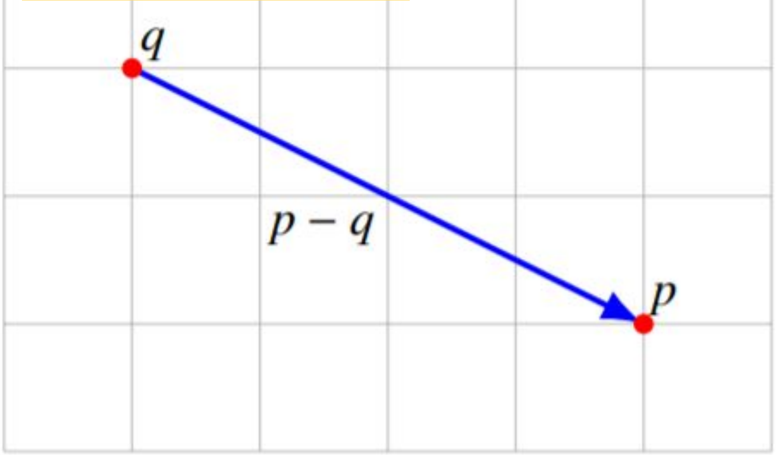
C) (4, -2)

D) (2, 2)

Sumando desplazamientos

el desplazamiento del punto **q** al punto **p** es **p-q**

¿Cuál es el vector `p-q`?
Puede usar Numpy



A) (5, 1)

B) (1, 3)

C) (4, -2)

D) (2, 2)

Sumando desplazamientos

el desplazamiento del punto **q** al punto **p** es **p-q**

```
a = np.array([5,1])

b = np.array([1,3])

print(a-b)
```

```
array([4,-2])
```

A) (5, 1)

B) (1, 3)

C) (4, -2)

D) (2, 2)

Sumando desplazamientos

el desplazamiento del punto **q** al punto **p** es **p-q**

Cuando la interpretación que damos a los vectores es la ubicación, la operación de resta se interpreta como desplazamiento.

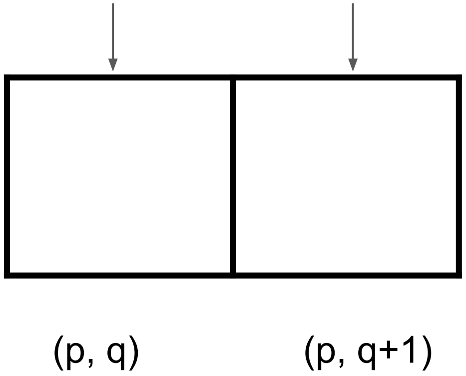
A) (5, 1)

B) (1, 3)

C) (4, -2)

D) (2, 2)

