CONQUER BLOCKS

PYTHON

TRABAJANDO CON ARRAYS (PARTE 1)

CLASE ANTERIOR

1) Qué es un módulo

2) Qué es un package/paquete

3) Qué es una library/librería

4) Qué es un Array y las diferencias con una Lista

5) Cómo importar los módulos y librerías relacionadas con Arrays y su sintaxis

INSTALAR NUMPY

Instalación…

Activamos nuestro environment de trabajo:

```
(base) MacBook-Pro-4:Prueba Elena$ conda activate cblocks
(cblocks) MacBook-Pro-4:Prueba Elena$
```

Dentro del environment de trabajo instalamos numpy:

Podemos usar conda…

```
(cblocks) MacBook-Pro-4:Prueba Elena$ conda install numpy
```

O también pip…

```
(cblocks) MacBook-Pro-4:Prueba Elena$ pip install numpy
```

```
import numpy as np
my_array = np.array([1,2,3,4,5,6,7,8,9])
print(type(my_array))
```

✓ 0.3s

```
<class 'numpy.ndarray'>
```

DE LISTA A ARRAY

```
import numpy as np
# de lista a array
lista_a_array = np.array([1,2,3])
print(lista_a_array)
```

✓ 0.1s

```
[1 2 3]
```

```
import numpy as np
# de lista a array
my_list = [1,2,3]
lista_a_array = np.array(my_list)
print(lista_a_array)
```

✓ 0.0s

```
[1 2 3]
```

Parecen lo mismo...
Tienen prácticamente el mismo propósito…

¿Por qué convertir una lista en un array?

Eficiencia en memoria

DE LISTA A ARRAY

En una lista podemos guardar todo tipo de datos…

Boolean

String

DE LISTA A ARRAY

En una lista podemos guardar todo tipo de datos…

Boolean

True / False

1 / 0

String

DE LISTA A ARRAY

En una lista podemos guardar todo tipo de datos…

Boolean

True / False

1 / 0

2 opciones

String

DE LISTA A ARRAY

En una lista podemos guardar todo tipo de datos…

Boolean

True / False

1 / 0

2 opciones

String

26 letras minúsculas

26 letras mayúsculas

10 digitos

10+ signos de puntuación

DE LISTA A ARRAY

En una lista podemos guardar todo tipo de datos…

Boolean

True / False

1 / 0

2 opciones

String

26 letras minúsculas

26 letras mayúsculas

10 digitos

10+ signos de puntuación

70+ opciones

¡para un caracter!

DE LISTA A ARRAY

En una lista podemos guardar todo tipo de datos…

Boolean

True / False

2 opciones

String

26 letras minúsculas

26 letras mayúsculas

10+ signos de puntuación

70+ opciones

¡para un caracter!

Esta flexibilidad de las listas es al mismo tiempo una locura en términos de manejo de memoria

DE LISTA A ARRAY

Los Arrays están pensados para guardar un único tipo de dato

Y no solo en términos de si se trata de enteros o decimales, si no en términos del ***número de bits***

DE LISTA A ARRAY

Los Arrays están pensados para guardar un único tipo de dato

Y no solo en términos de si se trata de enteros o decimales, si no en términos del ***número de bits***

```
import numpy as np
# de lista a array
lista_a_array = np.array([1,2,3])
print(type(lista_a_array[0]))
```

✓ 0.0s

```
<class 'numpy.int64'>
```

Entero de 64 bits

DE LISTA A ARRAY

Los Arrays están pensados para guardar un único tipo de dato

Y no solo en términos de si se trata de enteros o decimales, si no en términos del *número de bits*

```
import numpy as np
# de lista a array
lista_a_array = np.array([1,2,3])
print(type(lista_a_array[0]))
```

✓ 0.0s

```
<class 'numpy.int64'>
```

Entero de 64 bits

¿Necesitamos 64 bits para representar los números 1, 2 y 3?

DE LISTA A ARRAY

Los Arrays están pensados para guardar un único tipo de dato

Y no solo en términos de si se trata de enteros o decimales, si no en términos del *número de bits*

```
import numpy as np
# de lista a array
lista_a_array = np.array([1,2,3])
print(type(lista_a_array[0]))
```

✓ 0.0s

```
<class 'numpy.int64'>
```

Entero de 64 bits

¿Necesitamos 64 bits para representar los números 1, 2 y 3?

decimal

binario

DE LISTA A ARRAY

Los Arrays están pensados para guardar un único tipo de dato

Y no solo en términos de si se trata de enteros o decimales, si no en términos del ***número de bits***

```
import numpy as np
# de lista a array
lista_a_array = np.array([1,2,3])
print(type(lista_a_array[0]))
```

✓ 0.0s

```
<class 'numpy.int64'>
```

Entero de 64 bits

¿Necesitamos 64 bits para representar los números 1, 2 y 3?

decimal	binario
1	0 1

DE LISTA A ARRAY

Los Arrays están pensados para guardar un único tipo de dato

Y no solo en términos de si se trata de enteros o decimales, si no en términos del *número de bits*

```
import numpy as np
# de lista a array
lista_a_array = np.array([1,2,3])
print(type(lista_a_array[0]))
```

