**5. Aleatoridad**

**En esta clase introducimos conceptos y técnicas relacionadas al azar. La aleatoriedad puede ser un gran aliado para realizar cálculos de fenómenos estocásticos pero también de fenómenos deterministas. Introducimos los números pseudoaleatorios, así como el módulo numpy y sus métodos más sencillos.**

**Presentamos una versión libre de un ejercicio lindísimo del curso Exactas Programa que te guía para que resuelvas la pregunta ¿cuántos paquetes de figuritas tengo que comprar para llenar un álbum? con un enfoque estadístico usando el**[**método de Montecarlo**](https://es.wikipedia.org/wiki/M%C3%A9todo_de_Montecarlo)**.**

**Introducimos los arrays de la biblioteca numpy en una sección un poco técnica pero importante para el futuro.**

**A lo largo de la clase iremos haciendo nuestros primeros gráficos en Python. Cerramos introduciendo los scatterplots que permiten visualizar dos variables en simultáneo y facilitan el análisis exploratorio de datos.**

* [**5.1 Random**](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/01_Random.md)
* [**5.2 NumPy**](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/02_NumPy_Arrays.md)
* [**5.3 El album de Figuritas**](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/03_Figuritas.md)
* [**5.4 Gráficos del Arbolado porteño**](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/04_Arboles3_plt.md)
* [**5.5 Cierre de la clase**](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/05_Cierre.md)

**5.1 Random**

En esta sección veremos algunas de las funciones del módulo random. Este módulo se usa para generar valores pseudo-aleatorios. Desde el punto de vista práctico, usaremos estos valores como perfectamente aleatorios --al ser la computadora una máquina determinística sabemos que esto no es completamente cierto. De hecho, en lo que sigue, por simplicidad, omitiremos el prefijo pseudo y hablaremos de números aleatorios aunque no lo sean exactamente.

**Valores discretos**

Podemos generar números enteros aleatorios entre dos extremos. Por ejemplo, para simular la tirada de un dado podemos generar un número entre 1 y 6.

import random

dado = random.randint(1,6) # devuelve un entero aleatorio entre 1 y 6

Si queremos simular una primera tirada del juego [la generala](https://es.wikipedia.org/wiki/Generala) tendremos que generar cinco valores al azar:

import random

tirada=[]

for i in range(5):

tirada.append(random.randint(1,6))

print(tirada)

**Ejercicios:**

**Ejercicio 5.1: Generala servida**

Queremos estimar la probabilidad de obtener una generala servida (cinco dados iguales) en una tirada de dados. Podemos hacer la cuenta usando un poco de teoría de probabilidades, o podemos *simular* que tiramos los dados muchas veces y ver cuántas de esas veces obtuvimos cinco dados iguales. En este ejercicio vamos a usar el segundo camino.

Escribí una función tirar() que devuelva una lista con cinco dados generados aleatoriamente. Escribí otra función llamada es\_generala(tirada) que devuelve True si y sólo si los cinco dados de la lista tirada son iguales.

Luego analizá el siguiente código. Correlo con N = 100000 varias veces y observá los valores que obtenés. Luego correlo algunas veces con N = 1000000 (ojo, hace un millón de experimentos, podría tardar un poco):

G = sum([es\_generala(tirar()) for i in range(N)])

prob = G/N

print(f'Tiré {N} veces, de las cuales {G} saqué generala servida.')

print(f'Podemos estimar la probabilidad de sacar generala servida mediante {prob:.6f}.')

¿Por qué varían más los resultados obtenidos con N = 100000 que con N = 1000000? ¿Cada cuántas tiradas en promedio podrías decir que sale una generala servida? ¿Cómo se puede calcular la probabilidad de forma exacta?

**Ejercicio 5.2: Generala no necesariamente servida**

Si uno juega con las reglas originales (se puede volver a tirar algunos de los cinco dados hasta dos veces, llegando hasta a tres tiradas en total) siguiendo una estrategia que intente obtener generala (siempre guardar los dados que más se repiten y tirar nuevamente los demás) es más probable obtener una generala que si sólo consideramos la generala servida. Escribí un programa que estime la probabilidad de obtener una generala en las tres tiradas de una mano y guardalo en un archivo generala.py.

*Extra:* Hay gente que, si en la primera tirada le salen todos dados diferentes, los mete al cubilete y tira los cinco nuevamente. Otras personas, eligen uno de esos dados diferentes, lo guardan, y tiran sólo los cuatro restantes. ¿Podés determinar, por medio de simulaciones, si hay una de estas estrategias que sea mejor que la otra?

**Semillas**

A veces queremos generar números (pseudo-)aleatorios de una manera reproducible. Puede sonar contradictorio, pero no lo es: es aquí donde se ve claramente la naturalez **pseudo**aleatoria de estos números. Si fijamos una semilla con el comando random.seed(semilla), donde semilla es un número entero, la secuencia de números aleatorios que obtengamos será reproducible utilizando la misma semilla.

Probá por ejemplo correr dos veces el siguiente código:

import random

random.seed(31415)

tirada=[]

for i in range(5):

tirada.append(random.randint(1,6))

print(tirada)

**Elecciones con reposición**

A veces queremos elegir al azar un elemento de una lista y no solo un número. En el caso que vimos recién de los dados, nuestra lista sería [1, 2, 3, 4, 5, 6] pero podría ser también ['uno', 'dos', 'tres', 'cuatro', 'cinco', 'seis'].

La función random.choice() toma una secuencia y devuelve un elemento aleatorio.

caras = ['uno', 'dos', 'tres', 'cuatro', 'cinco', 'seis']

print(random.choice(caras))

Si queremos realizar múltiples elecciones aleatorias de la lista podemos usar la función random.choices()

print(random.choices(caras,k=5))

Estos son experimentos *con reposición* en el sentido de que si en el primer dado sacamos un dos, al tirar el segundo dado podemos sacar otro dos, repitiendo el valor. El término *reposición* viene de pensar en una urna con bolitas. Si un dado lo pensamos como una urna con seis bolitas (etiquetadas del uno al seis), luego de sacar una bolita (tirar el dado una vez) *reponemos* la bolita que sacamos, de forma que en el siguiente experimento (tirar nuevamente el dado) podamos obtener el mismo valor.

**Elecciones sin reposición**

Si queremos modelar un juego con un mazo de naipes, es natural modelarlo sin reposición. Cuando le damos tres cartas a un jugador la segunda carta no puede ser igual a la primera y la tercera será diferente de las dos anteriores.

En un mazo de naipes españoles, cada carta tiene un palo y un valor. El mazo tiene 40 naipes. Los palos son oro, copa, espada y basto y los valores van del 1 al 7 y de del 10 al 12. Usaremos una comprensión doble de listas para generar los naipes (todas las combinaciones posibles de valores y palos).

valores = [1, 2, 3, 4, 5, 6, 7, 10, 11, 12]

palos = ['oro', 'copa', 'espada', 'basto']

naipes = [(valor,palo) for valor in valores for palo in palos]

Ahora podemos usar random.choice(naipes) para seleccionar un naipe. Sin embargo, si usáramos random.choices(naipes, k=3) para seleccionar tres naipes para un jugador, podríamos estar repitiendo el mismo naipe más de una vez, lo que es incorrecto. En este caso tenemos que usar elecciones múltiples *sin reposición*. Para eso usamos la función sample del módulo random: random.sample(naipes,k=3).

A diferencia de choices donde el parámetro k podía tomar cualquier valor, al dar la instrucción random.sample(naipes,k=?) la variable k no puede ser mayor que la cantidad de naipes (es decir 40) ya que no se puede sacar *sin reposición* más elementos que la cantidad total.

**Ejercicio 5.3: Envido**

Teniendo en cuenta las reglas del [Truco](https://es.wikipedia.org/wiki/Truco_argentino), estimá la probabilidad de obtener 31, 32 o 33 puntos de envido en una mano. ¿Son iguales estas tres probabilidades? ¿Por qué?

*Observación: como corresponde, en esta materia jugamos al truco****sin****flor. Si no conocés las reglas del Truco y no te dan ganas de aprenderlo ahora, simplemente salteá este ejercicio.*

Guardá este ejercicio en un archivo envido.py para entregar.

**Mezclar**

La última función que queremos introducir es útil en muchos contextos. En los juegos de naipes, para continuar con nuestro ejemplo, es muy usual mezclar el mazo entero antes de repartir. En Python usamos la función shuffle del módulo random.

valores = [1, 2, 3, 4, 5, 6, 7, 10, 11, 12]

palos = ['oro', 'copa', 'espada', 'basto']

naipes = [(valor,palo) for valor in valores for palo in palos]

random.shuffle(naipes)

print(naipes)

Observá que la función shuffle() modificó la lista que le pasamos como parámetro. Una vez mezclado el mazo, podemos consultar las tres cartas que quedaron al final:

naipes[-3:]

o directamente sacarlas del mazo:

n1 = naipes.pop()

n2 = naipes.pop()

n3 = naipes.pop()

print(f'Repartí el {n1[0]} de {n1[1]}, el {n2[0]} de {n2[1]} y el {n3[0]} de {n3[1]}. Quedan {len(naipes)} naipes en el mazo.')

**Valores continuos**

Además de generar valores (pseudo)aleatorios discretos, también es posible generar valores continuos. La funcion random.random() genera un número de punto flotante entre 0 y 1.

**Ejercicio 5.4: Calcular pi**

Es interesante ver cómo los algoritmos estocásticos (basados en elecciones aleatorias) también sirven para resolver problemas que no tienen nada de estocásticos. En este ejercicio vas a usar el generador random() para aproximar pi.

Por definición pi es el área del círculo de radio uno. Si generamos puntos (x,y) con:

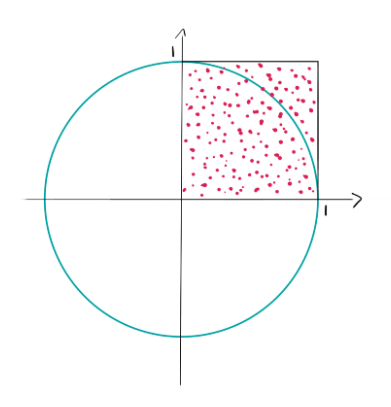
def generar\_punto():

x = random.random()

y = random.random()

return x,y

tendremos puntos dentro del cuadrado [0, 1]x[0, 1]. Algunos de estos puntos del cuadrado caerán dentro del círculo unitario (los que cumplan que x^2 + y^2 < 1) y otros puntos caerán afuera. La proporción de puntos que caigan dentro del cuarto de círculo guardará relación con la proporción entre el área del cuarto de círculo y el área del cuadrado. Obviamente hay una componente aleatoria, pero a medida que la cantidad de puntos crece, la proporción de puntos se acercará a la proporción entre las dos áreas.

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/cuadrante_circ.png)

Si el área del círculo completo es pi, el área de nuestro cuarto de círculo es pi/4. Por otro lado el área del cuadrado unitario es 1. Por lo tanto, si generamos N puntos con una distribución uniforme en el cuadrado unitario, esperamos que pi/4 de estos N puntos caigan dentro del cuarto del círculo y el resto afuera. Es decir que, si llamamos M al número de puntos que caen dentro del círculo, esperamos que M ~(pi/4 \* N).

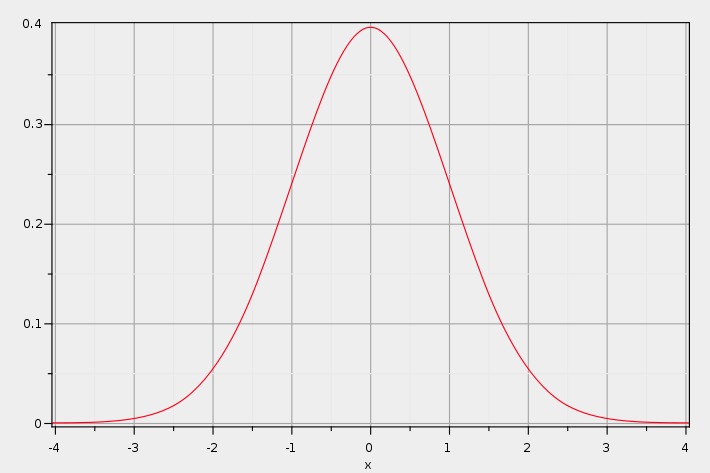
Despejando pi de esta estimación, obtenemos que pi ~ 4\*M/N. Esto nos permite estimar pi mirando cuántos puntos caen realmente dentro del círculo del total de puntos.

