



Recibe una cálida:

¡Bienvenida!

Te estábamos esperando 😊 +

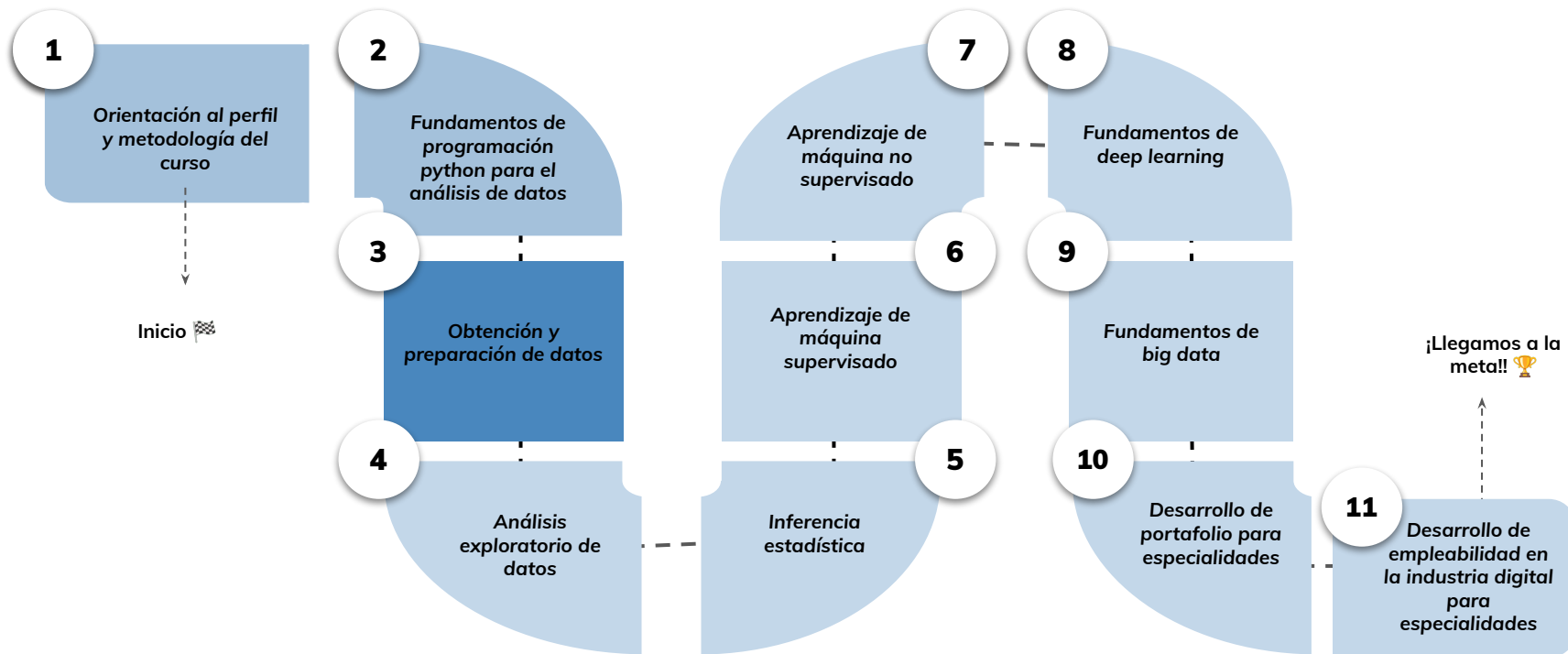


› La librería numpy

Aprendizaje Esperado: Manipular estructuras de datos vectoriales y matriciales utilizando biblioteca numpy para resolver un problema.

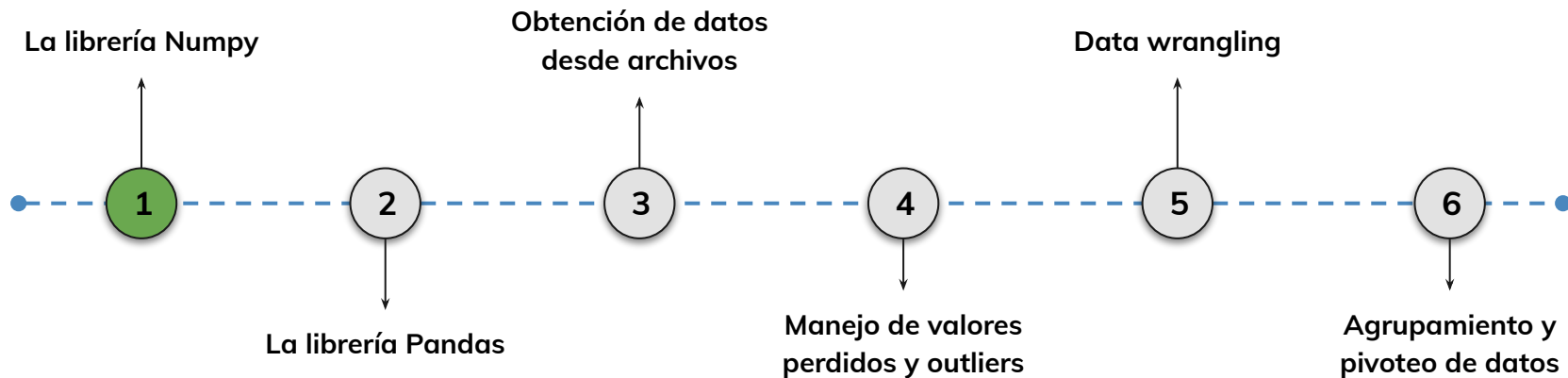
Hoja de ruta

¿Cuáles skills conforman el programa? **Fundamentos de Ciencia de Datos**



Roadmap de lecciones

¿Cuáles **lecciones** estaremos estudiando en este módulo?