✓ 0.0s

```
<class 'numpy.int64'>
```

Entero de 64 bits

¿Necesitamos 64 bits para representar los números 1, 2 y 3?

decimal	binario
1	0 1
2	1 0

DE LISTA A ARRAY

Los Arrays están pensados para guardar un único tipo de dato

Y no solo en términos de si se trata de enteros o decimales, si no en términos del ***número de bits***

```
import numpy as np
# de lista a array
lista_a_array = np.array([1,2,3])
print(type(lista_a_array[0]))
```

✓ 0.0s

```
<class 'numpy.int64'>
```

Entero de 64 bits

¿Necesitamos 64 bits para representar los números 1, 2 y 3?

decimal	binario
1	0 1
2	1 0
3	1 1

En este caso necesitamos un máximo de 2 bits

DE LISTA A ARRAY

Los Arrays están pensados para guardar un único tipo de dato

Y no solo en términos de si se trata de enteros o decimales, si no en términos del ***número de bits***

```
import numpy as np
# de lista a array
lista_a_array = np.array([1,2,3])
print(type(lista_a_array[0]))
```

✓ 0.0s

```
<class 'numpy.int64'>
```

Entero de 64 bits

```
import numpy as np
# de lista a array
lista_a_array = np.array([1,2,3], dtype = np.int8)
print(type(lista_a_array[0]))
```

✓ 0.0s

```
<class 'numpy.int8'>
```

Entero de 8 bits ⟶ Consumimos menos memoria

ARRAYS MULTIDIMENSIONALES

Hasta ahora hemos tratado con arrays unidimensionales ⟶ Todos los datos están en una linea

¿Y si queremos lineas y columnas?

ARRAY BIDIMENSIONAL

```
# Array bidimensional 2D
lista_a_array = np.array([[1,2,3],[4,5,6]], dtype = np.int8)
print(lista_a_array)
```

✓ 0.0s

```
[[1 2 3]
 [4 5 6]]
```

2 columnas

3 filas

ARRAYS MULTIDIMENSIONALES

Hasta ahora hemos tratado con arrays unidimensionales → Todos los datos están en una linea

¿Y si queremos lineas y columnas?

ARRAY BIDIMENSIONAL

```
# Array bidimensional 2D
lista_a_array = np.array([[1,2,3],[4,5,6]], dtype = np.int8)
print(lista_a_array.shape)
```

✓ 0.0s

```
(2, 3)
```

2 filas 3 columnas

ARRAYS MULTIDIMENSIONALES

Hasta ahora hemos tratado con arrays unidimensionales → Todos los datos están en una linea

¿Y si queremos lineas y columnas?

ARRAY TRIDIMENSIONAL

```
# Array tridimensional 3D
lista_a_array = np.array([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]], dtype = np.int8)
print(lista_a_array)
print(lista_a_array.shape)
```

✓ 0.0s

```
[[[ 1  2  3]
  [ 4  5  6]]

 [[ 7  8  9]
  [10 11 12]]]
(2, 2, 3)
```

ARRAYS MULTIDIMENSIONALES

Hasta ahora hemos tratado con arrays unidimensionales → Todos los datos están en una linea

¿Y si queremos lineas y columnas?

ARRAY TRIDIMENSIONAL

```
# Array tridimensional 3D
lista_a_array = np.array([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]], dtype = np.int8)
print(lista_a_array)
print(lista_a_array.ndim)
```

✓ 0.0s

```
[[[ 1  2  3]
  [ 4  5  6]]

 [[ 7  8  9]
  [10 11 12]]]
3
```

ARRAYS MULTIDIMENSIONALES

Podemos redimensionar el array con reshape():

```
# Array bidimensional shape 2,3
array_1 = np.array([[1,2,3],[4,5,6]], dtype = np.int8)
print("array_1 shape:", array_1.shape)
print(array_1)
array_2 = array_1.reshape((3,2))
print("array_2 shape:", array_2.shape)
print(array_2)
```

✓ 0.0s

```
array_1 shape: (2, 3)
[[1 2 3]
 [4 5 6]]
array_2 shape: (3, 2)
[[1 2]
 [3 4]
 [5 6]]
```

ARRAYS MULTIDIMENSIONALES

Podemos redimensionar el array con reshape():

```
# Array bidimensional shape 2,3
array_1 = np.array([[1,2,3],[4,5,6]], dtype = np.int8)
print("array_1 shape:", array_1.shape)
print(array_1)
array_2 = array_1.reshape((6,1))
print("array_2 shape:", array_2.shape)
print(array_2)
```

✓ 0.0s

```
array_1 shape: (2, 3)
[[1 2 3]
 [4 5 6]]
array_2 shape: (6, 1)
[[1]
 [2]
 [3]
 [4]
 [5]
 [6]]
```

ARRAYS MULTIDIMENSIONALES

Podemos redimensionar el array con reshape():

```
# Array bidimensional shape 2,3
array_1 = np.array([[1,2,3],[4,5,6]], dtype = np.int8)
print("array_1 shape:", array_1.shape, "dim", array_1.ndim)
print(array_1)
array_2 = array_1.reshape(6)
print("array_2 shape:", array_2.shape, "dim", array_2.ndim)
print(array_2)
```

✓ 0.0s

```
array_1 shape: (2, 3) dim 2
[[1 2 3]
 [4 5 6]]
array_2 shape: (6,) dim 1
[1 2 3 4 5 6]
```