Escribí un programa estimar\_pi.py que genere cien mil puntos aleatorios con la función generar\_punto(), calcule la proporción de estos puntos que caen en el círculo unitario (usando ¿x^2 + y^2 < 1?) y use este resultado para dar una aproximación de pi.

**Ejercicio 5.5: Gaussiana**

Con random.random() generamos valores aleatorios entre 0 y 1 con una distribución *uniforme*. En esa distribución, todos los valores posibles tienen la misma probabilidad de ser seleccionados. También es posible generar valores aleatorios con otras distribuciones. Una de las distribuciones más importantes es la **distribución normal** o [Gaussiana](https://es.wikipedia.org/wiki/Distribuci%C3%B3n_normal).

La distribución normal tiene dos parámetros, denominados media y desvío estándar y denotados usualmente con las letras griegas *mu* y *sigma*, respectivamente.

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/normal.jpg)

La función random.normalvariate(mu,sigma) genera números aleatorios según esta distribución de probabilidades. Por ejemplo, usando mu = 0 y sigma = 1 podemos generar 6 valores aleatorios así:

>>> for i in range(6):

print(f'{random.normalvariate(0,1):.2f}', end=', ')

-0.60, 0.06, -1.33, -0.62, -0.81, 0.63,

La distribución normal tiene muchos usos. Uno de ellos es modelar errores experimentales, es decir la diferencia entre el valor medido de una magnitud física y el valor real de dicha magnitud.

Hagamos algún ejercicio sencillo antes de terminar. Supongamos que una persona se compra un termómetro que mide la temperatura con un error aleatorio normal con media 0 y desvío estándar de 0.2 grados (error gaussiano). Si la temperatura real de la persona es de 37.5 grados, simulá usando normalvariate() (con mu y sigma adecuados) n = 99 valores medidos por el termómetro.

Imprimí los valores obtenidos en las mediciones de temperatura simuladas y luego, como resumen, cuatro líneas indicando el valor máximo, el mínimo, el promedio y la mediana de estas n mediciones. Guardá tu programa en el archivo termometro.py.

*Para encontrar el máximo y mínimo podés usar y agrandar tu código de busqueda\_en\_listas.py o usar las primitivas max() y min() de Python. El****promedio****es la suma de los valores dividido su cantidad; podés programarla desde cero o usar la primitiva sum() y un cociente por n. Finalmente, la****mediana****de una cantidad impar de valores es el valor en la posición central cuando los datos están ordenados. Acá podés usar el método sort() de listas. Y ya que estamos, ¿se te ocurre cómo encontrar los*[*cuartiles*](https://es.wikipedia.org/wiki/Cuartil)*?*

**5.2 NumPy**

Esta es una introducción a la biblioteca NumPy (**Numerical Python**) de Python. Se trata de una colección de módulos de código abierto que tiene aplicaciones en casi todos los campos de las ciencias y de la ingeniería. Es el estándar para trabajar con datos numéricos en Python. Muchas otras bibliotecas y módulos de Python como Pandas, SciPy, Matplotlib, scikit-learn, scikit-image usan numpy.

Esta biblioteca permite trabajar cómodamente con matrices multidimensionales por medio del tipo **ndarray**, un objeto n-dimensional homogéneo (es decir, con todas sus entradas del mismo tipo), y con métodos para operar eficientemente sobre él. numpy puede usarse para una amplia variedad de operaciones matemáticas sobre matrices. Le agrega a Python estructuras de datos muy potentes sobre las que puedés hacer cálculos y operar matemáticamente con eficiencia y a un alto nivel.

**Instalar e importar numpy**

Cuando quieras usar numpy en Python, primero tenés que importarlo:

import numpy as np

Acortamos numpy a np para ahorrar tiempo y mantener el código estandarizado. Todes escriben np.

Si no lo tenés instalado (te dará un error al importarlo) podés instalarlo escribiendo alguno de los siguientes comandos, según corresponda:

conda install numpy

pip install numpy

pip3 install numpy

**¿Cuál es la diferencia entre listas y arreglos?**

numpy ofrece varias formas muy eficientes de crear vectores y manipular datos numéricos. Mientras que una lista de Python puede contener diferentes tipos de datos en su interior, los elementos de un vector numpy serán todos del mismo tipo. De esta forma numpy garantiza un muy alto rendimiento en las operaicones matemáticas.

Además, los arreglos están pensados para tener un tamaño fijo, mientras que las listas están diseñadas para agregar y sacar elementos. Son estructuras de datos similares desde un punto de vista superficial, pero muy diferentes en cuanto a las posibilidades que brindan.

Las operaciones matemáticas sobre vectores de numpy son más rápidas que sobre listas. Además los vectores ocupan menos memoria que las listas análogas. En cambio, modificar el tamaño de una lista es algo muy sencillo mientras que el de un vector es costoso. Y combinar diferentes tipos de datos es sencillo en las listas pero imposible en los vectores de numpy.

**Arreglos n-dimensionales**

Los vectores (unidimensionales) y matrices (bidimensiones) se generalizan a arreglos n-dimensionales. Esta estructura de datos es la central de la biblioteca numpy. Un arreglo (ndarray) tiene una grilla de valores (datos crudos) junto con información sobre cómo ubicarlos y cómo interpretarlos. Los elementos de esta grilla pueden ser indexados de diversas maneras y, como ya dijimos, son todos del mismo tipo. Este tipo es frecuentemente abreviado como dtype (por data type).

Un arreglo puede ser indexado por tuplas de enteros no negativos, por variables booleanas, por otro arreglo o por enteros. El rango (rank) de un arreglo es su número de dimensiones. Su forma (shape) es una tupla de enteros que dice su tamaño en cada dimensión.

Una forma de inicializar un arreglo de numpy es mediante una lista de números. Esto nos da un vector (arreglo de dimensión uno). Usando listas anidadas, podemos definir arreglos de más altas dimensiones.

Por ejemplo:

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

o:

>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])

Podemos acceder a los elementos de un arreglo usando corchetes. Acordate que los índices comienzan a contar en 0. Esto significa que si querés acceder al primer elemento, vas a acceder al elemento “0”.

>>> print(a[0]) # si tiene múltiples dimensiones, esto me da una "rebanada" de una dimensión menos

[1 2 3 4]

>>> print(a[2]) # otra rebanada

[ 9, 10, 11, 12]

>>> print(a[2][3]) # accedo al cuarto elemento del tercer vector de a

12

>>> print(a[2,3]) # o, equivalentemente, accedo al elemento en la tercera fila y cuarta columna de a

12

**Más información sobre arreglos**

Ocasionalmente vas a ver que alguien se refiere a un arreglo como un “ndarray” que es una forma breve de decir arreglo n-dimensional. Un arreglo n-dimensional es simplemente un arreglo con n dimensiones. Recordemos que cuando son unidimensionales los llamamos vectores y si son bidimensionales los llamamos matrices.

**¿Qué atributos tiene un arreglo?**

Un arreglo es usualmente un contenedor de tamaño fijo de elementos del mismo tipo. Su forma (shape) es una tupla de enteros no negativos que especifica el tamaño del arreglo en cada dimensión. Un arreglo tiene tantas dimensiones como coordenadas en la tupla.

En numpy, las dimensiones se llaman **axes** (ejes). Esto significa que si tenés un arreglo bidimensional que se ve así:

[[0., 0., 0.],

[1., 1., 1.]]

el arreglo tendrá dos ejes. El primer eje tiene tamaño dos, el segundo tamaño tres (sí, se cuentan primero filas, luego columnas).

De la misma forma que los otros objetos contenedores de Python, los elementos de un arreglo pueden ser accedidos y modificados usando índices y rebanadas.

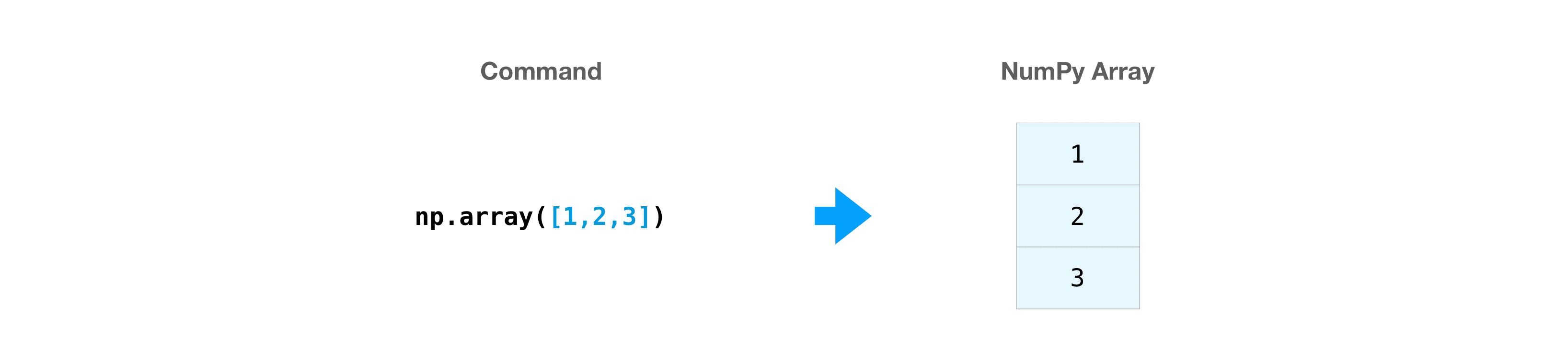
**Crear un arreglo básico**

Para crear un arreglo de numpy podés usar la función np.array(). Lo único que necesitás es pasarle una lista. Si querés, podés especificar el tipo de datos que querés que tenga.

>>> import numpy as np

>>> a = np.array([1, 2, 3])

Vamos a representar la creación con este gráfico:

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/np_array.png)

*Ojo, estas visualizaciones son simplificaciones para representar lo que esta pasando y darte un entendimiento básico de los conceptos y mecanismos de numpy. Los arreglos y sus operaciones tienen aspectos más complejos que los que quedan capturados en estos dibujitos.*

Además de crear un arreglo a partir de una secuencia de elementos, podés crear un arreglo lleno de 0’s:

>>> np.zeros(2)

array([0., 0.])

O uno lleno de 1’s:

>>> np.ones(2)

array([1., 1.])

¡O incluso uno no inicializado! La función empty crea un arreglo cuyo contenido inicial depende del estado de la memoria. Lo bueno de usar empty en lugar de zeros (o ones) es la velocidad - al no inicilizar los valores no perdemos tiempo. ¡Pero asegurate de ponerle valores con sentido luego!

>>> # Crea un arreglo con dos elementos

>>> np.empty(2)

array([ 3.14, 42. ]) # puede variar

También podés crear vectores a partir de un rango de valores:

>>> np.arange(4)

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

También un vector que contiene elementos equiespaciados, especificando el **primer número**, el **límite**, y el **paso**.

>>> np.arange(2, 9, 2) # o np.arange(2, 10, 2)

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

El límite derecho nunca está en la lista.

También podés usar np.linspace() para crear un vector de valores equiespaciados especificando el **primer número**, el **último número**, y la **cantidad** de elementos:

>>> np.linspace(0, 10, num=5)

array([ 0. , 2.5, 5. , 7.5, 10. ])

**Ejercicio 5.6: arange() y linspace()**

Generá un vector que tenga los números impares entre el 1 y el 19 inclusive usando arange(). Repetí el ejercicio usando linspace(). ¿Qué diferencia hay en el resultado?

**Especificar el tipo de datos**

Si no lo especificás, el tipo de datos (por omisión) de los arreglos es el punto flotante (np.float64). Sin embargo, podés explicitar otro tipo de datos usando la palabra clave dtype.

