NumPy desde Cero

Versión Mejorada

Bienvenidos a la versión mejorada del curso de NumPy.
Más visual, más claro, más práctico. ¡Comencemos!

1	5	9	3	2
7	42	0	8	4
3	9	1	7	5

```
import numpy as np
data = np.array([[1, 5, 9, 3, 2],
                 [7, 42, 0, 8, 4],
                 [3, 9, 1, 7, 5]])
# El poder de NumPy en tus manos
```

Operaciones Análisis Machine Learning Estructuras Rendimiento

NumPy

Índice del Curso Mejorado

Fundamentos

Broadcasting Expandido

Operaciones

Estadística Expandido

Conclusión

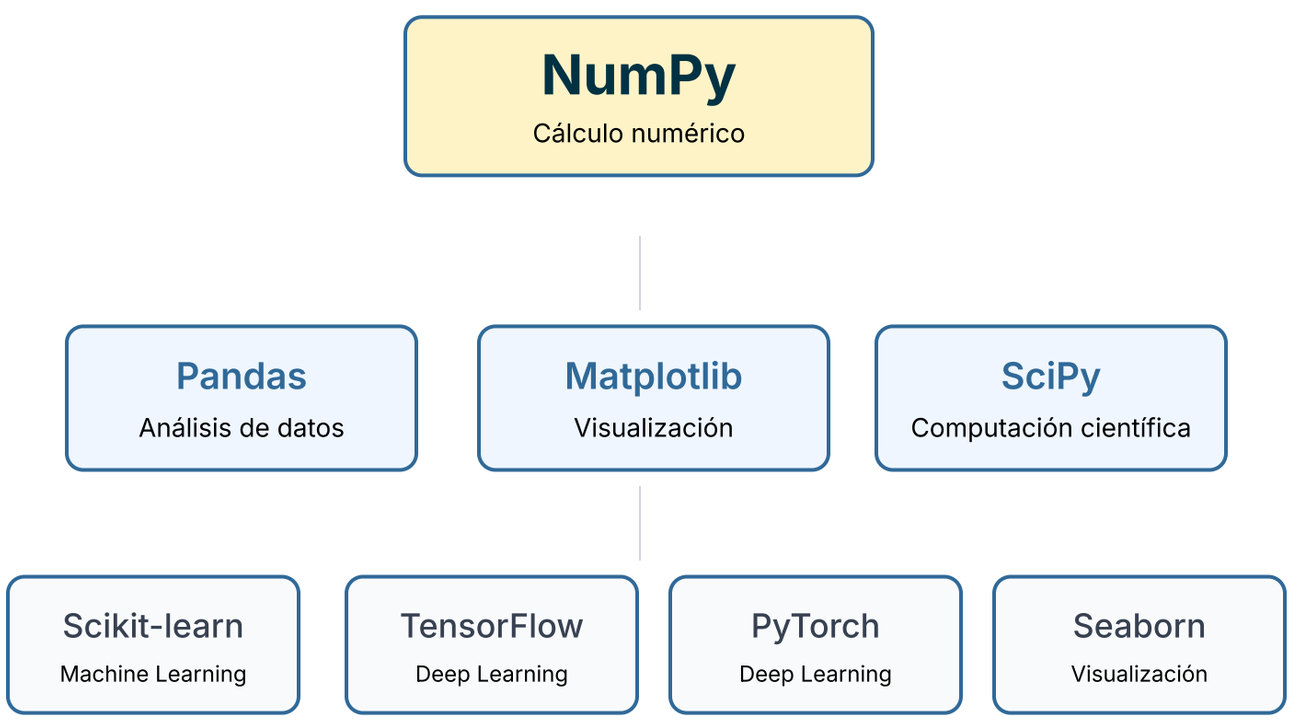
Se han **expandido** las secciones de Broadcasting y Estadística con más ejemplos prácticos y visuales.

¿Qué es NumPy?

La base fundamental del ecosistema científico de Python

NumPy (Numerical Python) es la librería central para computación numérica en Python. Proporciona estructuras de datos de alto rendimiento, funciones matemáticas y herramientas que permiten manipular grandes conjuntos de datos de manera eficiente.

Ecosistema Python para Data Science



¿Por qué NumPy es fundamental?

Rendimiento

Operaciones vectorizadas hasta 50x más rápidas que Python puro gracias a su implementación en C.

Arrays N-dimensionales

Estructura ndarray optimizada para cálculos numéricos con múltiples dimensiones.

Interoperabilidad

Compatible con todo el ecosistema científico de Python y con código C/C++/Fortran.

NumPy

Instalación paso a paso

Diferentes métodos según tu entorno de trabajo

Instalación con pip

Método estándar usando el gestor de paquetes de Python:

```
# Instalar NumPy
pip install numpy

# Instalar versión específica
pip install numpy==1.24.3
```

Verificar instalación:

```
>>> import numpy as np
>>> print(np.__version__)
1.24.3
```

Recomendado para:

- Entornos virtuales (venv)
- Instalaciones minimalistas
- Integración con requirements.txt

Instalación con Anaconda

Incluido por defecto en Anaconda Distribution. Si necesitas actualizarlo:

```
# Actualizar NumPy
conda update numpy

# Instalar versión específica
conda install numpy=1.24.3
```

1. **Crear un ambiente específico:**

```
conda create -n data_science numpy pandas
conda activate data_science
```

Recomendado para:

- Proyectos de ciencia de datos
- Entornos que requieran múltiples paquetes
- Facilidad en la gestión de dependencias

Consideraciones importantes

- Usa entornos virtuales para evitar conflictos
- Verifica compatibilidad con otras librerías
- En Google Colab, NumPy ya viene instalado
- NumPy requiere Python 3.8 o superior

NumPy

Arrays: ¿por qué no listas?