Hemos convertido un array de dimension 2 en un array de dimension 1

CREACIÓN DE ARRAYS SIN USAR LISTAS

Podemos crear arrays con numpy.arange():

```
import numpy as np
my_array = np.arange(8)
print(my_array)
print(type(my_array))
```

✓ 0.0s

```
[0 1 2 3 4 5 6 7]
<class 'numpy.ndarray'>
```

```
import numpy as np
my_array = np.array([0,1,2,3,4,5,6,7])
print(my_array)
print(type(my_array))
```

✓ 0.0s

```
[0 1 2 3 4 5 6 7]
<class 'numpy.ndarray'>
```

CREACIÓN DE ARRAYS SIN USAR LISTAS

Podemos crear arrays con numpy.arange():

Stop Value

```
import numpy as np
my_array = np.arange(8)
print(my_array)
print(type(my_array))
```

✓ 0.0s

```
[0 1 2 3 4 5 6 7]
<class 'numpy.ndarray'>
```

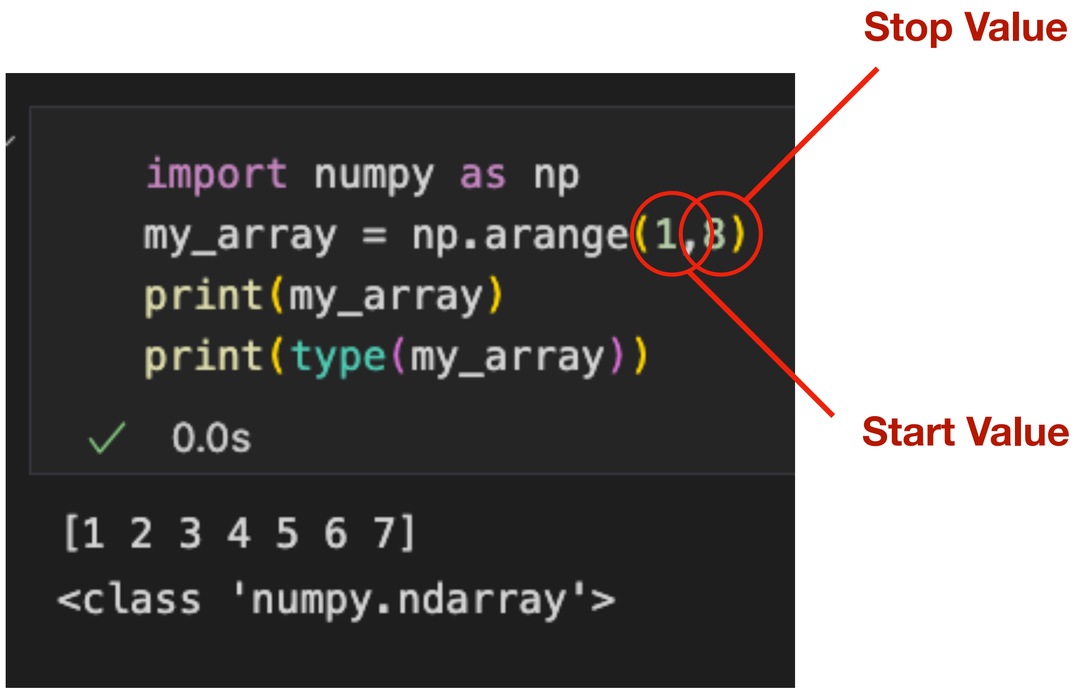
```
import numpy as np
my_array = np.array([0,1,2,3,4,5,6,7])
print(my_array)
print(type(my_array))
```