>>> x = np.ones(2, dtype=np.int64)

>>> x

array([1, 1])

En estos dos casos el 64 de los tipos de datos se refiere a la cantidad de bits usados para representar el número en el sistema binario: 64 bits.

**Agregar, borrar y ordenar elementos**

Ordenar un vector es sencillo usando np.sort(). Por ejemplo, si comenzás con este vector:

>>> arr = np.array([2, 1, 5, 3, 7, 4, 6, 8])

Podés ordenar sus elementos con:

>>> np.sort(arr)

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

Fijate que el vector arr quedó desordenado. sort simplemente devolvió una copia ordenada de los datos pero no modificó el original.

Otra operación usual es la concatenación. Si empezás con estos dos vectores:

>>> a = np.array([1, 2, 3, 4])

>>> b = np.array([5, 6, 7, 8])

los podés concatenar usado np.concatenate().

>>> np.concatenate((a, b))

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

Un ejemplo un poco más complejo es el siguiente:

>>> x = np.array([[1, 2], [3, 4]])

>>> y = np.array([[5, 6]])

Los podés concatenar usando:

>>> np.concatenate((x, y), axis=0)

array([[1, 2],

[3, 4],

[5, 6]])

**Conocer el tamaño, dimensiones y forma de un arreglo**

ndarray.ndim te dice la cantidad de ejes (o dimensiones) del arreglo.

ndarray.shape te va a dar una tupla de enteros que indican la cantidad de elementos en cada eje. Si tenés una matriz con 2 filas y 3 columnas de va a dar (2, 3).

ndarray.size te dice la cantidad de elementos (cantidad de números) de tu arreglo. Es el producto de la tupla shape. En el ejemplo del renglón anterior, el size es 6.

Por ejemplo, si creás este arreglo de tres dimensiones:

>>> array\_ejemplo = np.array([[[0, 1, 2, 3],

... [4, 5, 6, 7]],

...

... [[0, 1, 2, 3],

... [4, 5, 6, 7]],

...

... [[0 ,1 ,2, 3],

... [4, 5, 6, 7]]])

Vas a tener

>>> array\_ejemplo.ndim # cantidad de dimensiones

3

>>> array\_ejemplo.shape # cantidad de elementos en cada eje

(3, 2, 4)

>>> array\_ejemplo.size # total de elementos 3\*2\*4

24

**Cambiar la forma de un arreglo**

Usando arr.reshape() le podés dar una nueva forma a tu arreglo sin cambiar los datos. Solo tené en cuenta que antes y después del reshape el arreglo tiene que tener la misma cantidad de elementos. Por ejemplo, si comenzás con un arreglo con 12 elementos, tendrás que asegurarte que el nuevo arreglo siga teniendo 12 elementos.

Por ejemplo:

>>> a = np.arange(6)

>>> print(a)

[0 1 2 3 4 5]

Podés usar reshape() para cambiarle la forma y que en lugar de ser un vector de 6 elementos, sea una matriz de 3 filas y dos columnas:

>>> b = a.reshape(3, 2)

>>> print(b)

[[0 1]

[2 3]

[4 5]]

**Agregar un nuevo eje a un arreglo**

A veces pasa que tenemos un vector con n elementos y necesitamos pensarlo como una matriz de una fila y n columnas o de n filas y una columna. Podés usar np.newaxis para agregarle dimensiones a un vector existente.

Usando np.newaxis una vez podés incrementar la dimensión de tu arreglo en uno. Por ejemplo podés pasar de un vector a una matriz o de una matriz a un arreglo tridimensional, etc.

Por ejemplo, si comenzás con este vector:

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

>>> a.shape

(6,)

Podés usar np.newaxis para agregarle una dimensión y convertirlo en un vector fila:

>>> vec\_fila = a[np.newaxis, :]

>>> vec\_fila.shape

(1, 6)

O, para convertirlo en un vector columna, podés unsertar un eje en la segunda dimensión:

>>> vec\_col = a[:, np.newaxis]

>>> vec\_col.shape

(6, 1)

**Índices y rebanadas**

Podés indexar y rebanar arreglos de numpy como hicimos con las listas.

Para obtener elementos de un arreglo, lo más sencillo es usar los índices para seleccionar los que queremos conservar.

>>> data = np.array([1, 2, 3])

>>> data[1]

2

>>> data[0:2]

array([1, 2])

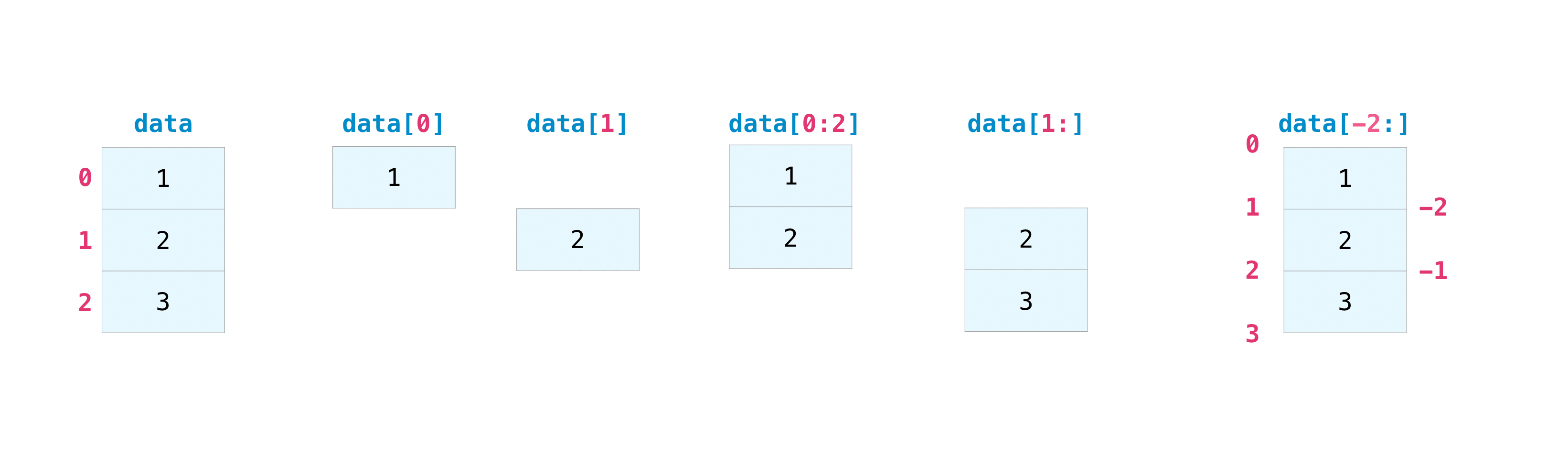
>>> data[1:]

array([2, 3])

>>> data[-2:]

array([2, 3])

Lo podés visualizar así:

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/np_indexing.png)

Otra operación muy útil es seleccionar los elementos que cumplen cierta condición. Por ejemplo, si comenzás con un arreglo así:

>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])

Podés imprimir todos los valores menores que cinco.

>>> print(a[a < 5])

[1 2 3 4]

También podés seleccionar, por ejemplo, aquellos elementos mayores o iguales que 5 y usar el resultado para indexar el arreglo.

>>> five\_up = (a >= 5)

>>> print(a[five\_up])

[ 5 6 7 8 9 10 11 12]

Es interesante que five\_up da un arreglo de valores booleanos. True si satisface la condición y False si no la satisface.

Podés seleccionar los elementos pares:

>>> pares = a[a%2==0]

>>> print(pares)

[ 2 4 6 8 10 12]

Usando los operadores lógicos & y | podés combinar dos o más condiciones.

Ya sea para seleccionar elementos directamente:

>>> c = a[(a > 2) & (a < 11)]

>>> print(c)

[ 3 4 5 6 7 8 9 10]

o para definir una nueva variable booleana:

>>> five\_up = (a > 5) | (a == 5)

>>> print(five\_up)

[[False False False False]

[ True True True True]

[ True True True True]]

Finalmente, podés usar np.nonzero() para obtener las coordenadas de ciertos elementos de un arreglo.

Si empezamos con este arreglo:

>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])

Podés usar np.nonzero() para imprimir los índices de los elementos que son, digamos, menores que 5:

>>> b = np.nonzero(a < 5)

>>> print(b)

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

En este ejemplo, la respuesta es una tupla de arreglos: uno por cada dimensión. El primer arreglo representa las filas de los elementos que satisfacen la condición y el segundo sus columnas.

Si querés generar la lista de coordenadas donde se encuentran estos elementos, podés zipear los arreglos, convertir el resultado en una lista e imprimirla:

>>> lista\_de\_coordenadas = list(zip(b[0], b[1]))

Surge naturalmente la pregunta: ¿porqué tengo que convertir el objeto zip a una lista? Veremos en la segunda mitad de la materia más detalles sobre *generadores* en Python para entender exactamente lo que está pasando aquí. Simplemente digamos que al zipear b[0] y b[1] no se genera la lista realmente, sino potencialmente. Sólo al solicitar sus elementos (iterando sobre ello o con list) se generan realmente las coordenadas.

>>> for coord in lista\_de\_coordenadas:

... print(coord)

(0, 0)

(0, 1)

(0, 2)

(0, 3)

Podés usar np.nonzero() para imprimir o seleccionar los elementos del arreglo que son menores que 5:

>>> print(a[b])

[1 2 3 4]

Si la condición que ponés no la satisface ningún elemento del arreglo entonces el arreglo de índices que obtenés con np.nonzero() será vacío. Por ejemplo:

>>> no\_hay = np.nonzero(a == 42)

>>> print(no\_hay)

(array([], dtype=int64), array([], dtype=int64))

**Crear arreglos usando datos existentes**

Es sencillo crear un nuevo arreglo usando una sección de otro arreglo.

Suponete que tenés este:

>>> a = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

Podés crear otro arreglo a partir de una sección de a, simplemente especificando qué parte querés.

>>> arr1 = a[3:8]

>>> arr1

array([4, 5, 6, 7, 8])

Es importante saber que este método genera una *vista* del arreglo original y no una verdadera copia. Si modificás un elemento de la vista, ¡también se modificará en el original!

>>> arr1[0] = 44

>>> print(a)

[ 1 2 3 44 5 6 7 8 9 10]

El concepto de *vista* es importante para entender lo que está pasando. Las operaciones más frecuentes devuelven vistas y no copias. Esto ahorra memoria y es más veloz, pero si no lo sabés puede traerte problemas.

Veamos este ejemplo:

>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])

Ahora creamos b1 a partir de una rebanada de a y modificamos su primer elemento. ¡Esto va a modificar el elemento correspondiente de a también!

>>> b1 = a[0, :]

>>> b1

array([1, 2, 3, 4])

>>> b1[0] = 99

>>> b1

array([99, 2, 3, 4])

>>> a

array([[99, 2, 3, 4],

[ 5, 6, 7, 8],

[ 9, 10, 11, 12]])

Podés usar el método copy para copiar los datos. Por ejemplo:

>>> b2 = a[1, :].copy()

>>> b2

array([5, 6, 7, 8])

>>> b2[0] = 95

>>> b2

array([95, 6, 7, 8])

>>> a # ¡no se modifica el 5!

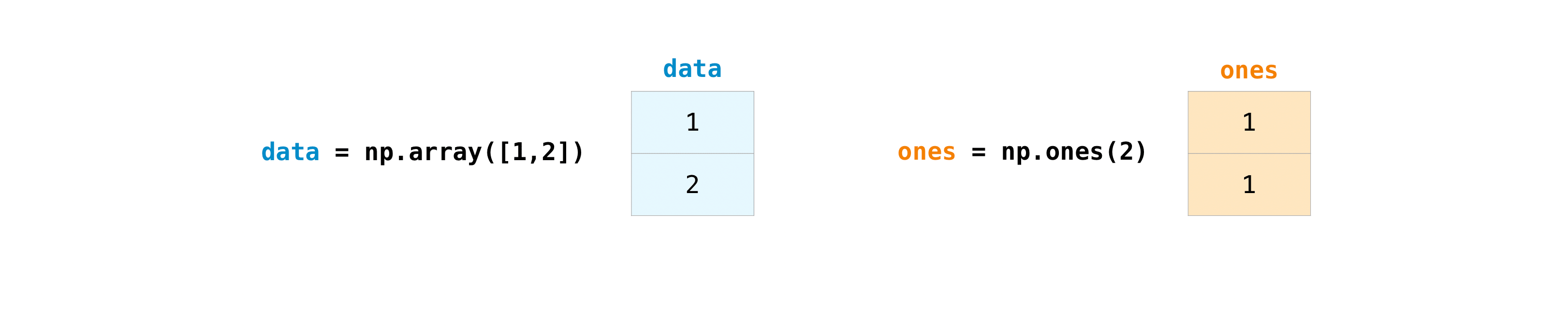
array([[99, 2, 3, 4],

[ 5, 6, 7, 8],

[ 9, 10, 11, 12]])

**Operaciones básicas sobre arreglos**

Una vez que sabés crear arreglos podés empezar a trabajar con ellos. Imaginemos que tenés dos arreglos, uno llamados “data” y otro llamado “ones”.