Learning Path

¿Cuáles temas trabajaremos hoy?

1.

Manipulación de estructuras NumPy

Aprenderemos a crear, transformar y operar sobre vectores y matrices usando NumPy para resolver problemas de álgebra lineal, machine learning y análisis numérico.

Vectores en NumPy

Creación de vectores y operaciones básicas

Producto punto, uso en modelos y simulaciones

Matrices en NumPy

Creación de matrices, acceso y redimensionado

Operaciones entre matrices y aplicaciones

Learning Path

¿Cuáles temas trabajaremos hoy?

1.

Indexación, operaciones y álgebra lineal en NumPy

Aprenderemos a acceder, filtrar, transformar y operar con estructuras NumPy, abordando slicing, máscaras booleanas, funciones estadísticas, broadcasting y funciones de álgebra lineal.

Acceso y manipulación

Operaciones y álgebra

Indexación y slicing

Selección condicional y copias

Operaciones vectorizadas y broadcasting

Funciones estadísticas, ufuncs y np.linalg

Objetivos de aprendizaje

¿Qué aprenderás?

- Manipular vectores y matrices con NumPy en Python
- Realizar operaciones matemáticas vectorizadas
- Crear arreglos con funciones como arange, zeros, ones, eye y random
- Aplicar funciones como reshape, dot, y linspace en problemas reales
- Entender la aplicación de NumPy en áreas como machine learning, álgebra lineal y visualización de datos


Objetivos de aprendizaje

¿Qué aprenderás?

- Usar slicing e indexación para extraer y transformar estructuras
- Aplicar condiciones lógicas para filtrar datos en arreglos
- Distinguir entre referencias y copias
- Utilizar funciones matemáticas, estadísticas y de álgebra lineal
- Ejecutar operaciones avanzadas sin bucles, maximizando rendimiento

Repaso clase anterior

¿Quedó alguna duda?

En la clase anterior trabajamos :

- Profundizamos en herencia, encapsulamiento y polimorfismo en Python
- Implementamos clases con atributos privados y uso de `@property`
- Aplicamos métodos especiales como `__str__()` y `__add__()`
- Creamos clases abstractas y demostramos su utilidad como contrato
- Modelamos jerarquías de empleados utilizando herencia múltiple y rutinas reutilizables

La librería numpy

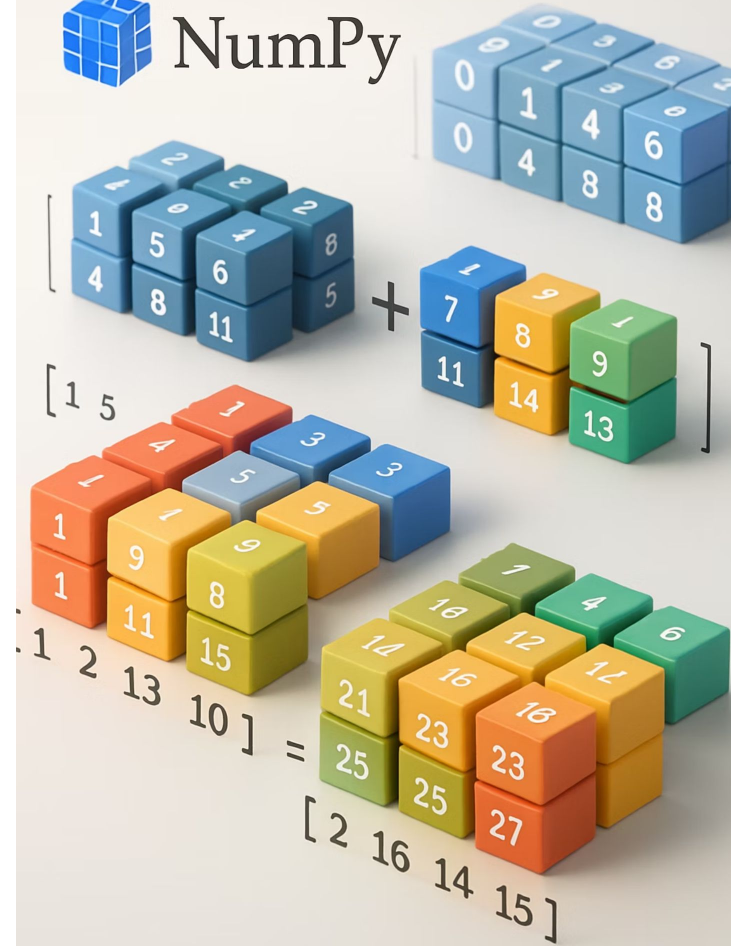


Librería Numpy

NumPy es una biblioteca esencial para el manejo eficiente de datos numéricos en Python. Su estructura optimizada y su capacidad para realizar operaciones matemáticas de manera eficiente la convierten en una herramienta indispensable en campos como la ciencia de datos, la inteligencia artificial y la computación científica.