✓ 0.0s

```
[0 1 2 3 4 5 6 7]
<class 'numpy.ndarray'>
```

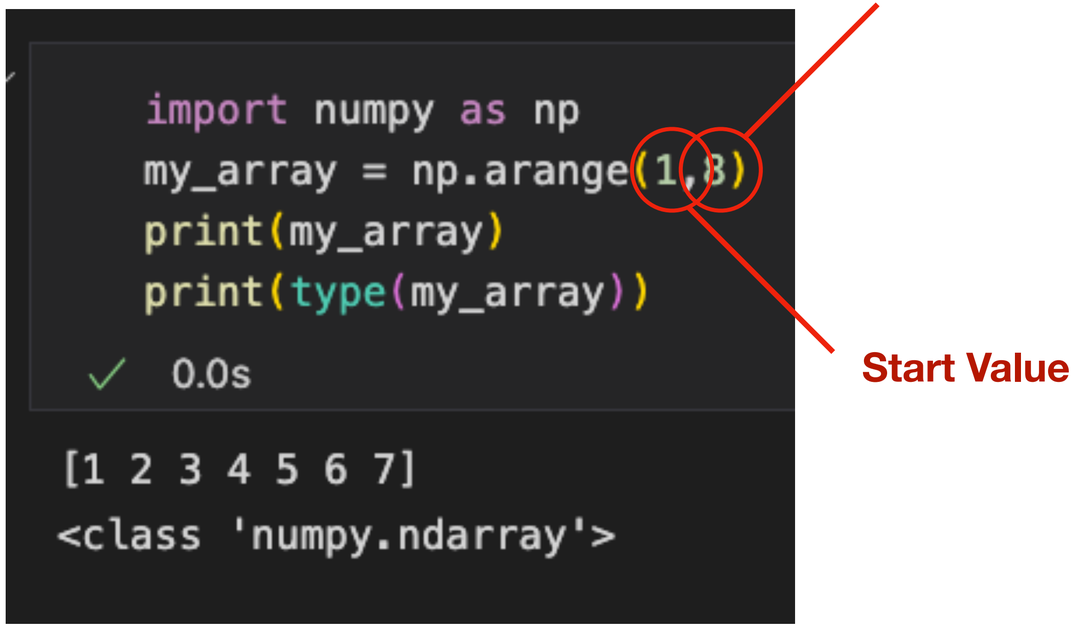
CREACIÓN DE ARRAYS SIN USAR LISTAS

Podemos crear arrays con numpy.arange():



CREACIÓN DE ARRAYS SIN USAR LISTAS

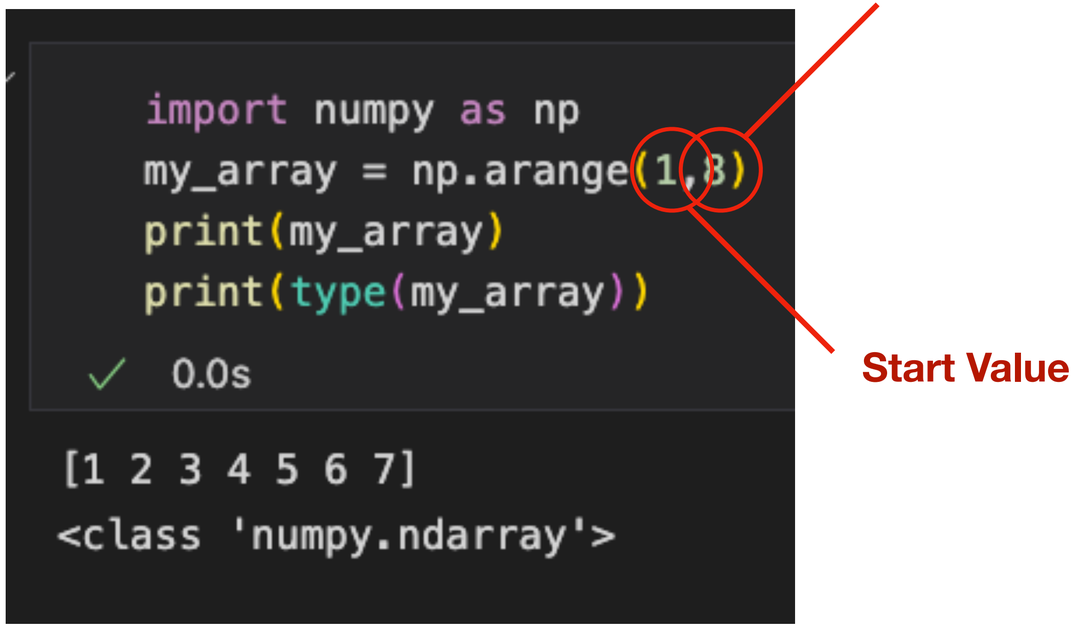
Podemos crear arrays con numpy.arange():



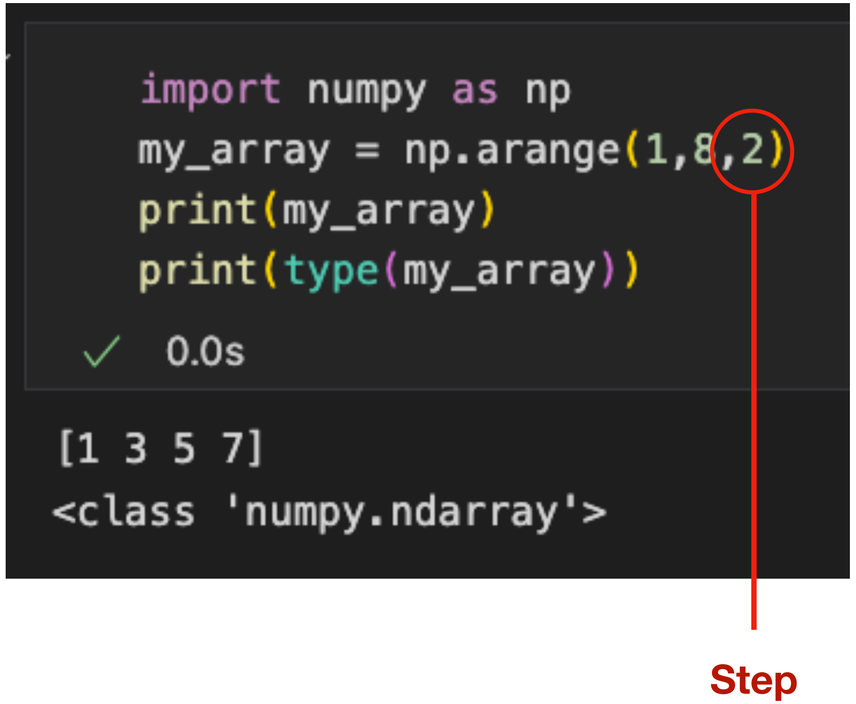
¿Y si solo quisiese números impares?