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/np_array_dataones.png)

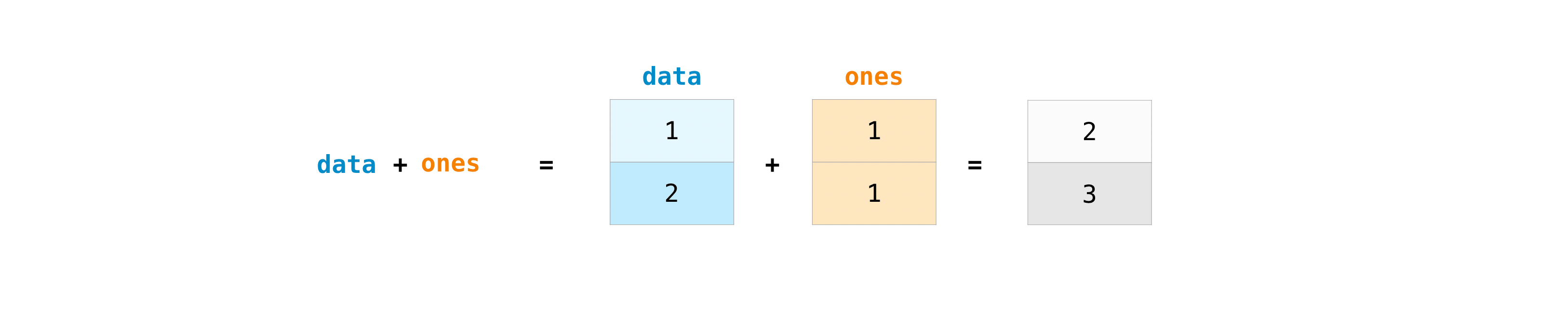
Podés sumarlos con el signo más.

>>> data = np.array([1, 2])

>>> ones = np.ones(2, dtype=int)

>>> data + ones

array([2, 3])

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/np_data_plus_ones.png)

Obviamente, podés hacer otras operaciones.

>>> data - ones

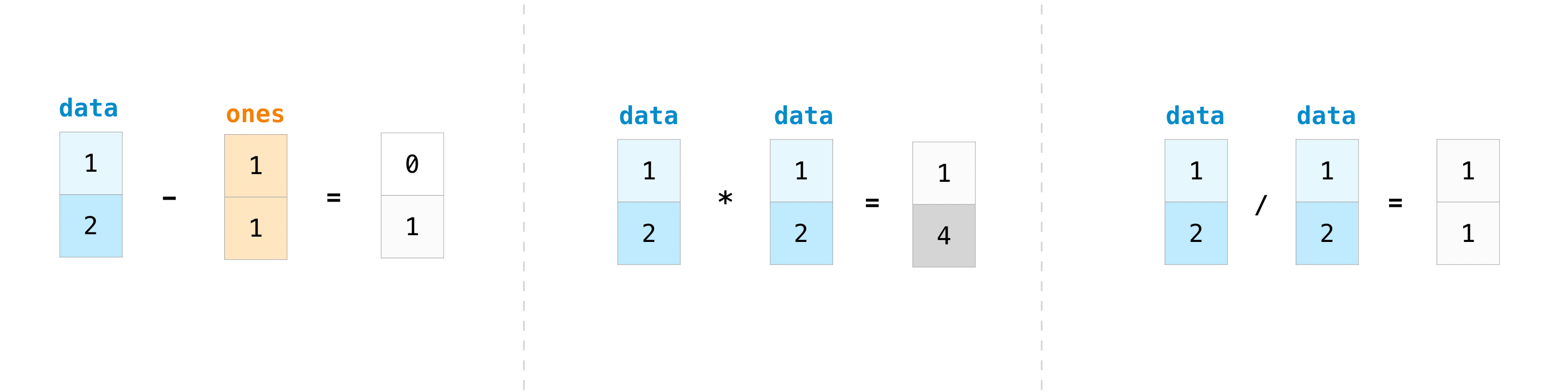
array([0, 1])

>>> data \* data

array([1, 4])

>>> data / data

array([1., 1.])

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/np_sub_mult_divide.png)

Estas operaciones básicas son simples con numpy. Si querés calcular la suma de los elementos del arreglo, podés usar sum(). Esto funciona para vectores, matrices y arreglos de dimensión más alta también.

>>> a = np.array([1, 2, 3, 4])

>>> a.sum()

10

Para sumar los valores por fila o por columna en una matriz, simplemente tenés que especificar el eje sobre el que se hará la suma.

Si tenés la matriz:

>>> b = np.array([[1, 1], [2, 2]])

podés sumar los datos de cada columna con:

>>> b.sum(axis=0)

array([3, 3])

y los datos de cada fila usando:

>>> b.sum(axis=1)

array([2, 4])

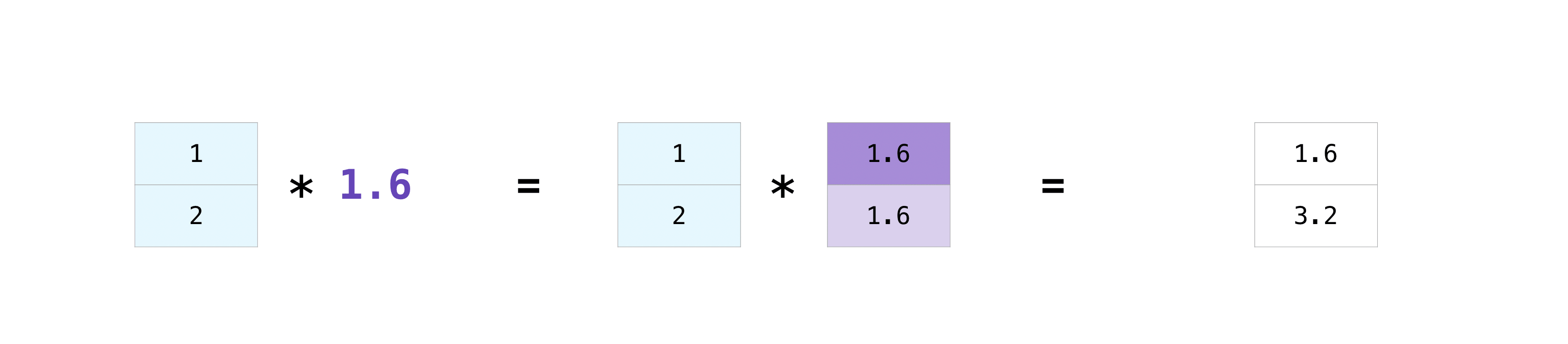
**Broadcasting**

Hay veces en que necesitás realizar una operación entre un arreglo y un número (en matemática le decimos, un *escalar*). Por ejemplo tenés un vector con distancias en millas (lo llamamos "data") y lo necesitás convertir a distancias en kilómetros. Podés hacer esta operación así:

>>> data = np.array([1.0, 2.0])

>>> data \* 1.6

array([1.6, 3.2])

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/np_multiply_broadcasting.png)

numpy entiende que la multiplicación debe ocurrir en cada celda del vector. Este concepto se llama **broadcasting**. El mecanismo de broadcasting le permite a numpy realizar operaciones en arreglos de diferente tamaño, pero los tamaños deben ser compatibles. Por ejemplo si ambos arreglos tienen el mismo tamaño o si uno tiene tamaño 1 (escalar). Si los tamaños no son compatibles, te va a dar un ValueError.

**Operaciones un poco más complejas**

numpy también te permite realizar operaciones que resumen los datos. Además de min, max, y sum, podés usar mean para obtener el promedio, prod para calcular el producto, std para obtener el desvío estándar de los datos, y más.

>>> data.max()

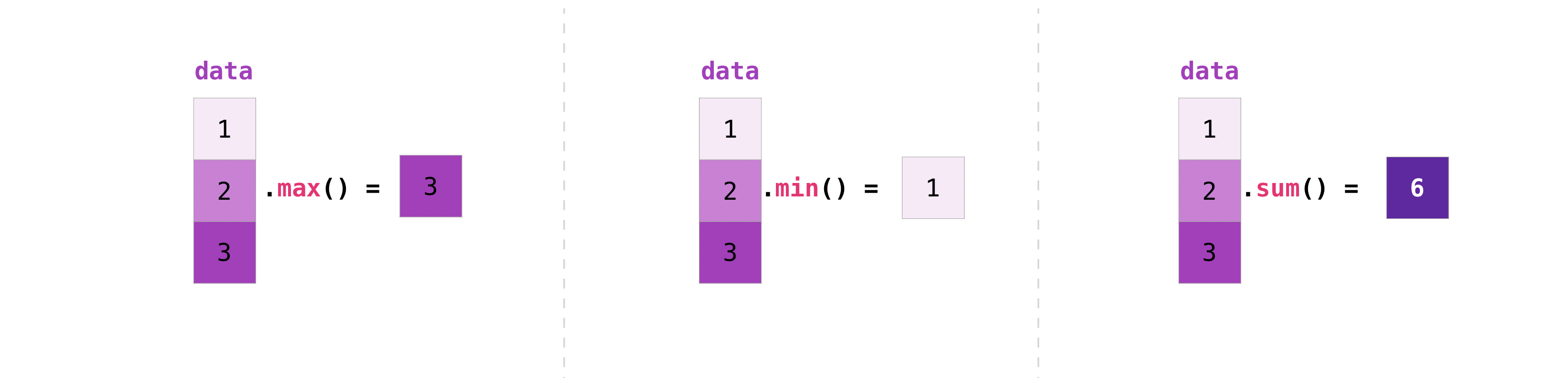
2.0

>>> data.min()

1.0

>>> data.sum()

3.0

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/np_aggregation.png)

Supongamos que tenemos un arreglo, llamado “a”

>>> a = np.array([[0.45053314, 0.17296777, 0.34376245, 0.5510652],

... [0.54627315, 0.05093587, 0.40067661, 0.55645993],

... [0.12697628, 0.82485143, 0.26590556, 0.56917101]])

Es usual procesar los datos por fila o por columna. Si no lo aclarás, numpy procesa los datos de todo el arreglo. Para encontrar la suma o el mínimo del arreglo, usá:

>>> a.sum()

4.8595784

O:

>>> a.min()

0.05093587

Podés especificar qué eje querés considerar. Por ejemplo, para calcular el mínimo de cada columna especificás axis=0.

>>> a.min(axis=0)

array([0.12697628, 0.05093587, 0.26590556, 0.5510652 ])

Son cuatro valores, porque la matriz tiene cuatro columnas.