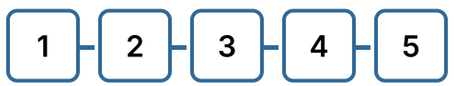
Comparación de estructuras y eficiencia

Estructura y Características

Listas Python	Arrays NumPy
Heterogéneas (elementos de distintos tipos)	Homogéneas (todos los elementos del mismo tipo)
Dinámicas (tamaño variable)	Tamaño fijo (más eficiente en memoria)
Optimizadas para operaciones secuenciales	Optimizadas para operaciones numéricas
Sin soporte nativo para operaciones matemáticas	Operaciones matemáticas vectorizadas

Representación en memoria

Lista Python: Referencias a objetos



Array NumPy: Bloque contiguo en memoria

Comparación de código

Lista Python:

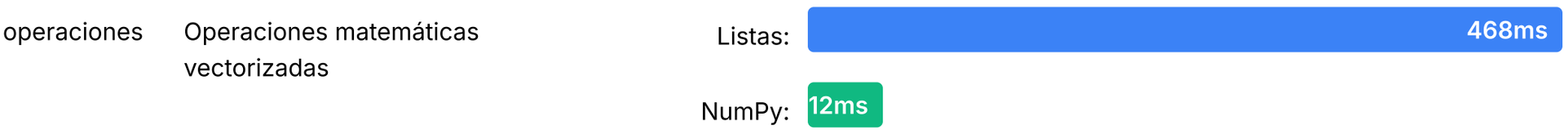
```
# Operación con lista
lista = [1, 2, 3, 4, 5]
resultado = []

for x in lista:
  resultado.append(x * 2)
```

Array NumPy:

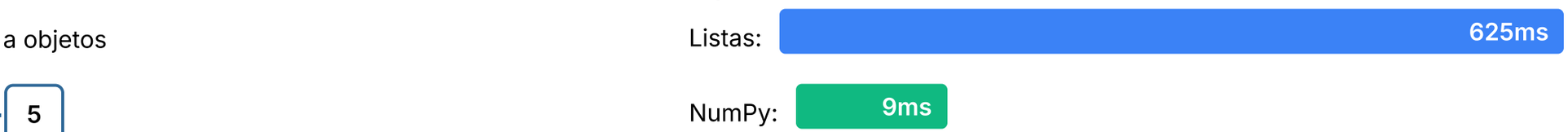
```
# Operación con array
import numpy as np
array = np.array([1, 2, 3, 4, 5])

resultado = array * 2
```

Eficiencia (1 millón de elementos)

Sumando elementos:



Multiplicando elementos:



Ventaja clave: Los arrays de NumPy pueden ser **hasta 100 veces más rápidos** que las listas de Python para operaciones numéricas, gracias a su implementación en C y las operaciones vectorizadas.

NumPy

Creando arrays con propósito

NumPy ofrece diferentes métodos para crear arrays según tus necesidades específicas

Desde listas (np.array)

```
import numpy as np

# Array 1D
arr1d = np.array([1, 2, 3, 4, 5])

# Array 2D (matriz)
arr2d = np.array([[1, 2, 3], [4, 5, 6]])
```

```
# Resultado 1D:
[1 2 3 4 5]

# Resultado 2D:
[[1 2 3]
 [4 5 6]]
```

Arrays predefinidos

```
# Array de ceros (3x3)
zeros = np.zeros((3, 3))

# Array de unos (2x2)
ones = np.ones((2, 2))

# Matriz identidad (3x3)
identity = np.eye(3)
```

```
# zeros(3,3):
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]

# eye(3):
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Secuencias numéricas

```
# Rango 0-9 (como range)
arr_range = np.arange(10)

# Rango con paso: inicio, fin, paso
```

Arrays aleatorios

```
# Valores aleatorios uniformes [0,1)
random = np.random.random((2, 3))

# Enteros aleatorios [low, high)
```

Consejos prácticos:

- Usa `dtype` para especificar el tipo de datos: `np.zeros((2,2), dtype=int)`
- Para casos de testing, `np.random.seed(42)` hace reproducibles los valores aleatorios

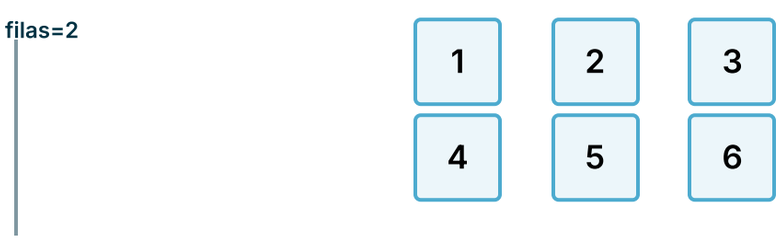
NumPy

Propiedades clave de los arrays

Atributos fundamentales para entender y manipular datos

```
# Crear un array de ejemplo
import numpy as np

matriz = np.array([[1, 2, 3], [4, 5, 6]])
```

Representación visual



Forma (shape)	Dimensiones (ndim)
(2, 3)	2
Tamaño (size)	**Tipo (dtype)**
6	int64

size=6 elementos

Propiedades importantes

shape **Forma del array**

Tupla con las dimensiones del array. Indica el número de elementos en cada eje.

```
>>> matriz.shape
(2, 3) # 2 filas, 3 columnas
```

dtype **Tipo de datos**

Tipo de datos homogéneo para todos los elementos del array.

```
>>> matriz.dtype
dtype('int64') # Enteros de 64 bits
```

size **Tamaño total**

Número total de elementos del array, independientemente de su forma.

```
>>> matriz.size
6 # Total de elementos (2×3)
```

ndim **Dimensiones**

Número de ejes (dimensiones) del array.