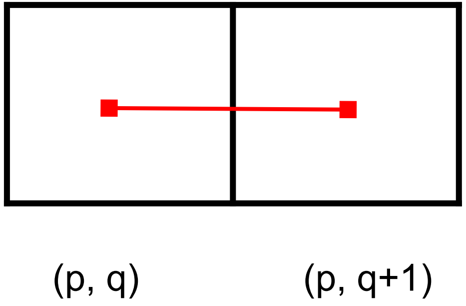
Distancia entre píxeles: horizontal



A) 1 pixel

B) 2 pixeles

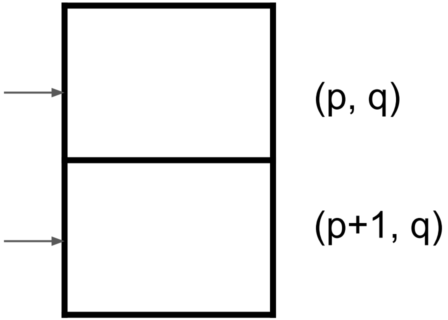
Distancia entre píxeles: horizontal



A) 1 pixel

B) 2 pixeles

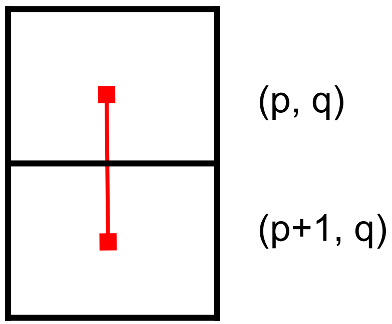
Distancia entre píxeles: vertical



A) 1 pixel

B) 2 pixeles

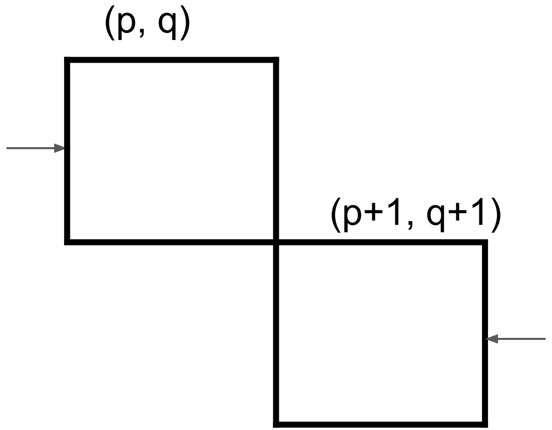
Distancia entre píxeles: vertical



A) 1 pixel

B) 2 pixeles

Distancia entre píxeles: Diagonal

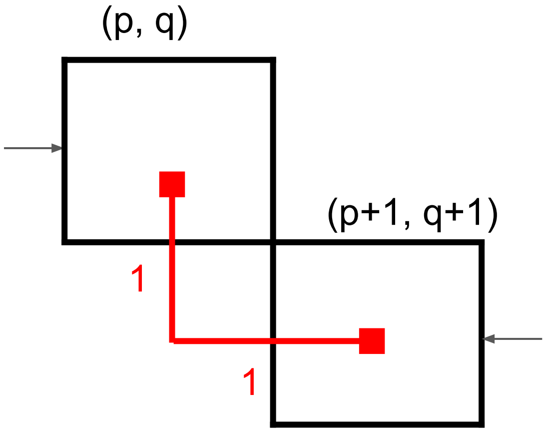


A) 1 pixel

B) $\sqrt{2}$ pixeles

C) 2 pixeles

Distancia entre píxeles: Diagonal



A) 1 pixel

B) √2 pixeles

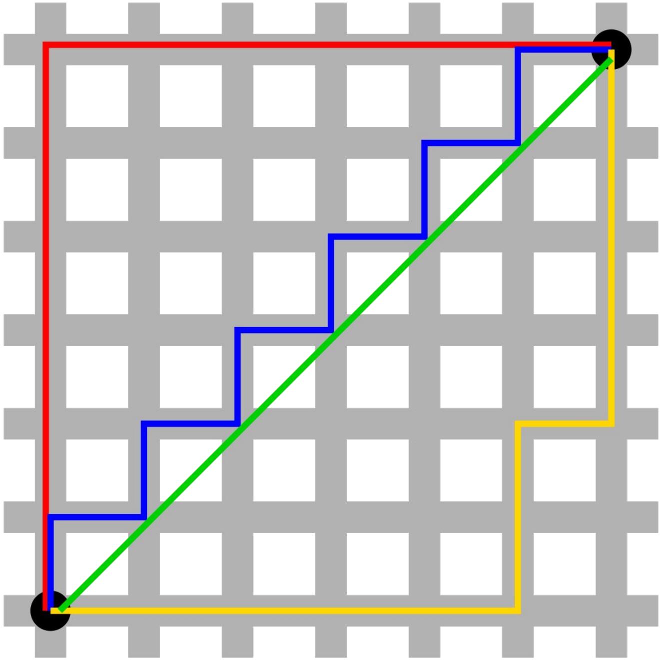
C) 2 pixeles

Distancia euclidiana

- La distancia que hemos visto hasta el momento se llama "euclidiana"
- Si tenemos dos píxeles **p** y **q** con coordenadas (*x, y*) e (*s, t*) entonces

$$\mathbf{D_{euc}}\,(\mathbf{p}, \mathbf{q}) = [(x-s)^2 + (y-t)^2]^{1/2}$$

Distancia City-Block / Manhattan / o D_4



- Si tenemos dos píxeles **p** y **q** con coordenadas (*x*, *y*) e (*s*, *t*) entonces

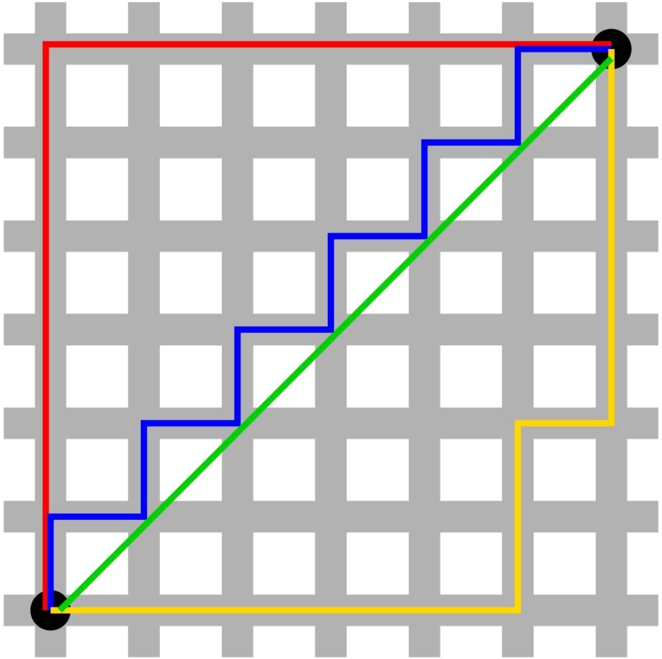
$$\mathbf{D}_{\mathbf{cityblock}}(\mathbf{p}, \mathbf{q}) = |x-s| + |y-t|$$