**Crear matrices**

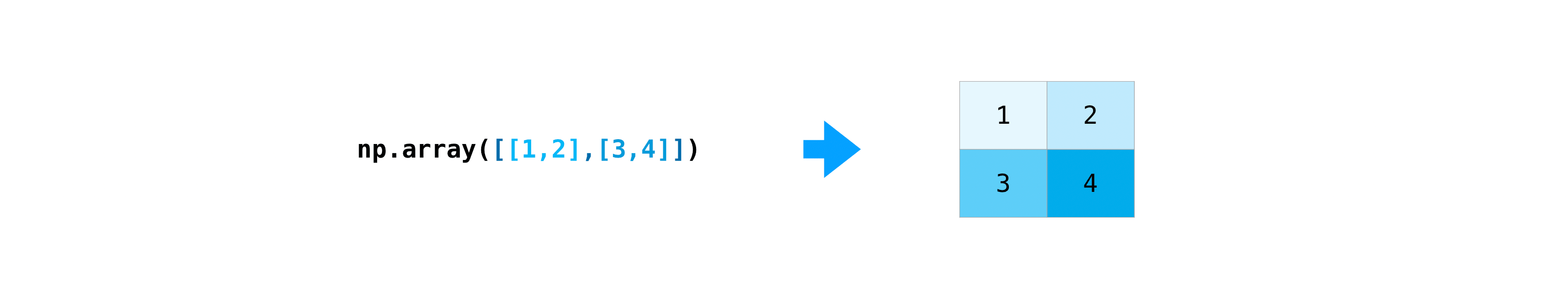
Podés usar listas de listas para crear arreglos bidimensionales (matrices).

>>> data = np.array([[1, 2], [3, 4]])

>>> data

array([[1, 2],

[3, 4]])

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/np_create_matrix.png)

Las técnicas de indexación y rebanadas son muy útiles para manipular matrices:

>>> data = np.array([[1, 2], [3, 4], [5, 6]])

>>> data[0, 1]

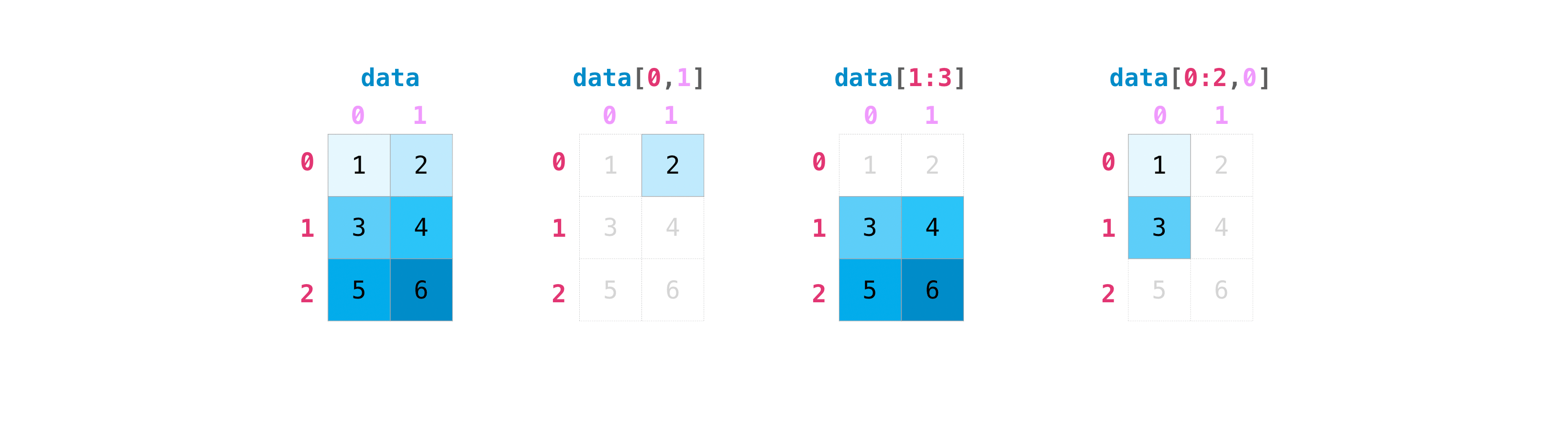
2

>>> data[1:3]

array([[3, 4], [5, 6]])

>>> data[0:2, 0]

array([1, 3])

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/np_matrix_indexing.png)

Podés procesar los datos de matrices como lo hicimos con vectores:

>>> data.max()

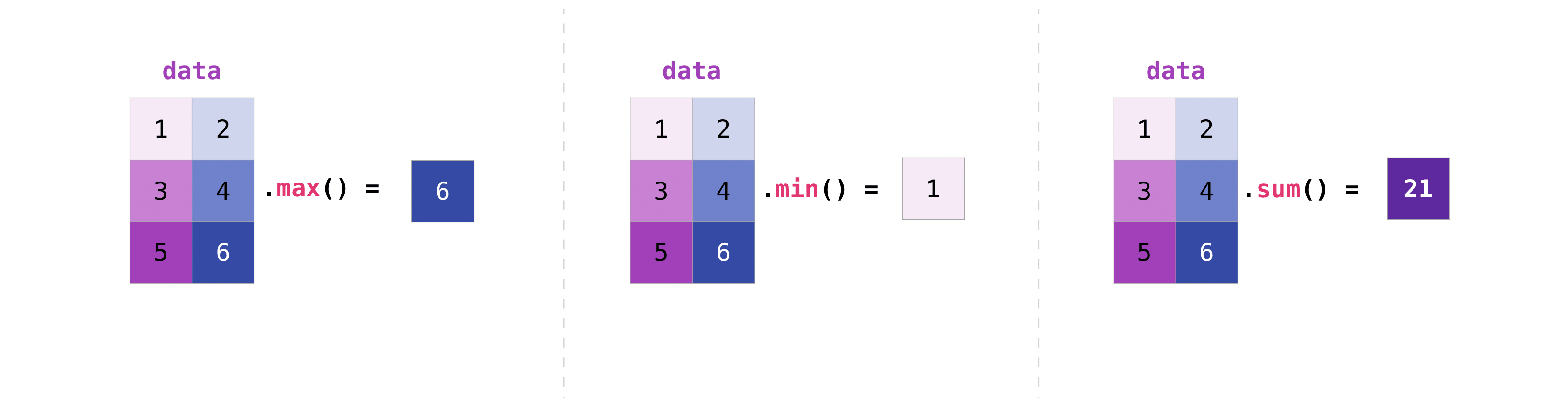
6

>>> data.min()

1

>>> data.sum()

21

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/np_matrix_aggregation.png)

Podés procesar todos, o hacerlo por fila o por columna usando el parámetro axis:

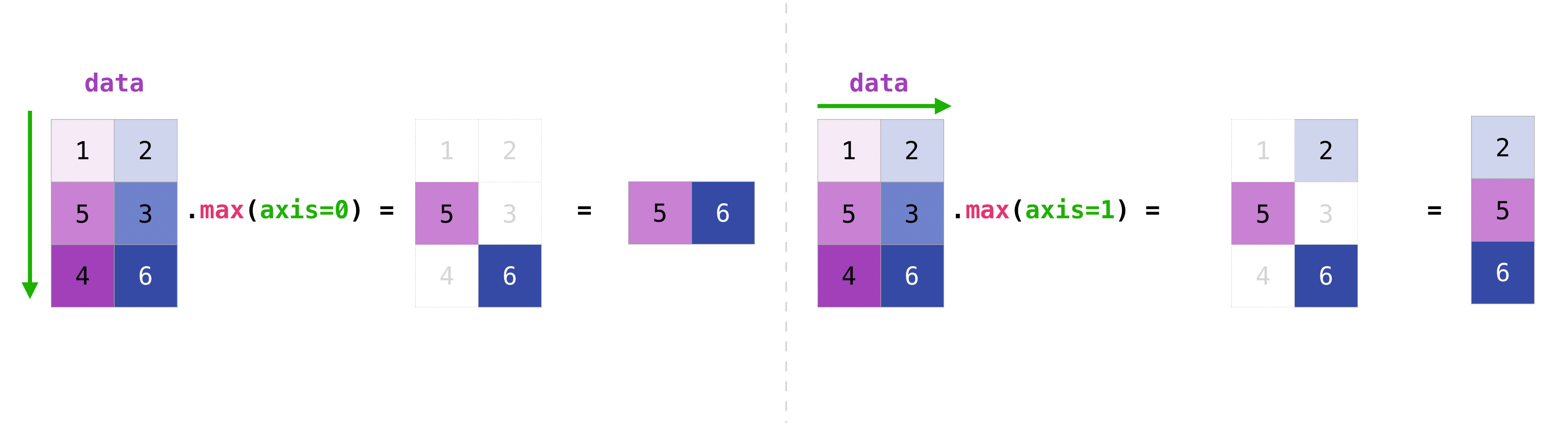
>>> data = np.array([[1, 2], [5, 3], [4, 6]])

>>> data.max(axis=0)

array([5, 6])

>>> data.max(axis=1)

array([2, 5, 6])

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/np_matrix_aggregation_row.png)

Una vez que tenés tus matrices creadas, podés hacer artimética con pares de matrices del mismo tamaño.

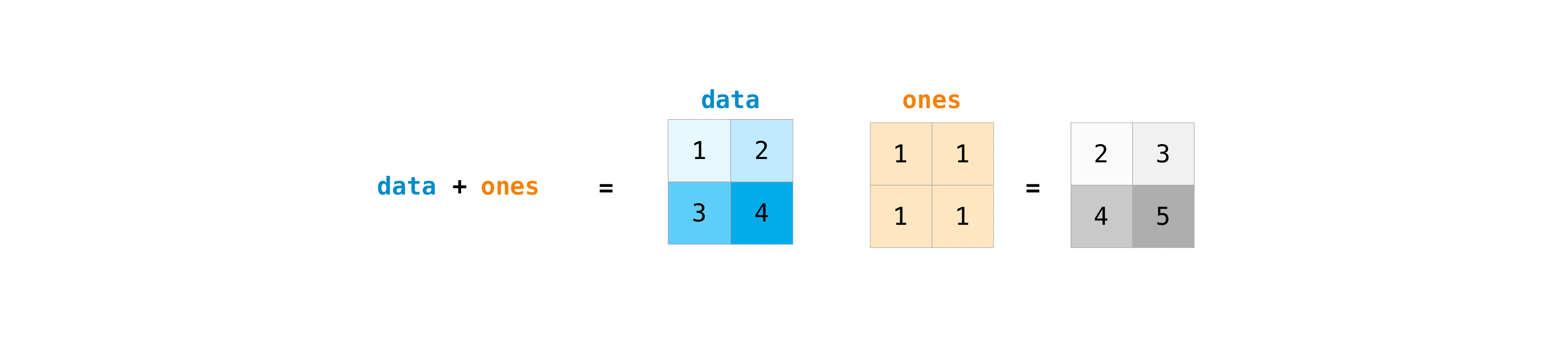
>>> data = np.array([[1, 2], [3, 4]])

>>> ones = np.array([[1, 1], [1, 1]])

>>> data + ones

array([[2, 3],

[4, 5]])

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/np_matrix_arithmetic.png)

También se pueden sumar matrices de tamaños diferentes, pero sólo si una de ellas tiene una sola fila o una sola columna. En este caso, numpy va a usar las reglas de *broadcast* para la operación.

>>> data = np.array([[1, 2], [3, 4], [5, 6]])

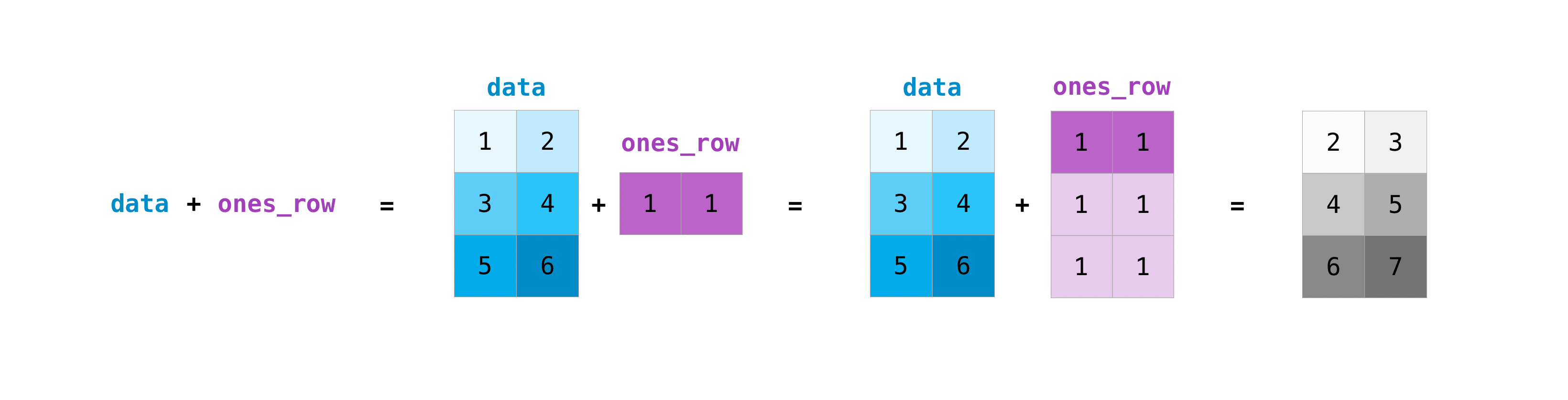
>>> ones\_row = np.array([[1, 1]])

>>> data + ones\_row

array([[2, 3],

[4, 5],

[6, 7]])

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/np_matrix_broadcasting.png)

Tené en cuenta que cuando numpy imprime arreglos n-dimensionales, el último eje se itera más rápido y el primero más lento. Por ejemplo:

>>> np.ones((4, 3, 2))

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

[1., 1.],

[1., 1.]],

[[1., 1.],

[1., 1.],

[1., 1.]],

[[1., 1.],

[1., 1.],

[1., 1.]],

[[1., 1.],

[1., 1.],

[1., 1.]]])