CREACIÓN DE ARRAYS SIN USAR LISTAS

Podemos crear arrays con numpy.arange():



¿Y si solo quisiese números impares?



CREACIÓN DE ARRAYS SIN USAR LISTAS

También podemos trabajar con decimales:

```
import numpy as np
my_array = np.arange(1,8,0.5)
print(my_array)
print(type(my_array))
```

✓ 0.0s

```
[1.  1.5 2.  2.5 3.  3.5 4.  4.5 5.  5.5 6.  6.5 7.  7.5]
<class 'numpy.ndarray'>
```

CREACIÓN DE ARRAYS SIN USAR LISTAS

Y con números negativos:

```
import numpy as np
my_array = np.arange(-1,8,0.5)
print(my_array)
print(type(my_array))
```

✓ 0.0s

```
[-1.  -0.5  0.   0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5  5.   5.5
  6.   6.5  7.   7.5]
<class 'numpy.ndarray'>
```

CREACIÓN DE ARRAYS “VACÍOS”

A veces no sabemos de antemano que valores va a contener el array.
En estos casos puede interesarnos crear un array “vacío”

```
# array vacio
array_vacio = np.zeros((2,3))
print(array_vacio)
```

✓ 0.0s

```
[[0. 0. 0.]
 [0. 0. 0.]]
```

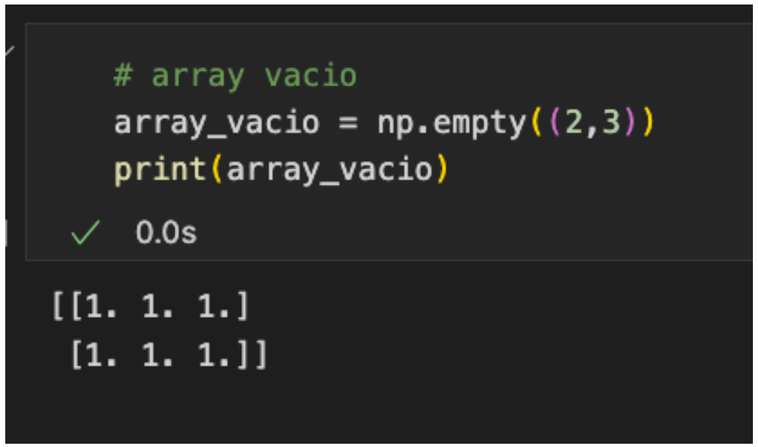
```
# array vacio
array_vacio = np.ones((2,3))
print(array_vacio)
```

✓ 0.0s

```
[[1. 1. 1.]
 [1. 1. 1.]]
```

CREACIÓN DE ARRAYS “VACÍOS”

A veces no sabemos de antemano que valores va a contener el array.
En estos casos puede interesarnos crear un array “vacío”



CREACIÓN DE ARRAYS “VACÍOS”

A veces no sabemos de antemano que valores va a contener el array.
En estos casos puede interesarnos crear un array “vacío”

```
# array vacio
array_vacio = np.empty((2,3))
print(array_vacio)
```

✓ 0.0s

```
[[1. 1. 1.]
 [1. 1. 1.]]
```

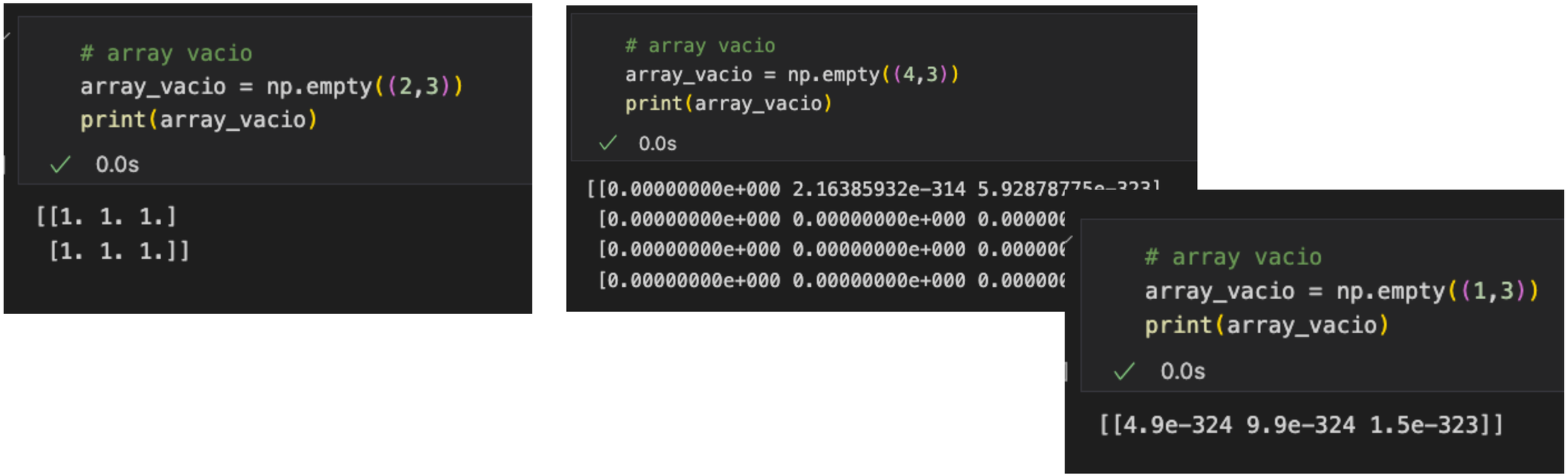
```
# array vacio
array_vacio = np.empty((4,3))
print(array_vacio)
```