https://en.wikipedia.org/wiki/Taxicab_geometry

Distancia Manhattan vs Distancia Euclidiana

https://www.pinterest.co.uk/pin/572660908851051781/

Distancia City-Block / Manhattan / o D_4



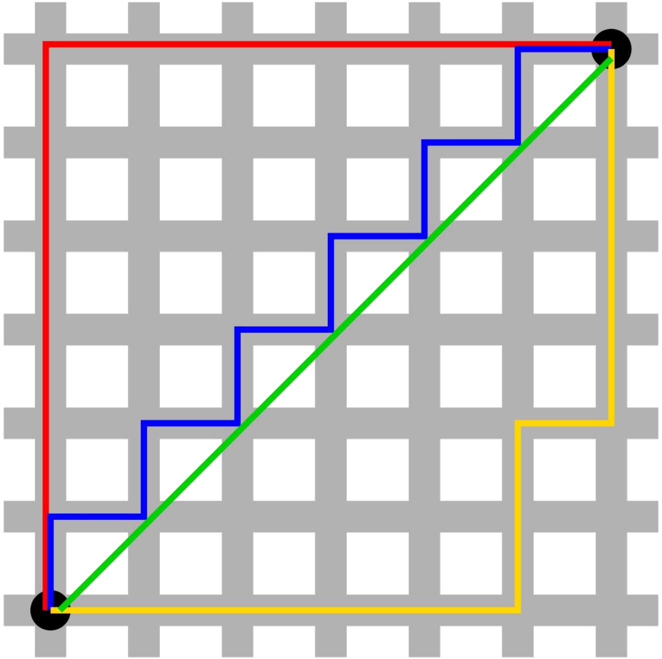
¿El camino mínimo cityblock entre dos puntos es único?

A) Verdadero

B) Falso

https://en.wikipedia.org/wiki/Taxicab_geometry

Distancia City-Block / Manhattan / o D_4



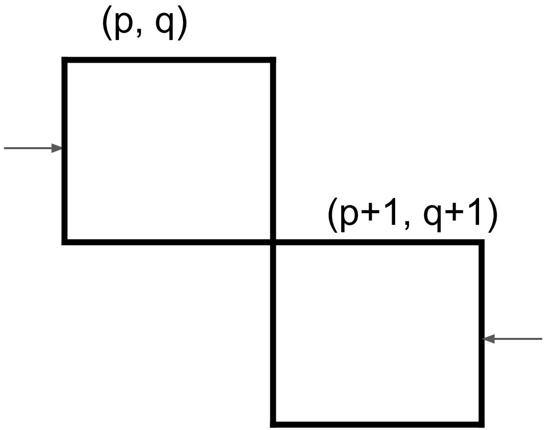
¿El camino mínimo cityblock entre dos puntos es único?

A) Verdadero

B) Falso

https://en.wikipedia.org/wiki/Taxicab_geometry

Distancia cityblock entre píxeles: Diagonal

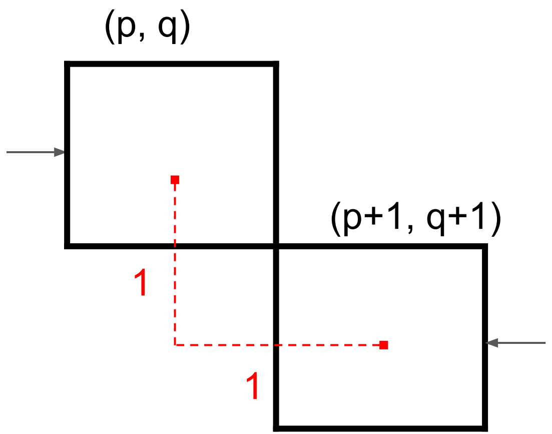


A) 1 pixel

B) $\sqrt{2}$ pixeles

C) 2 pixeles

Distancia cityblock entre píxeles: Diagonal



A) 1 pixel

B) $\sqrt{2}$ pixeles

C) 2 pixeles

Multiplicación por escalar

- **β** un escalar y **a** un vector a n-dimensionales pueden ser modificados (también denotado por **aβ**)

$$\beta a = (\beta a_1, \ldots, \beta a_n)$$

- Ejemplo

$$(-2)\begin{bmatrix} 1 \\ 9 \\ 6 \end{bmatrix} = \begin{bmatrix} -2 \\ -18 \\ -12 \end{bmatrix}$$