```
>>> matriz.ndim
2 # Array bidimensional (2D)
```

Consejo: Conocer estas propiedades es esencial para manipular correctamente los arrays. Por ejemplo, el **shape** te permite verificar si dos arrays son compatibles para operaciones matemáticas, mientras que **dtype** asegura que los cálculos sean precisos según el tipo de datos.

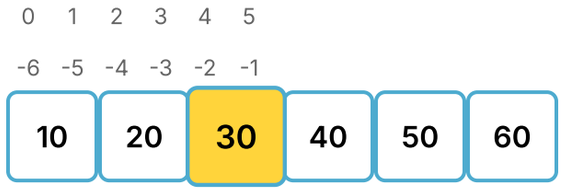
NumPy

Indexing y Slicing visual

NumPy permite acceder a elementos individuales o subconjuntos de arrays de forma flexible y eficiente, similar a las listas de Python pero con capacidades expandidas.

Array 1D: Acceso y Slicing

```
arr = np.array([10, 20, 30, 40, 50, 60])
```

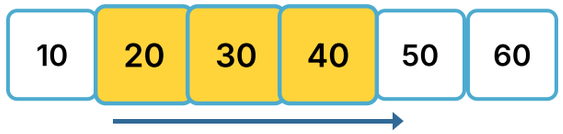


Acceso individual

```
arr[2] # 30
arr[-2] # 50
```

Slicing básico

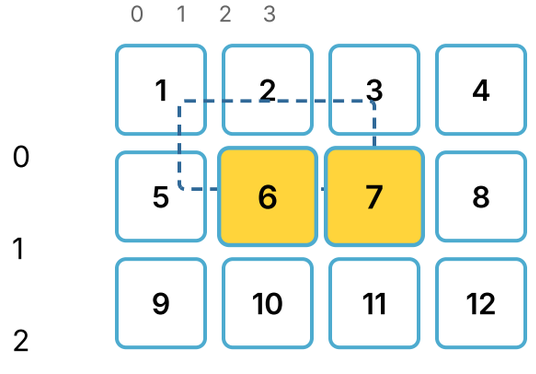
```
arr[1:4] # [20,30,40]
arr[::2] # [10,30,50]
```




```
arr[1:4] # [20,30,40]
```

Array 2D: Matrices

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



Elemento específico

```
matriz[1,2] # 7
matriz[0,0] # 1
```

Fila completa

```
matriz[1,:] # [5,6,7,8]
matriz[2] # [9,10,11,12]
```

Columna completa

```
matriz[:,1] # [2,6,10]
```

Submatriz

```
matriz[1:2,1:3] # [[6,7]]
```

Copias vs Vistas

Vista (slicing)

Los slices de arrays retornan **vistas**, no copias. Modificar la vista afecta al array original.

```
a = np.array([1,2,3])
b = a[0:2] # Vista
b[0] = 99 # Modifica también 'a'
# a es ahora [99,2,3]
```

Copia (copy)

Para crear una **copia** independiente, usa el método `.copy()`.

```
a = np.array([1,2,3])
b = a.copy() # Copia completa
b[0] = 99 # Solo modifica 'b'
# a sigue siendo [1,2,3]
```

Introducción a Broadcasting

¿Cómo opera NumPy con arrays de diferentes formas?

Broadcasting

Mecanismo que permite a NumPy realizar operaciones aritméticas en arrays de diferentes dimensiones **sin crear copias físicas** de los datos.

Sin Broadcasting (Python puro)

```
# Código Python sin broadcasting
A = [[1, 2, 3], [4, 5, 6]]
b = [10, 20, 30]

resultado = []
for i in range(len(A)):
  fila = []
  for j in range(len(A[0])):
    fila.append(A[i][j] + b[j])
  resultado.append(fila)
```

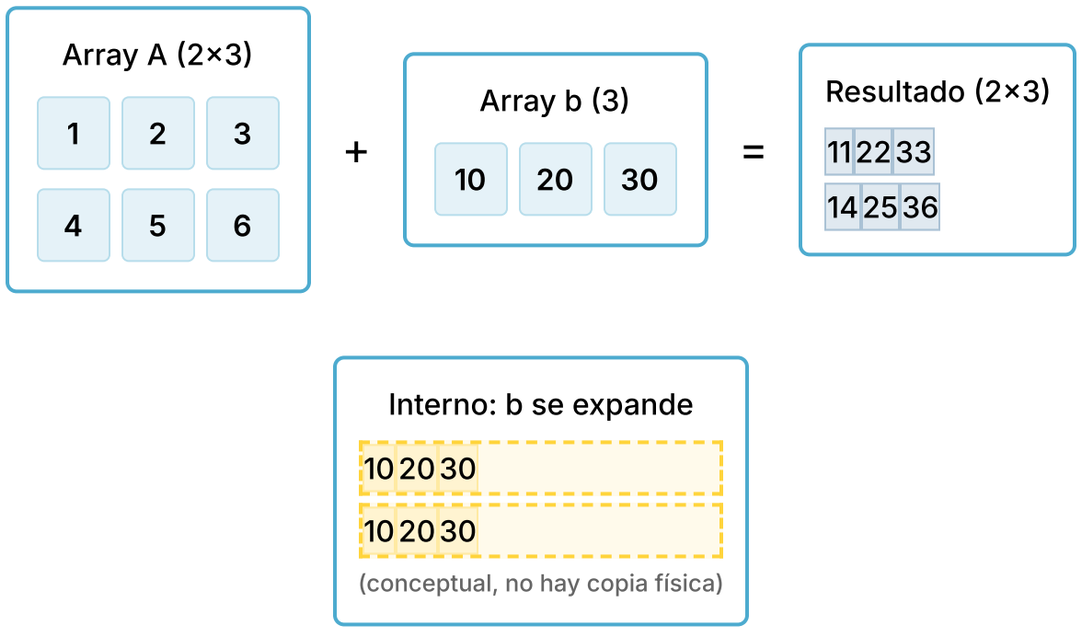
Código complejo, propenso a errores y de bajo rendimiento

Con Broadcasting (NumPy)