Es frecuente que querramos inicializar los valores de una matriz. numpy ofrece las funciones ones() y zeros(), así como también la clase random.Generator que genera número aleatorios. Sólo hay que pasarle la cantidad de elementos que queremos generar:

>>> np.ones(3)

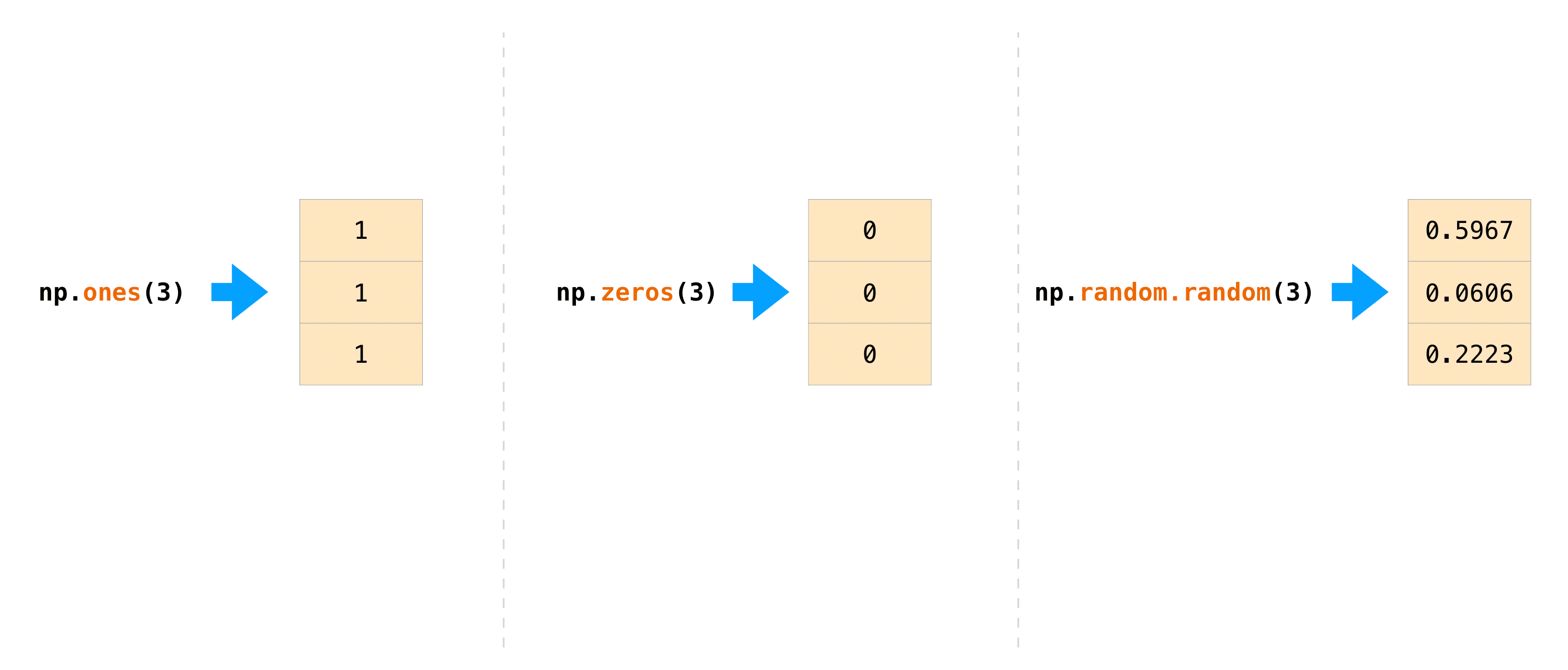
array([1., 1., 1.])

>>> np.zeros(3)

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

>>> np.random.random(3)

array([0.63696169, 0.26978671, 0.04097352]) # puede variar

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/np_ones_zeros_random.png)

También podés usar ones(), zeros(), y random() para crear matrices, si le pasás una tupla describiendo la forma de la matriz:

>>> np.ones((3, 2))

array([[1., 1.],

[1., 1.],

[1., 1.]])

>>> np.zeros((3, 2))

array([[0., 0.],

[0., 0.],

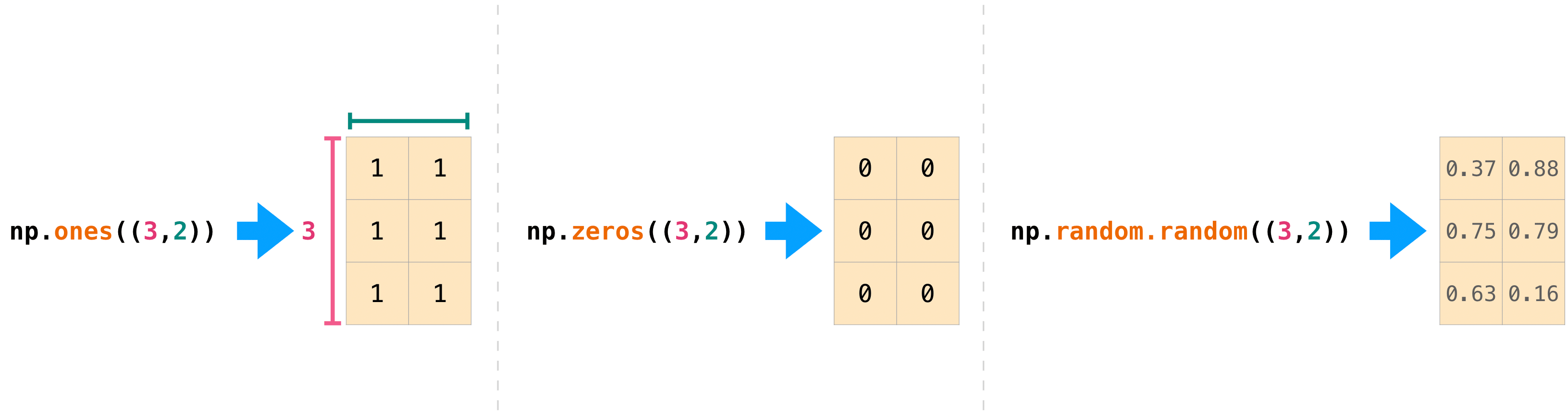
[0., 0.]])

>>> np.random.random((3, 2))

array([[0.01652764, 0.81327024],

[0.91275558, 0.60663578],

[0.72949656, 0.54362499]]) # puede variar

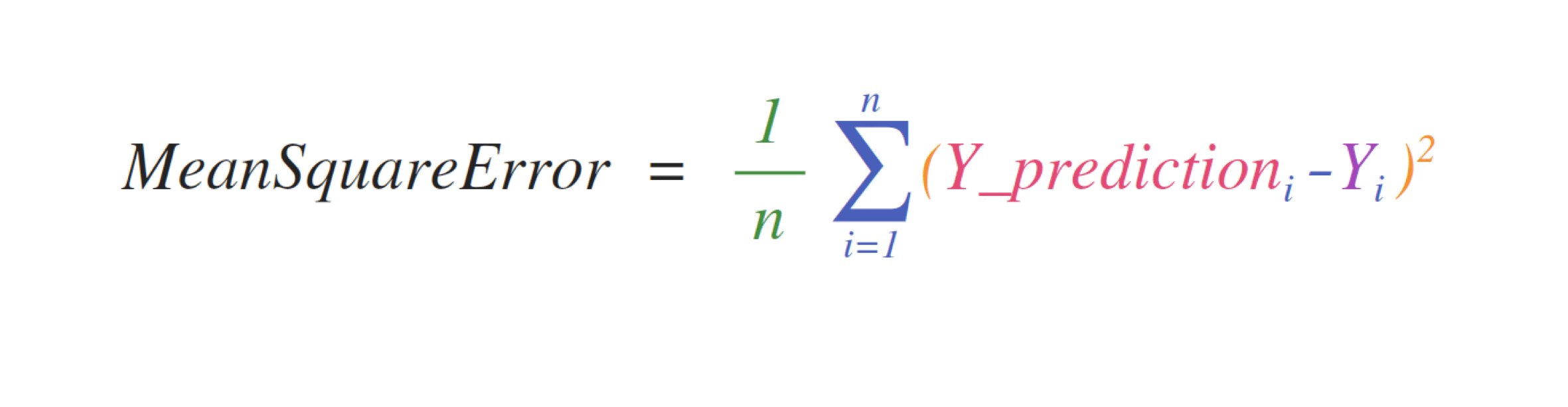
[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/np_ones_zeros_matrix.png)

Esta idea se generaliza a dimensiones más altas.

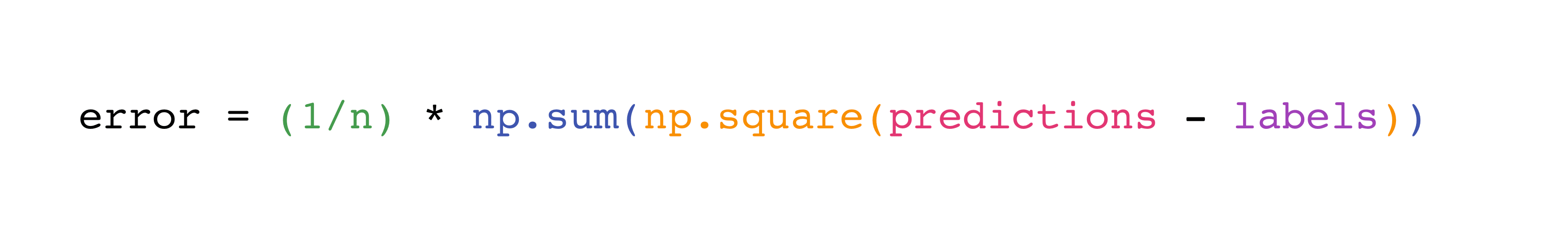
**Fórmulas matemáticas**

La facilidad para implementar fórmulas matemáticas sobre un arreglo es una de las características de numpy que lo hacen tan ampliamente usado en la comunidad científica de Python.