- en Numpy

```
beta = -2
a = np.array([1,9,6])
beta*a
```

```
array([ -2, -18, -12])
```

Propiedades de la multiplicación por escalar

- asociatividad:

$$(\beta\gamma)a = \beta(\gamma a)$$

- distributiva por la izquierda:

$$(\beta + \gamma)a = \beta a + \gamma a$$

- distributiva por la derecha:

$$\beta(a + b) = \beta a + \beta b$$

Combinaciones lineales

- para los vectores $\mathbf{a}_1, \ldots, \mathbf{a}_m$ y los escalares $\boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_m$,

$$\boldsymbol{\beta}_1 \mathbf{a}_1 + \ldots \boldsymbol{\beta}_m \mathbf{a}_m$$

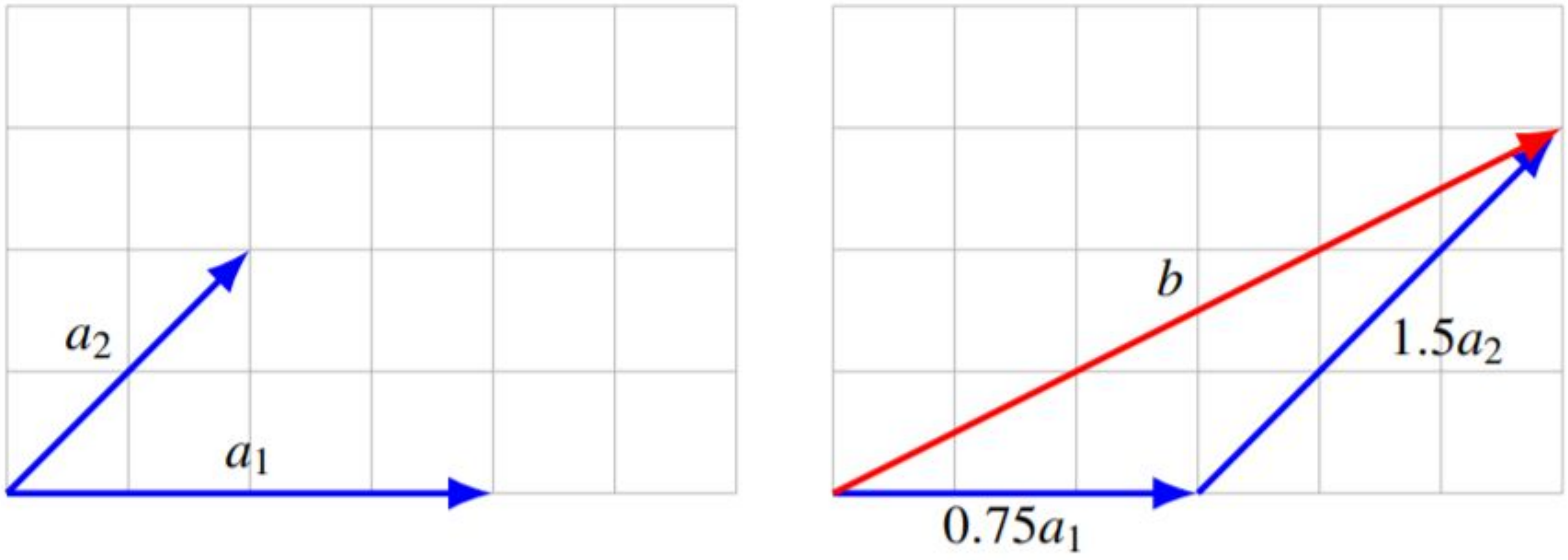
  es una combinación lineal de los vectores

- $\boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_1$, son los coeficientes
- Una identidad simple para cada vector n-dimensional $\mathbf{b}$

$$\mathbf{b} = \mathbf{b}_1 \mathbf{e}_1 + \ldots + \mathbf{b}_n \mathbf{e}_n$$

Ejemplo de combinación lineal

dos vectores $\mathbf{a}_1$ y $\mathbf{a}_2$, y la combinación lineal $\mathbf{b} = 0.75\mathbf{a}_1 + 1.5\mathbf{a}_2$



Ejemplo de combinación lineal en Numpy

```
beta1 = -2
beta2 = 4
a1 = np.array([1,9,6])
a2 = np.array([3,1,3])
beta1*a1 + beta2*a2
```

```
array([ 10, -14,   0])
```