✓ 0.0s

```
[[0.00000000e+000 2.16385932e-314 5.92878775e-323]
 [0.00000000e+000 0.00000000e+000 0.00000000e+000]
 [0.00000000e+000 0.00000000e+000 0.00000000e+000]
 [0.00000000e+000 0.00000000e+000 0.00000000e+000]]
```

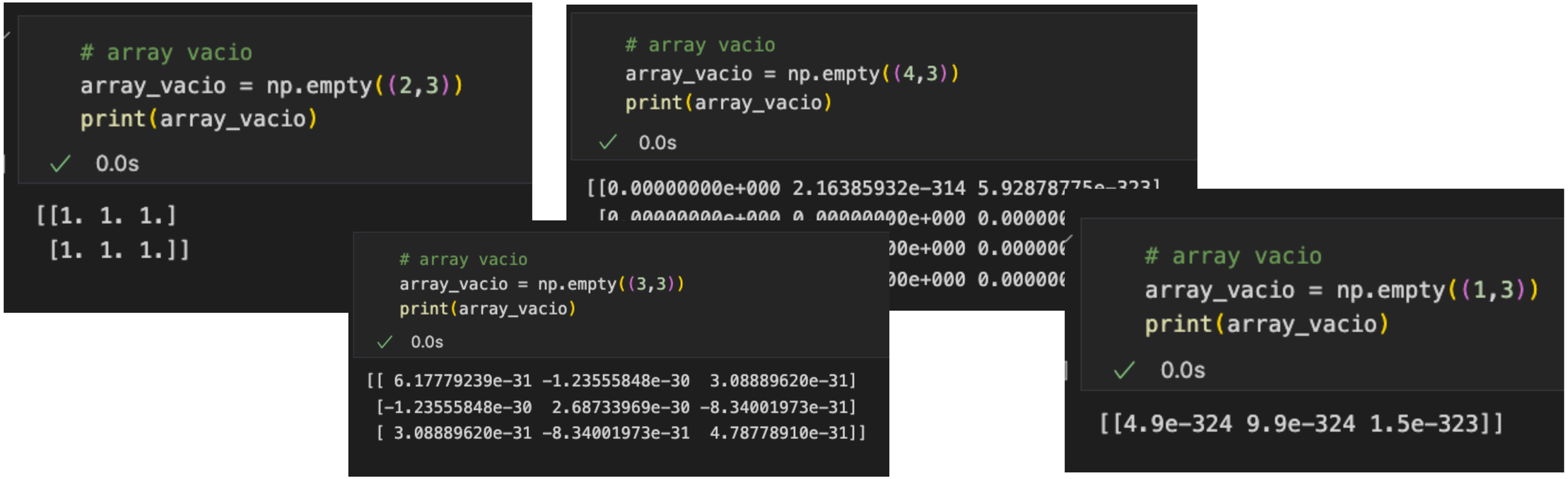
CREACIÓN DE ARRAYS “VACÍOS”

A veces no sabemos de antemano que valores va a contener el array.
En estos casos puede interesarnos crear un array “vacío”



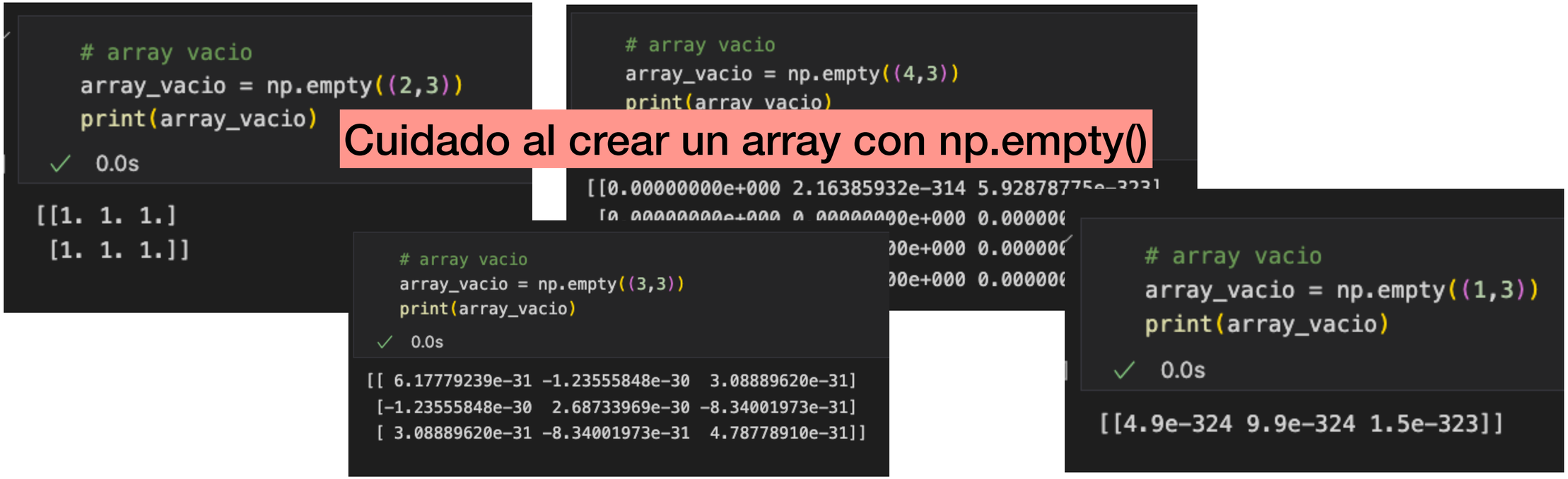
CREACIÓN DE ARRAYS “VACÍOS”