```
# Código NumPy con broadcasting
import numpy as np
A = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([10, 20, 30])

resultado = A + b # Broadcasting automático
```

Código limpio, compacto, eficiente y fácil de entender

Visualización del proceso



Reglas de Broadcasting

Regla 1: Comparación de derecha a izquierda

Las dimensiones se comparan comenzando por la última (derecha).

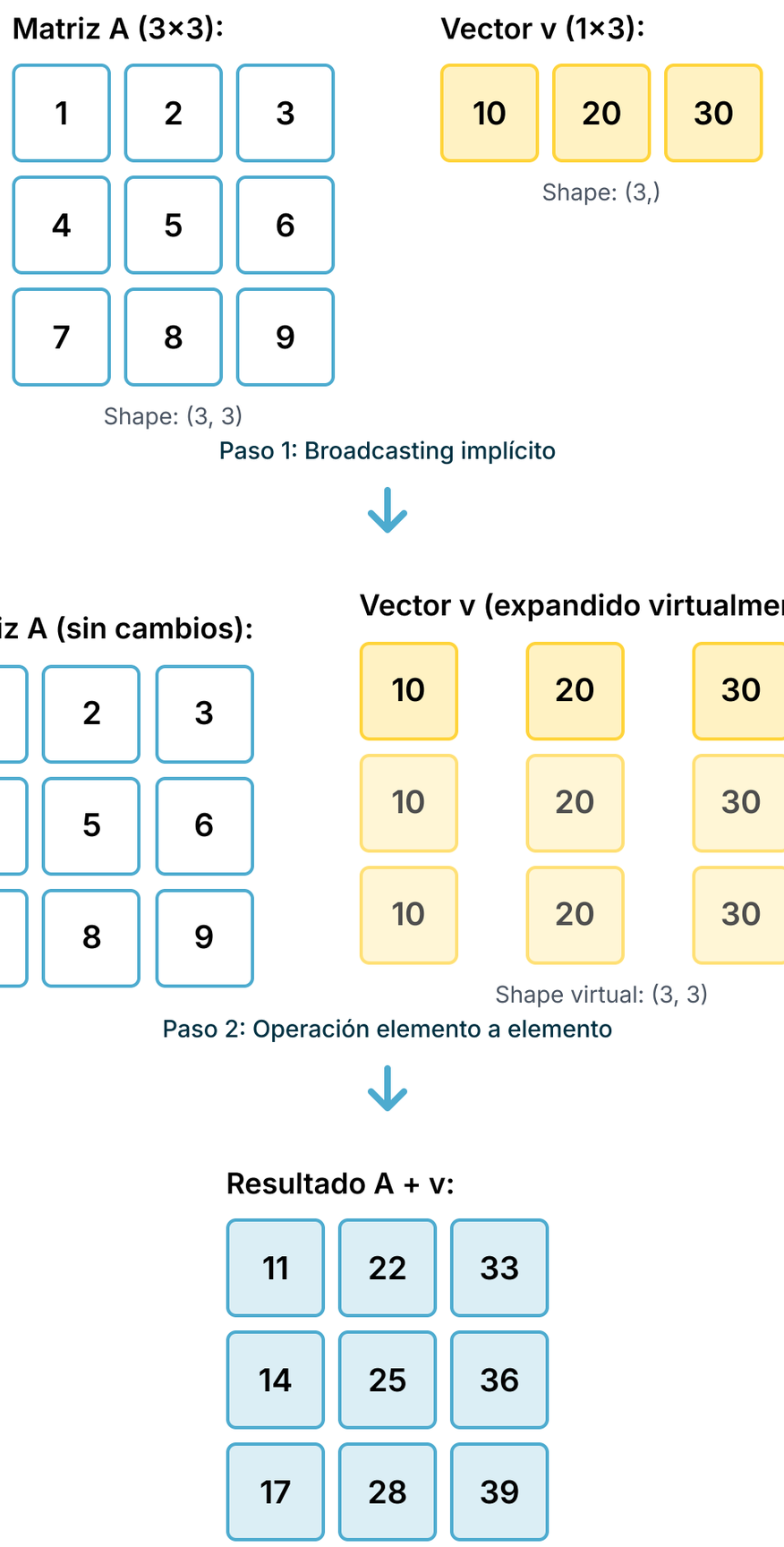
Regla 2: Compatibilidad dimensional

Dimensiones deben ser iguales o una debe ser 1 (o inexistente).

Broadcasting paso a paso

Visualización del proceso de expansión

Veamos un caso práctico: **suma entre una matriz 3×3 y un vector de longitud 3**. El broadcasting "expande" automáticamente el vector para hacerlo compatible con la matriz.




```
import numpy as np

# Definir arrays
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
v = np.array([10, 20, 30])

# Sumar usando broadcasting
resultado = A + v

# NumPy hace esto sin crear copias físicas en memoria
# El vector se "expande" virtualmente para coincidir con la matriz
```

Clave: NumPy no crea realmente una matriz completa para el broadcasting. La operación se realiza de forma eficiente, reutilizando los valores del vector para cada fila sin duplicar datos en memoria.

NumPy

Casos prácticos de Broadcasting

Sumando/multiplicando arrays de diferentes dimensiones

Broadcasting automático

```
# Creamos una matriz 3x3 y un vector
import numpy as np

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

vector = np.array([10, 20, 30])

# Sumamos matriz + vector
resultado = matriz + vector
```

Resultado (visualización fila por fila):

1	2	3	+	10	20	30	=	11	22	33
4	5	6	+	10	20	30	=	14	25	36
7	8	9	+	10	20	30	=	17	28	39

```
>>> resultado
array([[11, 22, 33],
       [14, 25, 36],
       [17, 28, 39]])
```

Método manual (sin broadcasting)

```
# Enfoque con bucles en Python puro
resultado_manual = []

for i in range(len(matriz)):
    fila_resultado = []
    for j in range(len(matriz[0])):
        suma = matriz[i][j] + vector[j]
        fila_resultado.append(suma)
    resultado_manual.append(fila_resultado)
```

¿Qué está pasando?

NumPy aplica **broadcasting** automáticamente, expandiendo el vector para que coincida con cada fila de la matriz.

Internamente, NumPy trata el vector `[10, 20, 30]` como si fuera una matriz donde cada fila es igual al vector, sin duplicar realmente los datos en memoria.

Ejemplo con multiplicación