Producto Interno

- El producto interno (o producto punto -- **dot** en inglés) de los vectores n-dimensionales a y b es

$$a^T b = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

- Otras notaciones usadas $\langle a, b \rangle, \langle a | b \rangle, (a, b), a \cdot b$
- Ejemplo

$$\begin{bmatrix} -1 \\ 2 \\ 2 \end{bmatrix}^T \begin{bmatrix} 1 \\ 0 \\ -3 \end{bmatrix} = (-1)(1) + (2)(0) + (2)(-3) = -7$$

```python
a = np.array([-1,2,2])
b = np.array([1,0,-3])
np.dot(a,b)
```

```
-7
```

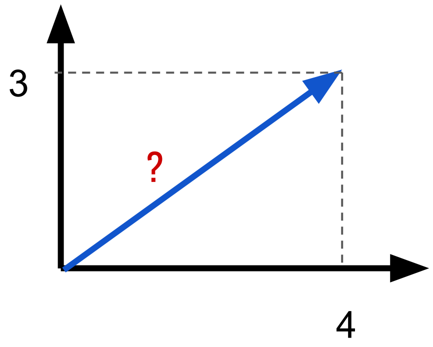
Norma

- La norma Euclidiana de un vector n-dimensional x es

$$||x|| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2} = \sqrt{x^T x}$$

- Se usa para medir el ¨tamaño¨ de un vector
- Se reduce al valor absoluto cuando **n=1**

Norma en Numpy



Opción 1

```
x = np.array([4,3])
np.sqrt(np.dot(x,x))
```

```
5.0
```

Opción 2

```
x = np.array([4,3])
np.sqrt(np.sum(x**2))
```

```
5.0
```

Opción 3

```
x = np.array([4,3])
np.linalg.norm(x)

```

```
5.0
```

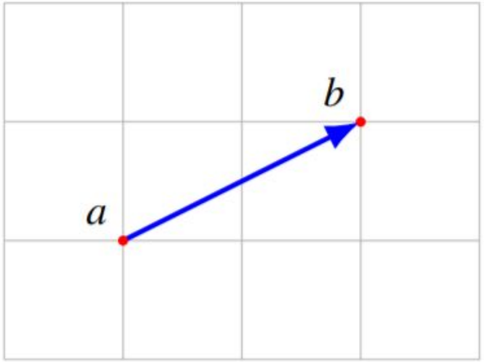
Otras Normas en numpy.linalg.norm

ord	norm for matrices	norm for vectors
None	Frobenius norm	2-norm
'fro'	Frobenius norm	–
'nuc'	nuclear norm	–
inf	max(sum(abs(x), axis=1))	max(abs(x))
-inf	min(sum(abs(x), axis=1))	min(abs(x))
0	–	sum(x != 0)
1	max(sum(abs(x), axis=0))	as below
-1	min(sum(abs(x), axis=0))	as below
2	2-norm (largest sing. value)	as below
-2	smallest singular value	as below
other	–	sum(abs(x)**ord)**(1./ord)

Distancia

- La distancia (euclidiana) entre los vectores a y b, es

$$\textbf{dist}(a,b) = \|a - b\|$$



Podemos pensar en la distancia como el "tamaño" (norma) del vector desplazamiento. Funciona para 2d, 3d, etc.

Matrices

- Una matriz es un arreglo rectangular de números

$$\begin{bmatrix} 0 & 1 & -2.3 & 0.1 \\ 1.3 & 4 & -0.1 & 0 \\ 4.1 & -1 & 0 & 1.7 \end{bmatrix}$$

```
B = np.array([[0, 1, -2.3, 0.1],
              [1.3,4,-0.1,0],
              [4.1,-1, 0, 1.7]])
print(B)
```

```
[[ 0.   1.  -2.3  0.1]
 [ 1.3  4.  -0.1  0. ]
 [ 4.1 -1.   0.   1.7]]
```

Matrices

- Su tamaño está dado por (dimensión filas) x (dimensión columnas). Por ejemplo la matriz anterior es **3 x 4.** En Numpy podemos usar el atributo **shape**

```
B = np.array([[0, 1, -2.3, 0.1],
              [1.3,4,-0.1,0],
              [4.1,-1, 0, 1.7]])
print(B)
```

```
[[ 0.   1.  -2.3  0.1]
 [ 1.3  4.  -0.1  0. ]
 [ 4.1 -1.   0.   1.7]]
```

```
B.shape
```

```
(3, 4)
```

Matrices

- Los elementos también son llamados entradas o coeficientes
- $\mathbf{B}_{\mathbf{ij}}$ es el elemento **i,j** de la matriz **B**
- **i** es el índice de la fila y **j** el índice de la columna

$$\begin{bmatrix} 0 & 1 & -2.3 & 0.1 \\ 1.3 & 4 & -0.1 & 0 \\ 4.1 & -1 & 0 & 1.7 \end{bmatrix}$$

<u>Dos formas de acceder a las entradas</u>

```
B[1,0]
```

```
1.3
```

```
B[1][0]
```

```
1.3
```

Recordar que en Numpy se indexa desde 0

Matrices

- Dos matrices son iguales, si tienen la misma dimensión y las entradas correspondientes son iguales

```
B = np.array([[0, 1, -2.3, 0.1],
              [1.3,4,-0.1,0],
              [4.1,-1, 0, 1.7]])
A = np.array([[0, 1, -2.3, 0.1],
              [1.3,4,-0.1,0],
              [4.1,-1, 0, 1.7]])
np.array_equal(A,B)
```

```
True
```

numpy.array_equal

numpy.array_equal(*a1*, *a2*)

True if two arrays have the same shape and elements, False otherwise.