Por ejemplo, ésta es la fórmula del error cuadrático medio:

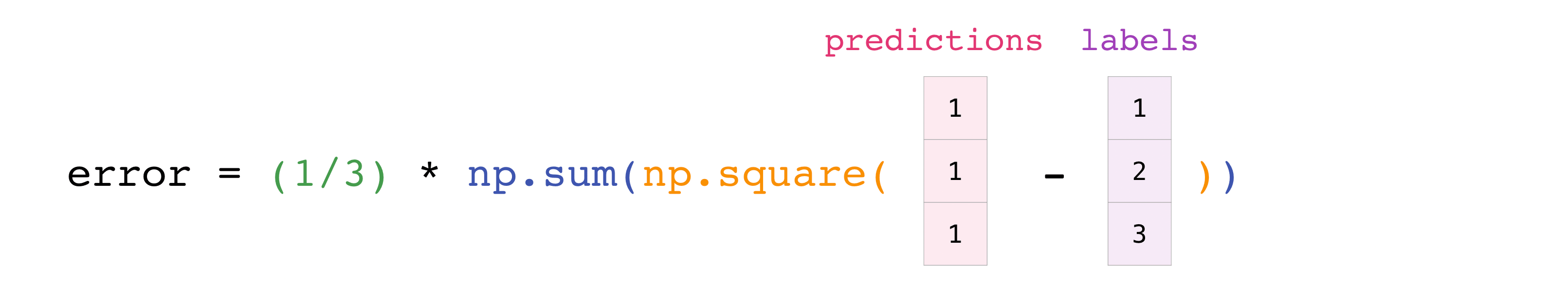
[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/np_MSE_formula.png)

Implementar esta fórmula es simple y directo con numpy:

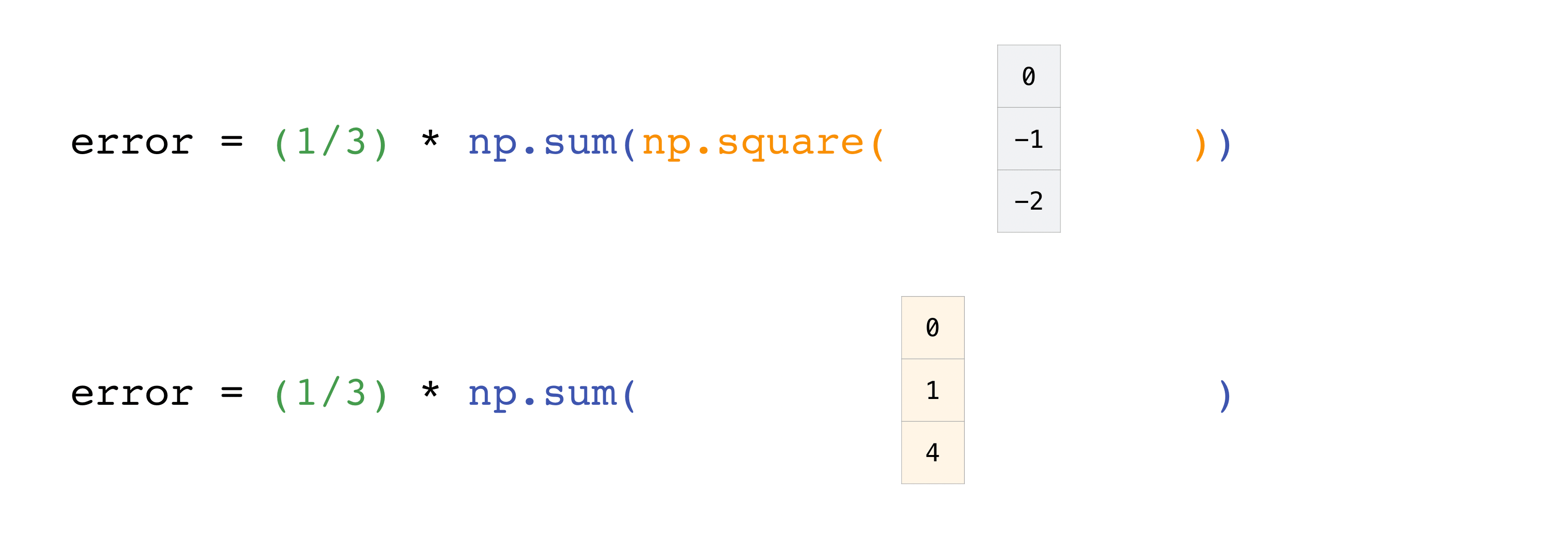
[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/np_MSE_implementation.png)

Lo genial de esta abstracción es que predictions y labels puede tener uno o mil valores. Sólo tienen que tener el mismo tamaño para que todo funcione.

Lo podés visulalizar así:

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/np_mse_viz1.png)

En este ejemplo, tanto las predicciones como las etiquetas tienen tres valores. Es decir n vale tres. Luego de hacer la resta los valores se elevan al cuadrado. Luego numpy suma los valores, divide por tres, y el resultado es el error de esa predicción y puede usarse como un *puntaje* que mide la calidad del modelo que predice.

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/np_mse_viz2.png) [](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/np_MSE_explanation2.png)

**Guardar y cargar objetos de numpy**

Si seguís usando Python después de este curso, es muy probable que en cierto punto quieras guardar tus matrices (o arreglos n-dimensionales) para cargarlas en otro momento sin tener que volver a correr el código que las genera. Hay un par de formas de guardar objetos de numpy. Los objetos ndarray pueden guardarse y leerse de disco con las funciones loadtxt y savetxt usando archivos de texto (tienen la ventaja de que los podés ver con un editor de textos como el [sublime](https://www.sublimetext.com/) o [geany](https://www.geany.org/)), y con las funciones load y save que guardan archivos binarios con extensión **.npy**. Los archivos **.npy** guardan los datos, la forma, el tipo del arreglo y otra información necesaria que permiten reconstruirlos correctamente, incluso en otra máquina con otra arquitectura.

Es sencillo guardar un arreglo con np.save(). Solo asegurate de especificar el arreglo que querés guardar y el nombre del archivo. Por ejemplo, si creás este vector:

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

Lo podés guardar en “filename.npy” con:

>>> np.save('filename', a)

Y lo podés cargar con np.load() para reconstruir tu vector.

>>> b = np.load('filename.npy')

Para verificarlo, usá:

>>> print(b)

[1 2 3 4 5 6]

En formato de texto plano, lo podés guardar como **.csv** o **.txt** con np.savetxt.

Por ejemplo, si tenés este vector:

>>> csv\_arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

lo podés guardar en un archivo .csv con nombre “new\_file.csv” así:

>>> np.savetxt('new\_file.csv', csv\_arr)

Y lo podés cargar fácilmente usando loadtxt():

>>> np.loadtxt('new\_file.csv')

array([1., 2., 3., 4., 5., 6., 7., 8.])

Las funciones savetxt() y loadtxt() aceptan parámetros adicionales para especificar el encabezado y los delimitadores. Si bien los archivos de texto son sencillos para compartir, los archivos .npy (y .npz) son más pequeños y se leen más rápidamente.

**Ejercicio 5.7: Guardar temperaturas**

Ampliá el código de termometro.py que escribiste en el [Ejercicio 5.5](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/01_Random.md#ejercicio-55-gaussiana) para que guarde el vector con las temperaturas simuladas en el directorio Data de tu carpeta de ejercicios, en un archivo llamado Temperaturas.npy. Hacé que corra 999 veces en lugar de solo 99.

**Ejercicio 5.8: Empezando a plotear**

En un rato vamos a empezar a hacer gráficos con Python. Aquí solo un botón de muestra.

Escribí un archivo plotear\_temperaturas.py que lea el archivo de datos Temperaturas.npy con 999 mediciones simuladas que creaste recién y, usando el siguiente ejemplo, hacé un histograma de las temperaturas simuladas:

import matplotlib.pyplot as plt

plt.hist(temperaturas,bins=25)

plt.show() #el show no hace falta en algunos entornos. A veces lo omitiremos.

Ajustá la cantidad de *bins* para que el gráfico se vea lo mejor posible.

**5.3 El album de Figuritas**

**Las figuritas del mundial**

Esta es una adaptación de una actividad que diseñaron nuestres colegas de Exactas-Programa y amablemente nos dejaron usar aquí.

El objetivo de esta actividad es hacer un programa en Python que responda la pregunta: **¿Cuántas figuritas hay que comprar para completar el álbum del Mundial?** Guardá todo lo que hagas en un archivo figuritas.py, te lo vamos a pedir al finalizar la clase.

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/completo.jpg)

Esta pregunta es noticia cada cuatro años:

* [Mundial de Brasil 2014](https://www.pagina12.com.ar/diario/contratapa/13-250187-2014-07-06.html)
* [Mundial de Rusia 2018](https://www.lanacion.com.ar/2125275-rusia-2018-cuantos-sobres-de-figuritas-hacen-falta-para-llenar-el-album-del-mundial)
* Incluso hay un paper que [salió referido en el diario](https://www.infobae.com/2014/05/29/1568512-dos-cientificos-calculan-cuantos-paquetes-hay-que-comprar-completar-el-album-del-mundial/)

**Datos:**

1. Álbum con 670 figuritas.
2. Cada figurita se imprime en cantidades iguales y se distribuye aleatoriamente.
3. Cada paquete trae cinco figuritas.

Vamos a utilizar este disparador para presentar conceptos clave.

**Herramientas útiles de Python**

Para que estén disponibles más funciones de Python, tenés que usar el comando import. En particular, en esta actividad vamos a usar dos módulos:

* El módulo random lo vamos a importar con el comando import random y lo vamos a usar para generar figuritas (pseudo) aleatoriamente.
* El módulo numpy lo vamos a importar con el comando import numpy as np y lo vamos a usar para operar numéricamente.

**El modelo del álbum de figuritas**

Vamos a representar un álbum de n figuritas utilizando un vector de NumPy con posiciones numeradas de 0 a n-1. Cada posición representa el estado de una figurita con dos valores: 0 para indicar que aún no la conseguimos y 1 para indicar que sí (o, si preferís, podés usar un número positivo para representar cuántas de esas figus tenés, contando repes).

Por ejemplo, si tuviéramos un álbum de seis figuritas vacío lo vamos a representar como [0 0 0 0 0 0]. Cuando consigamos la figurita 3 tendremos que indicarlo poniendo un 1 en el tercer lugar de la lista, es decir album[2]=1 y el álbum nos va a quedar [0 0 1 0 0 0], y si queremos representar que nos tocó dos veces la figurita 3, asignamos album[2] += 1 y el álbum queda [0 0 2 0 0 0].

**Primera simplificación**

Suponé por ahora que las figuritas se compran **individualmente** (de a una, no en un paquete con cinco). En este caso, **la dinámica** del llenado es la siguiente:

* Iniciamos con un álbum vacío y sin haber comprado ninguna figurita.
* Compramos figuritas (de a una) hasta llenar el álbum; es decir, se repite la acción (*el paso*) de comprar y pegar figuritas *mientras* (while) el álbum está incompleto.
* Al terminar nos interesa saber cuántas figuritas tuvimos que comprar para llenar el álbum.

**Ejercicios con figus sueltas**

Vamos ahora a implementar computacionalmente este modelo. Queremos definir las funciones:

**Ejercicio 5.9: Crear**

Implementá la función crear\_album(figus\_total) que devuelve un álbum (vector) *vacío* con figus\_total espacios para pegar figuritas.

**Ejercicio 5.10: Incompleto**

¿Cuál sería el comando de Python que nos dice si hay al menos un cero en el vector que representa el álbum? ¿Qué significa que haya al menos un cero en nuestro vector?

Implementá la función album\_incompleto(A) que recibe un vector y devuelve True si el álbum A no está completo y False si está completo.

Esta función y la anterior son realmente sencillas --cada una puede escribirse en una sola línea. En otro contexto quizas podrías usar directamente *esa línea* y evitarte definir la función. Sin embargo, en esta etapa nos parece interesante que organices tu código definiendo estas funciones, por más que tengan línea de código cada una.

**Ejercicio 5.11: Comprar**

Alguna de las funciones que introdujimos en la [Sección 5.1](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/01_Random.md#valores-discretos) sirve para devolver un número entero aleatorio dentro de un rango (¿cuál era?). Implementá una función comprar\_figu(figus\_total) que reciba el número total de figuritas que tiene el álbum (dado por el parámetro figus\_total) y devuelva un número entero aleatorio que representa la figurita que nos tocó.

**Ejercicio 5.12: Cantidad de compras**

Implementá la función cuantas\_figus(figus\_total) que, dado el tamaño del álbum (figus\_total), genere un álbum nuevo, simule su llenado y devuelva la cantidad de figuritas que se debieron comprar para completarlo.

**Ejercicio 5.13:**

Ejecutá n\_repeticiones = 1000 veces la función anterior utilizando figus\_total = 6 y guardá en una lista los resultados obtenidos en cada repetición. Con los resultados obtenidos estimá cuántas figuritas hay que comprar, en promedio, para completar el álbum de seis figuritas.