```
# Multiplicación con broadcasting
resultado_mult = matriz * vector
```

```
>>> resultado_mult
array([[10, 40, 90],
       [40, 100, 180],
       [70, 160, 270]])
```

Ventajas del broadcasting:

- Código más limpio y legible
- Mucho más rápido (hasta 100x)
- Menos propenso a errores
- Optimizado para uso de memoria

Errores comunes y trucos de Broadcasting

El broadcasting es potente pero puede generar errores sutiles. Aprendamos a identificarlos y resolverlos.

Formas compatibles

(3,) (3, 1) y (3, 4)

(1, 3) y (4, 3)

(5, 1, 4) y (5, 3, 4)

Regla básica:

Las dimensiones se comparan de **derecha a izquierda** y deben ser:

- Iguales, o
- Una de ellas es 1, o
- Una de ellas no existe

Formas incompatibles

(3, 2) y (3, 4) ✗ 2 ≠ 4

(2, 3, 4) y (3, 3, 4) ✗ 2 ≠ 3

(3, 4, 5) y (4, 3, 2) ✗ Múltiples incompatibilidades

Error típico:

```
ValueError: operands could not be broadcast together with shapes
(3,2) (3,4)
```

Soluciones para problemas comunes

Problema 1: Dimensiones incorrectas

```python
import numpy as np
a = np.array([1, 2, 3]) # shape (3,)
b = np.array([[1, 2], [3, 4], [5, 6]]) # shape (3, 2)

# Error:
c = a + b # ¡Error de broadcasting!
```

```python
# Solución: reshape para cambiar dimensión
a_reshaped = a.reshape(3, 1)
# Ahora a_reshaped tiene shape (3, 1)

c = a_reshaped + b # Funciona correctamente
# Resultado: array con shape (3, 2)
```

Vector original: 1 2 3 — (3,)

Vector reshape: 1 / 2 / 3 — (3, 1)

Problema 2: Falta una dimensión

```python
import numpy as np
a = np.array([1, 2, 3, 4]) # shape (4,)
b = np.array([[1, 2, 3, 4], [5, 6, 7, 8]]) # shape (2, 4)

# Error si queremos sumar por columnas:
c = a + b # Suma por filas, no columnas
```

```python
# Solución: np.newaxis o None
a_col = a[:, np.newaxis] # o a[:, None]
# Ahora a_col tiene shape (4, 1)

c = a_col + b.T # Transponemos b (4, 2)
# Resultado: array con shape (4, 2)
```

Original: 1 2 3 4 — (4,)

Con newaxis: 1 / 2 / 3 / 4 — (4, 1)

Consejos prácticos:

- ✓ Usa `array.shape` para verificar dimensiones
- ✓ Usa `reshape` o `newaxis` estratégicamente
- ✓ Entiende el error: fíjate en las formas mencionadas
- ✓ Recuerda que `a[:, None]` = `a[:, np.newaxis]`

NumPy

Operaciones matemáticas y vectorizadas

Potencia y simplicidad del cálculo numérico

Operaciones aritméticas directas

```
import
numpy
as
np

# Crear un array de ejemplo
arr = np.array([1, 2, 3, 4, 5])

# Suma: elemento a elemento
arr + 10
# [11, 12, 13, 14, 15]

# Resta: elemento a elemento
arr - 1
# [0, 1, 2, 3, 4]

# Multiplicación: elemento a elemento
arr * 2
# [2, 4, 6, 8, 10]

# División: elemento a elemento
arr / 2
# [0.5, 1.0, 1.5, 2.0, 2.5]

# Potencias: elemento a elemento
arr ** 2
# [1, 4, 9, 16, 25]
```

Operaciones entre arrays

```
# Crear dos arrays
a = np.array([10, 20, 30])
b = np.array([1, 2, 3])

# Operaciones elemento a elemento
a + b
# [11, 22, 33]

a - b
# [9, 18, 27]

a * b
# [10, 40, 90]

a / b
# [10.0, 10.0, 10.0]
```

Python básico vs NumPy

Python básico

```
# Multiplicar por 2
lista = [1, 2, 3, 4, 5]
resultado = []

for
x
in
lista:
  resultado.append(x * 2)
```

NumPy vectorizado

```
# Multiplicar por 2
arr = np.array([1, 2, 3, 4,
5])

resultado = arr * 2
# [2, 4, 6, 8, 10]
```

Comparación de rendimiento

Tiempo para multiplicar un millón de elementos:

Python: 320ms

NumPy: 5ms

Funciones matemáticas

```
# Aplicar funciones a todo el array
arr = np.array([1, 4, 9, 16, 25])

np.sqrt(arr)
# [1.0, 2.0, 3.0, 4.0, 5.0]

np.log(arr)
# [0.0, 1.39..., 2.20..., 2.77..., 3.22...]