Parameters: a1, a2 : *array_like*

Input arrays.

Returns: b : *bool*

Returns True if the arrays are equal.

Nota: Verificar qué pasa cuando se hace **A==B**

Matrices cuadradas

- Caso especial en el que una matriz **A** de dimensión **m x n** en el que **m=n**

```
1   np.zeros((10, 10))
```

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

Vectores fila y Vectores Columna

- Consideramos una matriz **n x 1** como un vector ***n-dimensional***
- Consideramos una matriz **1 x 1** como un número
- Una matriz **1 x n** es llamada un vector fila

$$\begin{bmatrix} 1.2 & -0.3 & 1.4 & 2.6 \end{bmatrix}$$

que no es lo mismo que un vector columna

$$\begin{bmatrix} 1.2 \\ -0.3 \\ 1.4 \\ 2.6 \end{bmatrix}$$

Columnas y Filas de una matriz

- Suponga que **A** es una matriz **m x n** con entradas $\mathbf{A_{ij}}$ para **i = 1,...,*m*,j = 1,...*n***
- La columna *j-ésima* es (el vector m-dimensional)

$$\begin{bmatrix} A_{1j} \\ \vdots \\ A_{mj} \end{bmatrix}$$

```python
B = np.array([[0, 1, -2.3, 0.1],
              [1.3,4,-0.1,0],
              [4.1,-1, 0, 1.7]])
j = 1
B[:, j]
```

```
array([ 1.,  4., -1.])
```

- La fila *i-ésima* es (el vector fila n-dimensional)

$$\begin{bmatrix} A_{i1} & \cdots & A_{in} \end{bmatrix}$$

```python
B = np.array([[0, 1, -2.3, 0.1],
              [1.3,4,-0.1,0],
              [4.1,-1, 0, 1.7]])
i = 1
B[i, :]
```

```
array([ 1.3,  4. , -0.1,  0. ])
```

Slices de una matriz

- Un ‘slice’ de la matriz: $\mathbf{A}_{\mathbf{p:q,\ r:s}}$ es la matriz con dimensiones **(q - p + 1) x (s - r + 1)**

$$A_{p:q,r:s} = \begin{bmatrix} A_{pr} & A_{p,r+1} & \cdots & A_{ps} \\ A_{p+1,r} & A_{p+1,r+1} & \cdots & A_{p+1,s} \\ \vdots & \vdots & & \vdots \\ A_{qr} & A_{q,r+1} & \cdots & A_{qs} \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & -2.3 & 0.1 \\ 1.3 & \boxed{4} & \boxed{-0.1} & 0 \\ 4.1 & \boxed{-1} & \boxed{0} & 1.7 \end{bmatrix}$$

```
p=1
q=3
r=1
s=3

B[p:q, r:s]
```

```
array([[ 4. , -0.1],
       [-1. ,  0. ]])
```

$\underbrace{\hspace{6em}}$

¿Qué puede decir sobre el *slicing* de Numpy?

Matriz 0

- La matriz $\mathbf{0}_{mxn}$ con dimensión **m x n** tiene todas sus entradas iguales a **0**

```
1   np.zeros((10, 10))
```

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

Matriz identidad

- Es una matriz cuadrada con las entradas en la diagonal iguales a 1, o $I_{ii} = 1$ e $I_{ij} = 0$ para $i \neq j$

```
1   np.eye(10)
```

```
array([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]])
```

Transpuesta

- La transpuesta de una matriz A de dimensión m x n es denotada por A^T, y definida por

$$(A^T)_{ij} = A_{ji}, \quad i = 1, \ldots, n, \quad j = 1, \ldots, m$$

- Por ejemplo

$$\begin{bmatrix} 0 & 4 \\ 7 & 0 \\ 3 & 1 \end{bmatrix}^T = \begin{bmatrix} 0 & 7 & 3 \\ 4 & 0 & 1 \end{bmatrix}$$

```
A = np.array([[0,4], [7,0], [3,1]])
print(A)
```

```
[[0 4]
 [7 0]
 [3 1]]
```

```
print(A.T)
```

```
[[0 7 3]
 [4 0 1]]
```