A veces no sabemos de antemano que valores va a contener el array.
En estos casos puede interesarnos crear un array “vacío”



CREACIÓN DE ARRAYS “VACÍOS”

A veces no sabemos de antemano que valores va a contener el array.
En estos casos puede interesarnos crear un array “vacío”



CREACIÓN DE ARRAYS “UNIDAD”

Podemos usar numpy.eye() para crear arrays de ceros y unos:

```
eye_array = np.eye(3)
print(eye_array)
```

✓ 0.0s

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

```
eye_array = np.eye(3, k=-1)
print(eye_array)
```

✓ 0.0s

```
[[0. 0. 0.]
 [1. 0. 0.]
 [0. 1. 0.]]
```

```
eye_array = np.eye(3, k=1)
print(eye_array)
```

✓ 0.0s

```
[[0. 1. 0.]
 [0. 0. 1.]
 [0. 0. 0.]]
```

MANIPULACION DE ARRAYS

Una vez creado el array podemos reasignar sus valores:

(Podemos hacerlo con todos los arrays, los creemos con np.array(), np.arange() o np.eye())

```
eye_array = np.eye(3, k=1)
print(eye_array)
```

✓ 0.0s

```
[[0. 1. 0.]
 [0. 0. 1.]
 [0. 0. 0.]]
```

```
eye_array = np.eye(3, k=1)
eye_array[eye_array == 0] = 2
print(eye_array)
```

✓ 0.0s

```
[[2. 1. 2.]
 [2. 2. 1.]
 [2. 2. 2.]]
```

```
eye_array = np.eye(3, k=1)
eye_array[eye_array == 0] = 2
eye_array[eye_array < 2] = 9
print(eye_array)
```

✓ 0.0s

```
[[2. 9. 2.]
 [2. 2. 9.]
 [2. 2. 2.]]
```

MANIPULACION DE ARRAYS

Una vez creado el array podemos reasignar sus valores:

(Podemos hacerlo con todos los arrays, los creemos con np.array(), np.arange() o np.eye())

```
eye_array = np.eye(3, k=1)
eye_array[0] = 2
print(eye_array)
```

✓ 0.0s

```
[[2. 2. 2.]
 [0. 0. 1.]
 [0. 0. 0.]]
```

Sustituye la fila 0

```
eye_array = np.eye(3, k=1)
eye_array[:2] = 2
print(eye_array)
```

✓ 0.0s

```
[[2. 2. 2.]
 [2. 2. 2.]
 [0. 0. 0.]]
```

Sustituye todas las filas hasta la 2

```
eye_array = np.eye(3, k=1)
eye_array[1:] = 2
print(eye_array)
```

✓ 0.0s

```
[[0. 1. 0.]
 [2. 2. 2.]
 [2. 2. 2.]]
```

Sustituye desde la fila 1 hasta la última

MANIPULACION DE ARRAYS

Una vez creado el array podemos reasignar sus valores:

(Podemos hacerlo con todos los arrays, los creemos con np.array(), np.arange() o np.eye())

```
eye_array = np.eye(3, k=1)
eye_array[1:, 2:] = 2
print(eye_array)
```

✓ 0.0s

```
[[0. 1. 0.]
 [0. 0. 2.]
 [0. 0. 2.]]
```

Sustituye desde la **fila 1** hasta la última y desde la **columna 2** hasta la última

ODERNAR EL CONTENIDO DE LOS ARRAYS

Podemos ordenar el contenido usando numpy.sort():

```
    eye_array = np.eye(3, k=1)
    eye_array[eye_array == 0] = 2
    eye_array[eye_array < 2] = 9
    eye_array[1:, :2] = 4
    print(eye_array)
```

✓ 0.0s

```
[[2. 9. 2.]
 [4. 4. 9.]
 [4. 4. 2.]]
```

```
    eye_array = np.eye(3, k=1)
    eye_array[eye_array == 0] = 2
    eye_array[eye_array < 2] = 9
    eye_array[1:, :2] = 4
    sorted_array = np.sort(eye_array)
    print(sorted_array)
```

✓ 0.0s

```
[[2. 2. 9.]
 [4. 4. 9.]
 [2. 4. 4.]]
```

```
    eye_array = np.eye(3, k=1)
    eye_array[eye_array == 0] = 2
    eye_array[eye_array < 2] = 9
    eye_array[1:, :2] = 4
    sorted_array = np.sort(eye_array, axis=0)
    print(sorted_array)
```