np.sin(arr)
# [0.84..., -0.76..., 0.41..., -0.29..., 0.13...]
```

¿Por qué NumPy es más rápido?
Las operaciones vectorizadas se ejecutan en C optimizado en lugar de Python, evitando bucles e iteraciones. Además, NumPy aprovecha instrucciones SIMD (Single Instruction, Multiple Data) a nivel de CPU.

NumPy

Funciones estadísticas: visión general

Herramientas esenciales para análisis descriptivo

NumPy proporciona un conjunto completo de funciones estadísticas que permiten analizar datos de forma rápida y eficiente. Estas funciones pueden aplicarse sobre arrays completos o especificando un eje (axis) para cálculos parciales.

np.sum()

`arr.sum() / np.sum(arr)`

Calcula la suma de todos los elementos del array.

Cuando necesitas totales o agregaciones

np.mean()

`arr.mean() / np.mean(arr)`

Calcula el promedio (media aritmética) de los elementos.

Para valores típicos o representativos

np.median()

`np.median(arr)`

Encuentra el valor central (50% de los datos).

Mejor con valores atípicos o distribuciones sesgadas

np.std()

`arr.std() / np.std(arr)`

Calcula la desviación estándar, que mide la dispersión de los datos.

Para analizar variabilidad de datos

np.var()

`arr.var() / np.var(arr)`

Calcula la varianza (std al cuadrado).

Para análisis estadísticos más avanzados

np.min()/max()

`arr.min() / arr.max()`

Encuentra los valores mínimos y máximos en el array.

Para encontrar rangos y valores extremos

Parámetros comunes importantes

- `axis` Eje sobre el que aplicar la función (0=columnas, 1=filas)
- `ddof` Grados de libertad para std/var (0=poblacional, 1=muestral)
- `keepdims` Mantener las dimensiones originales (True/False)
- `where` Máscara booleana para aplicar función solo a elementos específicos

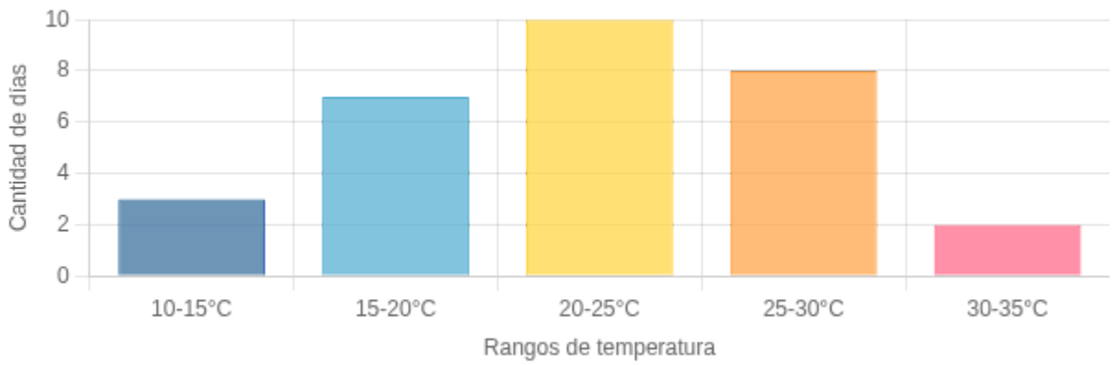
NumPy

Estadística descriptiva aplicada

Caso práctico: análisis de temperaturas diarias

```
# Simulación de temperaturas diarias (30 días)
import numpy as np
np.random.seed(42) # Para reproducibilidad

# Generar datos de temperatura (°C)
temperaturas = np.random.normal(22, 5, 30).round(1)

# Estadísticas descriptivas básicas
media = np.mean(temperaturas)
mediana = np.median(temperaturas)
desv_std = np.std(temperaturas)
minimo = np.min(temperaturas)
maximo = np.max(temperaturas)
rango = maximo - minimo

# Percentiles clave
p25 = np.percentile(temperaturas, 25)
p50 = np.percentile(temperaturas, 50) # Igual a mediana
p75 = np.percentile(temperaturas, 75)
iqr = p75 - p25 # Rango intercuartílico
```

Distribución de temperaturas



Resultados estadísticos

Temperatura media

22.1°C

Promedio de todas las mediciones diarias

Temperatura mediana

22.9°C

Valor central (P50)

Rango de temperaturas

22.2°C

Diferencia entre máx (33.3°C) y mín (11.1°C)

Desviación estándar

4.8°C

Dispersión respecto a la media

Interpretación de percentiles

P25: **19.2°C** P50: **22.9°C** P75: **25.2°C**

- El 25% de los días tuvo temperatura ≤ 19.2°C
- El 50% de los días tuvo temperatura ≤ 22.9°C
- El 75% de los días tuvo temperatura ≤ 25.2°C
- Rango intercuartílico (IQR): 6.0°C (dispersión central)

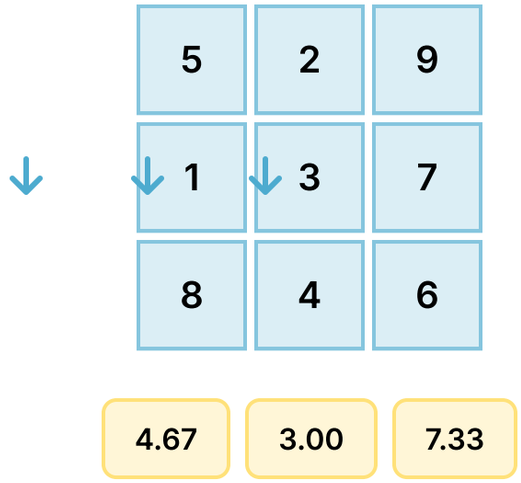
NumPy

Estadística sobre ejes y dimensiones

El poder del parámetro `axis`

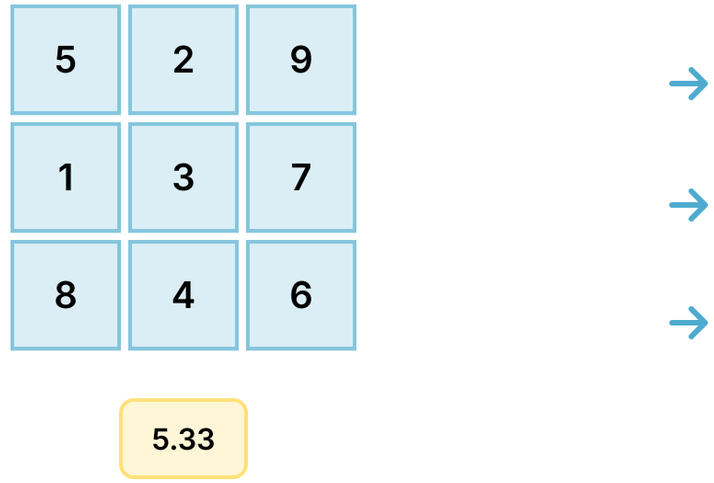
El parámetro `axis` permite aplicar funciones estadísticas a lo largo de diferentes dimensiones de un array, cambiando radicalmente los resultados obtenidos.

`axis=0` (Columnas)



Media por columna: `np.mean(arr, axis=0)`

`axis=1` (Filas)



Media por fila: `np.mean(arr, axis=1)`