- La transpuesta convierte vectores-columna en vectores-fila (y viceversa)
- $(A^T)^T = A$

Adición, sustracción y multiplicación por escalar

- Similar al caso de los vectores, podemos sumar o sustraer matrices del mismo tamaño

$$(A+B)_{ij} = A_{ij} + B_{ij}, \quad i = 1,\ldots,m, \quad j = 1,\ldots,n$$

```
A = np.array([[0,4], [7,0], [3,1]])
B = np.array([[5,2], [2,9], [4,1]])
A + B
```

```
array([[5, 6],
       [9, 9],
       [7, 2]])
```

- Multiplicación por escalar

$$(\alpha A)_{ij} = \alpha A_{ij}, \quad i = 1,\ldots,m, \quad j = 1,\ldots,n$$

```
A = np.array([[0,4], [7,0], [3,1]])
alpha = 0.5
alpha*A
```

```
array([[0. , 2. ],
       [3.5, 0. ],
       [1.5, 0.5]])
```

Norma de una matriz

- Para una matriz **A** de dimensión **m x n**, definimos

$$||A|| = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} A_{ij}^2 \right)^{1/2}$$

```
1  np.linalg.norm(A)
```

```
8.660254037844387
```

```
1  np.sqrt(np.sum(A**2))
```

```
8.660254037844387
```

- Concuerda con la norma de los vectores cuando **n = 1**
- Distancia entre 2 matrices: **|| A - B ||**
- Nota: Existen otras normas

Diferencia de np.array y np.matrix

- **np.array** es un objeto que admite arreglos n-dimensionales, en particular, cuando **n=2** estamos trabajando con matrices
- **np.matrix** hereda de **np.array**, pero admite únicamente **n=2**
- Se desalienta su uso

numpy.matrix

class **numpy.matrix**(*data*, *dtype=None*, *copy=True*) [source]

> **Note:**
> It is no longer recommended to use this class, even for linear algebra. Instead use regular arrays. The class may be removed in the future.

Returns a matrix from an array-like object, or from a string of data. A matrix is a specialized 2-D array that retains its 2-D nature through operations. It has certain special operators, such as `*` (matrix multiplication) and `**` (matrix power).

Parameters: **data** : *array_like or string*

If data is a string, it is interpreted as a matrix with commas or spaces separating columns, and semicolons separating rows.

Listas vs nd-array

- Instanciemos una lista y un arreglo de Numpy con 1000 elementos y comparemos los tiempos para
    - Elevar todos los elementos de la **lista** al cuadrado (*list comprehension*)
    - Elevar todos los elementos del **ndarray** al cuadrado (*list comprehension*)
    - Elevar todos los elementos del **ndarray** al cuadrado (*broadcasting*)

Especificando el tipo de dato

- Se puede especificar el tipo de dato con **dtype**

```
1   np.array([1,2,3,4,5], dtype=np.uint8)
```

```
array([1, 2, 3, 4, 5], dtype=uint8)
```

Algunos atributos

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

```
a.dtype
```

dtype('int64')

tipo de dato

```
a.itemsize
```

8

tamaño del elemento

```
a.nbytes
```

40

espacio ocupado por el arreglo

Slicing

- No crea un nuevo arreglo, sino que retorna una vista sobre el arreglo
- Si modifico algo en el arreglo original, también se modificará en la **vista** *(view)*

```
a = np.random.randn(3, 3)
a
```

```
array([[ 0.54968647,  1.71795939, -1.46465233],
       [-0.59643479,  0.78715214, -2.30566108],
       [-0.81750065,  1.11237911, -0.35839786]])
```

```
b = a[:2,:2]
b
```

```
array([[ 0.54968647,  1.71795939],
       [-0.59643479,  0.78715214]])
```

```
a[0,0]=99
```

```
a
```

```
array([[99.        ,  1.71795939, -1.46465233],
       [-0.59643479,  0.78715214, -2.30566108],
       [-0.81750065,  1.11237911, -0.35839786]])
```

```
b
```

```
array([[99.        ,  1.71795939],
       [-0.59643479,  0.78715214]])
```

Indexing

- Podemos usar el resultado de operaciones sobre los elementos de un ndarray para indexarlo
- Por ejemplo, si queremos filtrar los elementos que son mayores que 5

```
1   a = np.random.randint(-10,10 , size=(5, 5))
```

```
1   a
```

```
array([[-10,   8,  -7,   6,  -4],
       [-10,  -8, -10,  -1,   4],
       [  4,   0,  -5,   5,   5],
       [  5,   9,  -5,  -4,   9],
       [ -6,  -1,   2,   4,   0]])
```

```
1   a > 5
```