*Ayuda: El comando np.mean(l) devuelve el promedio de la lista l.*

¿Podés crear esta lista usando una comprensión de listas?

**Ejercicio 5.14:**

Calculá n\_repeticiones=100 veces la función cuantas\_figus(figus\_total=670) y guardá los resultados obtenidos en cada repetición en una lista. Con los resultados obtenidos estimá cuántas figuritas hay que comprar, en promedio, para completar el álbum (de 670 figuritas).

Guardá todo lo que hiciste hasta aquí sobre figuritas en un archivo figuritas.py. Lo que sigue profundiza un poco más en el asunto.

**Ahora con paquetes**

Estos ejercicios te recomendamos que los pienses y discutas con un compañere o alguna de tus otras personalidades (si es que tenés):

1. ¿Cómo impacta en lo realizado tener paquetes con figuritas en lugar de figus sueltas?
2. ¿Cómo puede representarse un paquete?

**Ejercicios con paquetes**

**Ejercicio 5.15:**

Simulá la generación de un paquete con cinco figuritas, sabiendo que el álbum es de 670. Tené en cuenta que, como en la vida real, puede haber figuritas repetidas en un paquete.

**Ejercicio 5.16:**

Implementá una función comprar\_paquete(figus\_total, figus\_paquete) que, dado el tamaño del álbum (figus\_total) y la cantidad de figuritas por paquete (figus\_paquete), genere un paquete (lista) de figuritas al azar.

**Ejercicio 5.17:**

Implementá una función cuantos\_paquetes(figus\_total, figus\_paquete) que dado el tamaño del álbum y la cantidad de figus por paquete, genere un álbum nuevo, simule su llenado y devuelva cuántos paquetes se debieron comprar para completarlo.

**Ejercicio 5.18:**

Calculá n\_repeticiones = 100 veces la función cuantos\_paquetes, utilizando figus\_total = 670, figus\_paquete = 5. Guarda los resultados obtenidos en una lista y calculá su promedio. Si te da la compu, hacelo con 1000 repeticiones.

**Graficar el llenado del álbum**

El siguiente código usa las funciones que hiciste antes para graficar la curva de llenado de un álbum a medida que comprás paquetes de figuritas. Es un primer ejemplo de gráfico de líneas. En las próximas clases estudiaremos los detalles sobre gráficos de una manera sistemática. Por ahora solo un botón de muestra.

def calcular\_historia\_figus\_pegadas(figus\_total, figus\_paquete):

album = crear\_album(figus\_total)

historia\_figus\_pegadas = [0]

while album\_incompleto(album):

paquete = comprar\_paquete(figus\_total, figus\_paquete)

while paquete:

album[paquete.pop()] = 1

figus\_pegadas = (album>0).sum()

historia\_figus\_pegadas.append(figus\_pegadas)

return historia\_figus\_pegadas

figus\_total = 670

figus\_paquete = 5

plt.plot(calcular\_historia\_figus\_pegadas(figus\_total, figus\_paquete))

plt.xlabel("Cantidad de paquetes comprados.")

plt.ylabel("Cantidad de figuritas pegadas.")

plt.title("La curva de llenado se desacelera al final")

plt.show()

**Ejercicios un toque más estadísticos:**

Los siguientes ejercicios suponen algunos conceptos un poco más avanzados de estadística. Son optativos pero interesantes.

**Ejercicio 5.19:**

Utilizando lo implementado en el ítem anterior, **estimá** la probabilidad de completar el álbum con 850 paquetes o menos.

*Sugerencia:* No leas esto antes de hacer el ejercicio. Hacelo primero y luego miralo. En este ejercicio resulta más compacto usar n\_paquetes\_hasta\_llenar=np.array(lista) para convertir a vector la lista conteniendo cuántos paquetes compraste en cada experimento hasta llenar el álbum. Trabajar con vectores tiene ventajas. Por ejemplo probá la siguiente instrucción:

(n\_paquetes\_hasta\_llenar <= 850).sum()

**Ejercicio 5.20: Plotear el histograma**

Usá un código similar al del [Ejercicio 5.8](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/02_NumPy_Arrays.md#ejercicio-58-empezando-a-plotear) para hacer un histograma de la cantidad de paquetes que se compraron en cada experimento, ajustando la cantidad de *bins* para que el gráfico se vea lo mejor posible.

**Ejercicio 5.21:**

Utilizando lo implementado, **estimá** cuántos paquetes habría que comprar para tener una chance del 90% de completar el álbum.

**Ejercicio 5.22:**

Repetí suponiendo que no hay figuritas repetidas en un paquete. ¿Cuánto cambian las probabilidades?

**Ejercicio 5.23: Cooperar vs competir**

Por último, suponé que cinco amigues se juntan y deciden compartir la compra de figuritas y el llenado de sus cinco álbumes solidariamente. Calculá cuántos paquetes deberían comprar si deben completar todos. Hacé 100 repeticiones y compará el resultado con la compra individual que calculaste antes.

Acordate de guardar todo lo que hiciste sobre figuritas en un archivo figuritas.py.

**5.4 Gráficos del Arbolado porteño**

**Ploteando datos reales**

En esta sección retomamos el dataset del arbolado porteño (arbolado-en-espacios-verdes.csv) para hacer algunos gráficos que nos permitan visualizar los datos. Te damos una guía muy elemental sobre cómo hacer esto y un par de punteros a la documentación oficial. Ya esperamos que empieces a poder buscar por tu cuenta la info que falte.

Seguiremos trabajando en el archivo arboles.py. Nos basaremos en el trabajo hecho con comprensión de listas la clase pasada. Los siguientes tres ejercicios hacelos dentro de tres funciones diferentes, guardalas y entregá el archivo arboles.py con estos agregados.

**Ejercicio 5.24: Histograma de altos de Jacarandás**

Usando tu trabajo en el [Ejercicio 4.16](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/05_Arboles2_LC.md#ejercicio-416-lista-de-altos-de-jacarand%C3%A1), generá un histograma con las alturas de los Jacarandás en el dataset.

Tu código debería verse similar a este:

import os

import matplotlib.pyplot as plt

os.path.join('..', 'Data', 'arbolado-en-espacios-verdes.csv')

arboleda = leer\_arboles(nombre\_archivo)

altos = [comprensión de listas]

plt.hist(altos,bins=...)

*Observación:* Spyder tiene opciones para mostrar las figuras dentro de la misma ventana o en una ventana nueva (Tools -> Preferences -> IPython console -> Graphics -> Backend). Te recomendamos generarlas en una ventana nueva. Luego, con plt.clf() podés borrar la figura actual y con plt.figure() generás una nueva figura por si querés dejar varias abiertas a la vez.

**Ejercicio 5.25: Scatterplot (diámetro vs alto) de Jacarandás**

En este ejercicio introducimos un nuevo tipo de gráfico: *el gráfico de dispersión* o *scatterplot*. El mismo usa coordenadas cartesianas para mostrar los valores de dos variables para un conjunto de datos.

En este caso vamos a graficar un punto en el plano (x,y) por cada árbol en el dataset (o para cada arbol de cierta especie). El punto correspondiente a un árbol con diámetro *d* y altura *h* será ubicado en la posición *x=d* y *y=h*. Este tipo de gráfico permite visualizar relaciones o tendencias entre las variables y es muy útil en el análisis exploratorio de datos.

Usando como base tu trabajo del [Ejercicio 4.17](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/05_Arboles2_LC.md#ejercicio-417-lista-de-altos-y-di%C3%A1metros-de-jacarand%C3%A1), vas a generar un scatterplot para visualizar la relación entre diámetro y alto de los Jacarandás del dataset.

Si ya tenés una lista o un vector *d* con diámetros y otra *h* con altos, es sencillo hacer un primer scatterplot:

import matplotlib.pyplot as plt

plt.scatter(d,h)

Algunas recomendaciones:

1. Convertí la lista generada en un ndarray de numpy, de esa forma podés usar rebanadas para obtener un vector *d* con diámteros y otro *h* con alturas inmediatamente.
2. Mirá algún ejemplo como [este](https://matplotlib.org/3.3.1/gallery/shapes_and_collections/scatter.html#sphx-glr-gallery-shapes-and-collections-scatter-py) y tratá de entender cómo se usan los parámetros opcionales *s* (de size, tamaño) y *c* (de color) y *alpha* (de transparencia) de la función [matplotlib.pyplot.scatter](https://matplotlib.org/3.3.1/api/_as_gen/matplotlib.pyplot.scatter.html" \l "matplotlib.pyplot.scatter).
3. Usando el parámetro *alpha* hacé que el gráfico permita visualizar dónde hay mayor densidad de datos.

¿Ves alguna relación entre el diámetro y el alto de los Jacarndás? ¿Te parece que es una relación lineal o de otro tipo?

Agregale nombres a los ejes y a la figura usando los siguientes comandos:

plt.xlabel("diametro (cm)")

plt.ylabel("alto (m)")

plt.title("Relación diámetro-alto para Jacarandás")

**Ejercicio 5.26: Scatterplot para diferentes especies**

Ahora vamos a usar la función medidas\_de\_especies() definida en el [Ejercicio 4.18](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/05_Arboles2_LC.md#ejercicio-418-diccionario-con-medidas).

Comenzando con éste código, hacé tres gráficos como en el ejercicio anterior, uno por cada especie.

import os

import matplotlib.pyplot as plt

os.path.join('..', 'Data', 'arbolado-en-espacios-verdes.csv')

arboleda = leer\_arboles(nombre\_archivo)

especies = ['Eucalipto', 'Palo borracho rosado', 'Jacarandá']

medidas = medidas\_de\_especies(especies, arboleda)

¿Se mantienen las relaciones que viste en el ejercicio anterior para las tres especies? ¿Hay diferencias entre las especies? Para un mismo alto, ¿cuál tiene mayor diámetro (tipicamente)?

Para poder comparar diferentes especies resulta conveniente fijar los límites en los ejes *x* e *y* en las diferentes figuras usando las funciones xlim() e ylim(). A continuación un ejemplo:

plt.xlim(0,30)

plt.ylim(0,100)

Acordate siempre de ponerle título a las figuras y nombres y unidades a los ejes. Guardá los últimos tres ejercicios dentro de tres funciones diferentes en tu archivo arboles.py. Te pediremos que lo entregues en la próxima página.

*Extra:* ¿podés hacer un solo gráfico que muestre dos de estas tres especies en diferentes colores y resulte claro? ¿Y las tres especies?

**5.5 Cierre de la clase**

En esta clase trabajamos con la generación de números (pseudo)aleatorios, el uso de la biblioteca NumPy y algunos ejemplos de aplicación de estos conceptos. También aprendimos a hacer algunos gráficos elementales en Python.

Para cerrar esta clase te pedimos dos cosas:

* Que recopiles las soluciones de los siguientes ejercicios:

1. El archivo generala.py del [Ejercicio 5.2](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/01_Random.md#ejercicio-52-generala-no-necesariamente-servida).
2. El archivo termometro.py del [Ejercicio 5.5](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/01_Random.md#ejercicio-55-gaussiana) y el [Ejercicio 5.7](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/02_NumPy_Arrays.md#ejercicio-57-guardar-temperaturas).
3. El archivo plotear\_temperaturas.py del [Ejercicio 5.8](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/02_NumPy_Arrays.md#ejercicio-58-empezando-a-plotear).
4. El archivo figuritas.py abarcando lo hecho con figuritas (al menos) hasta el [Ejercicio 5.14](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/03_Figuritas.md#ejercicio-514).
5. El archivo arboles.py incluyendo al menos el [Ejercicio 5.25](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/04_Arboles3_plt.md#ejercicio-525-scatterplot-di%C3%A1metro-vs-alto-de-jacarand%C3%A1s).

* Que completes [este formulario](https://docs.google.com/forms/d/1HX--FgwcYV1PJ6-UhXhaSUbrFSLsR0tpUnf_A1YLyVE) usando como identificación tu dirección de mail. Al terminar vas a obtener un link para enviarnos tus ejercicios y podrás participar de la revisión de pares.

¡Gracias! Nos vemos en la próxima clase.