```
import numpy as np

# Crear array 2D (matriz)
arr = np.array([[5, 2, 9],
                [1, 3, 7],
                [8, 4, 6]])

# Aplicar función estadística por columnas (axis=0)
col_means = np.mean(arr, axis=0)  # [4.67, 3.00, 7.33]

# Aplicar función estadística por filas (axis=1)
row_means = np.mean(arr, axis=1)  # [5.33, 3.67, 6.00]

# Aplicar sin especificar axis (media de todos los elementos)
overall_mean = np.mean(arr)       # 5.0
```

NumPy

Casos prácticos de análisis estadístico

Comparando calificaciones entre grupos de estudiantes

Análisis con NumPy

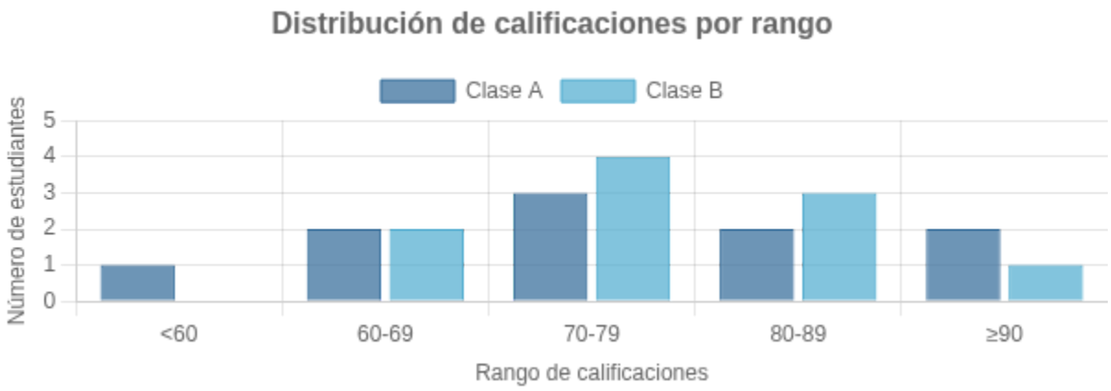
Comparemos las calificaciones de dos clases para identificar diferencias de rendimiento:

```
# Importamos NumPy
import
numpy as np

# Calificaciones de 0-100 para dos clases
clase_a = np.array([65, 78, 82, 90, 45, 88, 75, 95, 64,
70])
clase_b = np.array([72, 85, 68, 91, 79, 60, 77, 87, 69,
80])

# Estadísticas básicas para cada clase
print("Clase A - Media:", np.mean(clase_a))
print("Clase B - Media:", np.mean(clase_b))

print("Clase A - Mediana:", np.median(clase_a))
print("Clase B - Mediana:", np.median(clase_b))

print("Clase A - Desv. estándar:", np.std(clase_a))
print("Clase B - Desv. estándar:", np.std(clase_b))

# Notas máximas y mínimas
print("Clase A - Rango:", np.min(clase_a), "-",
np.max(clase_a))
print("Clase B - Rango:", np.min(clase_b), "-",
np.max(clase_b))

# ¿Cuántos estudiantes aprobaron? (nota ≥ 70)
aprobados_a = np.sum(clase_a >= 70)
aprobados_b = np.sum(clase_b >= 70)
```

Tip de análisis

NumPy permite comparar arrays completos con un solo operador, devolviendo un array booleano. Al aplicar `np.sum()` a este array, obtenemos el conteo de valores True.

Visualización y resultados



Clase A

Media:	75.2
Mediana:	76.5
Desv. Std:	14.7
Rango:	45 - 95
Aprobados:	7 de 10 (70%)

Clase B

Media:	76.8
Mediana:	78.0
Desv. Std:	9.7
Rango:	60 - 91
Aprobados:	8 de 10 (80%)

Interpretación

- La clase B tiene una media ligeramente superior (76.8 vs 75.2)
- La clase A muestra mayor variabilidad (σ=14.7 vs σ=9.7)
- La clase B tiene una tasa de aprobación mayor (80% vs 70%)
- La clase A tiene tanto la nota más baja como la más alta

NumPy y pandas: Para análisis más complejos, considera usar pandas que está construido sobre NumPy y ofrece funciones adicionales como `describe()` que genera automáticamente resúmenes estadísticos completos.

NumPy

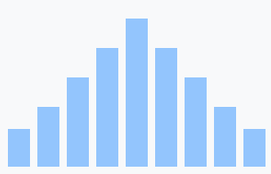
Visualizaciones estadísticas

Integrando NumPy con Matplotlib

NumPy y Matplotlib forman una combinación potente para el análisis visual de datos. Veamos cómo crear las visualizaciones estadísticas más comunes.

Histograma (distribución)

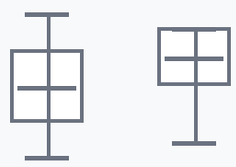
```
# Crear datos aleatorios normalmente distribuidos
import numpy as np
import matplotlib.pyplot as plt

data = np.random.normal(0, 1, 1000)