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

Indexing

- Se puede acceder directamente a los elementos que cumplen una condición por ejemplo

```
1   a = np.random.randint(-10,10 , size=(5, 5))
```

```
1   a
```

```
array([[-10,   8,  -7,   6,  -4],
       [-10,  -8, -10,  -1,   4],
       [  4,   0,  -5,   5,   5],
       [  5,   9,  -5,  -4,   9],
       [ -6,  -1,   2,   4,   0]])
```

```
1   a[a > 5]
```

```
array([8, 6, 9, 9])
```

```
1   np.sum(a[a>5])
```

```
32
```

Indexing

- También se puede utilizar la vista creada para alterar los valores

```
a[a<5] = 0
```

```
a
```

```
array([[0, 8, 0, 6, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 5, 5],
       [5, 9, 0, 0, 9],
       [0, 0, 0, 0, 0]])
```

Axis

- La clase **ndarray** soporta arreglos n-dimensionales, donde **n** es la cantidad (arbitraria) de dimensiones
- El atributo **ndim** nos indica la cantidad de dimensiones

Axis

```
A = np.random.randn(8,2)
print(A)
```

```
[[-2.12756946  1.03420631]
 [-0.63992641 -1.07280277]
 [-0.84002219  0.4977678 ]
 [-0.11522883  1.27649322]
 [ 1.44434719 -0.75183633]
 [ 0.07281258 -0.31710428]
 [-0.87762739 -0.4390274 ]
 [ 1.3043964   1.066951  ]]
```

```
A.sum()
```

```
-0.4841705502097673
```

```
A.sum(axis=0)
```

```
array([-1.77881811,  1.29464756])
```

```
A.sum(axis=1)
```

```
array([-1.09336315, -1.71272918, -0.34225439,  1.1612644 ,  0.69251086,
       -0.24429171, -1.31665479,  2.37134741])
```

Axis

```python
A = np.random.randn(8,2)
print(A)
```

```
[[-2.12756946  1.03420631]
 [-0.63992641 -1.07280277]
 [-0.84002219  0.4977678 ]
 [-0.11522883  1.27649322]
 [ 1.44434719 -0.75183633]
 [ 0.07281258 -0.31710428]
 [-0.87762739 -0.4390274 ]
 [ 1.3043964   1.066951  ]]
```

```python
A.sum()
```

```
-0.4841705502097673
```

```python
A.sum(axis=0)
```

```
array([-1.77881811,  1.29464756])
```

```python
A.sum(axis=1)
```

```
array([-1.09336315, -1.71272918, -0.34225439,  1.1612644 ,  0.69251086,
       -0.24429171, -1.31665479,  2.37134741])
```

Axis

```
A = np.random.randn(8,2)
print(A)
```

```
[[-2.12756946  1.03420631]
 [-0.63992641 -1.07280277]
 [-0.84002219  0.4977678 ]
 [-0.11522883  1.27649322]
 [ 1.44434719 -0.75183633]
 [ 0.07281258 -0.31710428]
 [-0.87762739 -0.4390274 ]
 [ 1.3043964   1.066951  ]]
```

```
A.sum()
```

```
-0.4841705502097673
```

```
A.sum(axis=0)
```

```
array([-1.77881811,  1.29464756])
```

```
A.sum(axis=1)
```

```
array([-1.09336315, -1.71272918, -0.34225439,  1.1612644 ,  0.69251086,
       -0.24429171, -1.31665479,  2.37134741])
```

Reshape

- Retorna una vista si es posible, o una copia
- -1 significa que esa dimensión se definirá implícitamente

```python
A = np.random.randn(8,2)
print(A)
```

```
[[-2.12756946  1.03420631]
 [-0.63992641 -1.07280277]
 [-0.84002219  0.4977678 ]
 [-0.11522883  1.27649322]
 [ 1.44434719 -0.75183633]
 [ 0.07281258 -0.31710428]
 [-0.87762739 -0.4390274 ]
 [ 1.3043964   1.066951  ]]
```

```python
A.reshape(4,-1)
```

```
array([[-2.12756946,  1.03420631, -0.63992641, -1.07280277],
       [-0.84002219,  0.4977678 , -0.11522883,  1.27649322],
       [ 1.44434719, -0.75183633,  0.07281258, -0.31710428],
       [-0.87762739, -0.4390274 ,  1.3043964 ,  1.066951  ]])
```

Broadcasting

- El término describe cómo actúa Numpy con arreglos de diferentes tamaños
- Muchas de las operaciones de Numpy se aplican elemento a elemento

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = np.array([2.0, 2.0, 2.0])
>>> a * b
array([ 2.,  4.,  6.])
```

Broadcasting: el caso básico

- El caso más básico es la multiplicación por escalar

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = 2.0
>>> a * b
array([ 2.,  4.,  6.])
```