✓ 0.0s

```
[[2. 4. 2.]
 [4. 4. 2.]
 [4. 9. 9.]]
```

ODERNAR EL CONTENIDO DE LOS ARRAYS

Podemos ordenar el contenido usando numpy.sort():

```
eye_array = np.eye(3, k=1)
eye_array[eye_array == 0] = 2
eye_array[eye_array < 2] = 9
eye_array[1:, :2] = 4
print(eye_array)
```

✓ 0.0s

```
[[2. 9. 2.]
 [4. 4. 9.]
 [4. 4. 2.]]
```

```
eye_array = np.eye(3, k=1)
eye_array[eye_array == 0] = 2
eye_array[eye_array < 2] = 9
eye_array[1:, :2] = 4
sorted_array = np.sort(eye_array)
print(sorted_array)
```

✓ 0.0s

```
[[2. 2. 9.]
 [4. 4. 9.]
 [2. 4. 4.]]
```

```
eye_array = np.eye(3, k=1)
eye_array[eye_array == 0] = 2
eye_array[eye_array < 2] = 9
eye_array[1:, :2] = 4
sorted_array = np.sort(eye_array, axis = -1)
print(sorted_array)
```

✓ 0.0s

```
[[2. 2. 9.]
 [4. 4. 9.]
 [2. 4. 4.]]
```

ODERNAR EL CONTENIDO DE LOS ARRAYS

También podemos indicar que algoritmo queremos usar para ordenar el array:

Distintos tipos de algoritmos de ordenamiento pueden ser mas o menos rápidos dependiendo del array y los datos con los que estemos tratando

(Al nivel al que estamos trabajando ahora eso aún no lo vamos a notar)

ODERNAR EL CONTENIDO DE LOS ARRAYS

También podemos indicar que algoritmo queremos usar para ordenar el array:

```
sorted_array = np.sort(eye_array, axis = 0, kind = 'quicksort')
print(sorted_array)
```

Por defecto

```
✓ 0.0s

[[2. 4. 2.]
```

```
sorted_array = np.sort(eye_array, axis = 0, kind = 'heapsort')
print(sorted_array)
```

```
✓ 0.0s

[[2. 4. 2.]
 [4. 4. 2.]
 [4. 9. 9.]]
```

Distintos tipos de algoritmos de ordenamiento pueden ser mas o menos rápidos dependiendo del array y los datos con los que estemos tratando

(Al nivel al que estamos trabajando ahora eso aún no lo vamos a notar)

ODERNAR EL CONTENIDO DE LOS ARRAYS

También podemos indicar que algoritmo queremos usar para ordenar el array:

```
sorted_array = np.sort(eye_array, axis = 0, kind = 'quicksort')
print(sorted_array)
```

✓ 0.0s

Por defecto

```
[[2. 4. 2.]
```

```
sorted_array = np.sort(eye_array, axis = 0, kind = 'heapsort')
print(sorted_array)
```

✓ 0.0s

```
[[2. 4. 2.]
```

```
sorted_array = np.sort(eye_array, axis = 0, kind = 'mergesort')
print(sorted_array)
```

✓ 0.0s

```
[[2. 4. 2.]
 [4. 4. 2.]
 [4. 9. 9.]]
```

Distintos tipos de algoritmos de ordenamiento pueden ser mas o menos rápidos dependiendo del array y los datos con los que estemos tratando

(Al nivel al que estamos trabajando ahora eso aún no lo vamos a notar)

COPIAR ARRAYS

Hay dos opciones, view() y copy():

view():

```
array_view = sorted_array.view()
array_view[:] = 5
print(array_view)
print(sorted_array)
```

✓ 0.0s

```
[[5. 5. 5.]
 [5. 5. 5.]
 [5. 5. 5.]]
[[5. 5. 5.]
 [5. 5. 5.]
 [5. 5. 5.]]
```

copy():

```
array_copy = sorted_array.copy()
array_copy[:] = 5
print(array_copy)
print(sorted_array)
```

✓ 0.0s

```
[[5. 5. 5.]
 [5. 5. 5.]
 [5. 5. 5.]]
[[2. 4. 2.]
 [4. 4. 2.]
 [4. 9. 9.]]
```

COPIAR ARRAYS

Hay dos opciones, view() y copy():

view():

```
array_view = sorted_array.view()
array_view[:] = 5
print(array_view)
print(sorted_array)
```

view() afecta también al array original

```
[[5. 5. 5.]
 [5. 5. 5.]
 [5. 5. 5.]]
[[5. 5. 5.]
 [5. 5. 5.]
 [5. 5. 5.]]
```

copy():

```
array_copy = sorted_array.copy()
array_copy[:] = 5
print(array_copy)
print(sorted_array)
```

copy() crea un array independiente

```
[[5. 5. 5.]
 [5. 5. 5.]
 [5. 5. 5.]]
[[2. 4. 2.]
 [4. 4. 2.]
 [4. 9. 9.]]
```

REPASO

1) Convertir Listas en Arrays

2) Multidimensionalidad en los Arrays

3) Crear Arrays sin usar Listas

4) Crear Arrays unidad

5) Reasignar el contenido de los Arrays

5) Ordenar el contenido de los Arrays

CONQUER BLOCKS