plt.figure(figsize=(8, 4))
plt.hist(data, bins=30, alpha=0.7, color='#4DABCF')
plt.title('Distribución normal')
plt.xlabel('Valor')
plt.ylabel('Frecuencia')
plt.show()
```



Vista previa: Histograma

Diagrama de caja y bigote

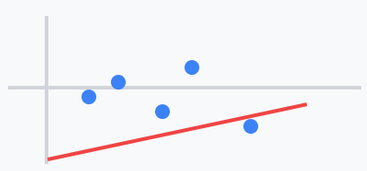
```
# Crear datos para tres grupos
grupo1 = np.random.normal(0, 1, 100)
grupo2 = np.random.normal(2, 0.5, 100)
grupo3 = np.random.normal(1, 1.5, 100)

data = [grupo1, grupo2, grupo3]

plt.figure(figsize=(8, 4))
plt.boxplot(data, labels=['Grupo A', 'Grupo B', 'Grupo C'])
plt.title('Comparación entre grupos')
plt.ylabel('Valores')
plt.grid(True, linestyle='--', alpha=0.7)
plt.show()
```



Vista previa: Box Plot

Gráfico de dispersión

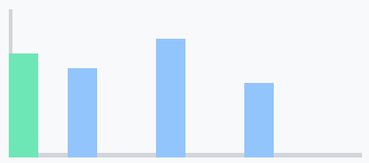
```
# Crear datos correlacionados
x = np.random.rand(50) * 10
y = x * 0.8 + np.random.normal(0, 1, 50)

# Calcular línea de regresión con polyfit
m, b = np.polyfit(x, y, 1)

plt.figure(figsize=(8, 4))
plt.scatter(x, y, alpha=0.7)
plt.plot(x, m*x + b, color='red')
plt.title('Correlación con línea de tendencia')
plt.xlabel('Variable X')
plt.ylabel('Variable Y')
plt.show()
```



Vista previa: Scatter Plot

Gráfico comparativo

```
# Datos de ejemplo
categorias = ['A', 'B', 'C', 'D', 'E']
valores1 = np.array([23, 45, 56, 78, 32])
valores2 = np.array([42, 31, 41, 65, 44])

# Calcular estadísticas
diferencia = valores1 - valores2
media1, media2 = np.mean(valores1), np.mean(valores2)

# Crear gráfico de barras
indice = np.arange(len(categorias))
ancho = 0.35

plt.figure(figsize=(10, 5))
plt.bar(indice, valores1, ancho, label='Grupo 1')
plt.bar(indice + ancho, valores2, ancho, label='Grupo 2')
plt.axhline(y=media1, color='r', linestyle='--', alpha=0.7)
plt.axhline(y=media2, color='b', linestyle='--', alpha=0.7)
plt.title('Comparación de grupos')
plt.xticks(indice + ancho/2, categorias)
plt.legend()
plt.show()
```



Vista previa: Bar Chart

Consejos para visualización

Usa `plt.tight_layout()` para evitar recortes y `plt.savefig('nombre.png')` para guardar el gráfico. Con `%matplotlib inline` muestras los gráficos directamente en Jupyter Notebooks.

Buenas prácticas y errores frecuentes

Optimiza tu código NumPy y evita problemas comunes

Buenas prácticas

Usar operaciones vectorizadas

```
# Bien: Operación vectorizada
resultado = array * 2 + 5
```

Aprovecha las operaciones vectorizadas en lugar de bucles para mejor rendimiento y código más limpio.

Usar fancy indexing

```
# Bien: Fancy indexing
indices = [0, 2, 4]
seleccionados = array[indices]
```

Selecciona múltiples elementos no consecutivos con un solo comando.

Aprovechar funciones integradas

```
# Bien: Uso de funciones NumPy
promedio = np.mean(array, axis=0)
ordenados = np.sort(array)
```

NumPy incluye cientos de funciones optimizadas. Búscalas antes de implementar tu propia solución.

Errores frecuentes

Errores de forma (shape)

```
# Error: Dimensiones incompatibles
a = np.array([1, 2, 3]) # (3,)
b = np.array([[1, 2], [3, 4]]) # (2, 2)
c = a + b # ¡Error!
```

Verifica siempre las dimensiones con `array.shape` antes de operar.

Copias vs. vistas

```
# Error: Modificar sin querer el original
b = a[1:3] # Esto es una vista
b[0] = 99 # Modifica 'a' también
```

Para crear una copia independiente, usa `array.copy()`.

Bucles innecesarios

```
# Mal: Bucle innecesario
for i in range(len(array)):
  array[i] = array[i] * 2
```

Evita bucles para operaciones elemento a elemento. NumPy está optimizado para operaciones vectorizadas.

Consejos generales

Optimiza memoria
Usa dtype adecuado según tus datos (int32 vs int64, float32 vs float64).

Inspecciona tus datos
Usa shape, dtype y size regularmente para verificar tus arrays.

Broadcasting con cuidado
Usa broadcasting para simplificar, pero verifica compatibilidad de dimensiones.

NumPy

Resumen general y preguntas

Puntos clave del curso

- NumPy es el pilar de la ciencia de datos en Python
- Operaciones vectorizadas son hasta 100x más rápidas
- Broadcasting simplifica operaciones entre arrays de diferentes dimensiones
- Funciones estadísticas optimizadas para análisis eficiente
- Mejor legibilidad de código = menos errores, más productividad

Próximos pasos

- Práctica: Ejercicios y mini-proyectos para consolidar
- Integración: Combinar NumPy con Pandas y Matplotlib
- Optimización: Mejorar rendimiento de código existente

Recursos recomendados

Documentación oficial

Referencia completa de todas las funcionalidades

numpy.org/doc

NumPy Tutorials

Guías paso a paso para casos prácticos

numpy.org/tutorials

Recursos curso

Ejercicios, notebooks y código fuente

github.com/curso-numpy

Comunidad

Foros y canales para resolver dudas

numpy-discussion.scipy.org

¿Preguntas?

Ahora es el momento de resolver cualquier duda sobre NumPy o cómo aplicarlo a tus proyectos específicos.

¡Gracias por participar en el curso!

Esperamos que NumPy potencie tus proyectos de Data Science