NumPy



Reseña de la Librería NumPy

NumPy (Numerical Python) es una de las bibliotecas más utilizadas en Python para el cálculo numérico. Proporciona una estructura de datos eficiente para manejar grandes volúmenes de datos numéricos en forma de arreglos multidimensionales y matrices. Su uso es fundamental en campos como la inteligencia artificial, la ciencia de datos y la estadística.



Características principales de NumPy

Integración con otras bibliotecas

Una característica clave de NumPy es su integración con otras bibliotecas populares como Pandas, Matplotlib y Scikit-learn. Estas librerías dependen de NumPy para manejar datos de manera eficiente, lo que lo convierte en un pilar fundamental dentro del ecosistema de ciencia de datos en Python.

Operaciones vectorizadas

Otra ventaja de NumPy es su capacidad para manejar operaciones vectorizadas. Esto significa que se pueden realizar cálculos matemáticos sobre conjuntos de datos completos sin necesidad de escribir bucles, lo que mejora la legibilidad del código y la eficiencia computacional.



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

1	2	3
4	5	6

`a = np.zeros(4)`

Creación de arreglos de NumPy

NumPy ofrece diversas formas de crear arreglos, desde vectores unidimensionales hasta matrices multidimensionales, con diferentes tipos de datos y estructuras.

Vectores en NumPy

Los vectores en NumPy son arreglos unidimensionales que pueden almacenar múltiples valores numéricos. Se pueden definir utilizando la función `numpy.array()` a partir de listas de Python.

Un vector puede ser de cualquier tipo de dato numérico, como enteros, flotantes o booleanos. NumPy permite la conversión automática del tipo de datos para optimizar la memoria utilizada en la ejecución de cálculos.

Aplicaciones de Vectores

Álgebra Lineal

Los vectores son especialmente útiles en el álgebra lineal, donde representan coordenadas, velocidades y fuerzas en el espacio.

Modelos matemáticos

También se utilizan en modelos matemáticos y de machine learning.

Operaciones eficientes

Podemos realizar operaciones matemáticas directamente sobre vectores sin necesidad de escribir bucles.

Operaciones con vectores

También se pueden realizar operaciones entre dos vectores, como la suma, la resta y la multiplicación elemento a elemento.

```
import numpy as np
vector = np.array([1, 2, 3, 4, 5])
print(vector * 2) # Multiplica cada elemento por 2
```

Operaciones entre Vectores

```
vector1 = np.array([1, 2, 3])  
vector2 = np.array([4, 5, 6])  
suma = vector1 + vector2  
print(suma) # Output: [5 7 9]
```

El producto punto entre dos vectores es una operación común en machine learning y álgebra lineal, y se puede calcular fácilmente con `np.dot()`.

Matrices en NumPy

Definición

Las matrices en NumPy son arreglos bidimensionales que almacenan datos en filas y columnas. Se pueden definir como listas de listas dentro de `numpy.array()`.

Aplicaciones

Son esenciales en muchas áreas, como la estadística, el procesamiento de imágenes y la simulación de datos.

Eficiencia

Las matrices permiten la aplicación de múltiples operaciones matemáticas de manera eficiente y rápida.

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

Creación de Matrices

Podemos crear una matriz de la siguiente forma:

```
import numpy as np

# Crear una matriz 3x3
matrix = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])
```

Acceso a elementos de una matriz

Para acceder a un elemento dentro de la matriz, utilizamos índices.

```
valor = matrix[1, 2] # Elemento en la fila 1, columna 2 print(value) # Output: 6
```

Operaciones con Matrices

```
matriz_cuadrada = matriz ** 2  
print(matriz_cuadrada)
```

Es posible realizar operaciones matemáticas directamente sobre matrices sin necesidad de iteraciones.

Multiplicación de matrices

Las matrices también pueden multiplicarse entre sí utilizando la función `np.dot()` o el operador `@`.

```
matriz2 = np.array([[1, 0], [0, 1], [1, 1]])  
resultado = np.dot(matriz, matriz2)  
print(resultado)
```

Matrices en Machine Learning



Conjuntos de Datos

En machine learning, las matrices se utilizan para representar conjuntos de datos estructurados.



Imágenes

Las imágenes digitales se representan como matrices de píxeles con valores de intensidad.



Redes Neuronales

Las operaciones en redes neuronales se realizan mediante multiplicaciones de matrices.

NumPy Built-In Array Creation Functions

zeros



ones



arange



linspace



Funciones preconstruidas de creación

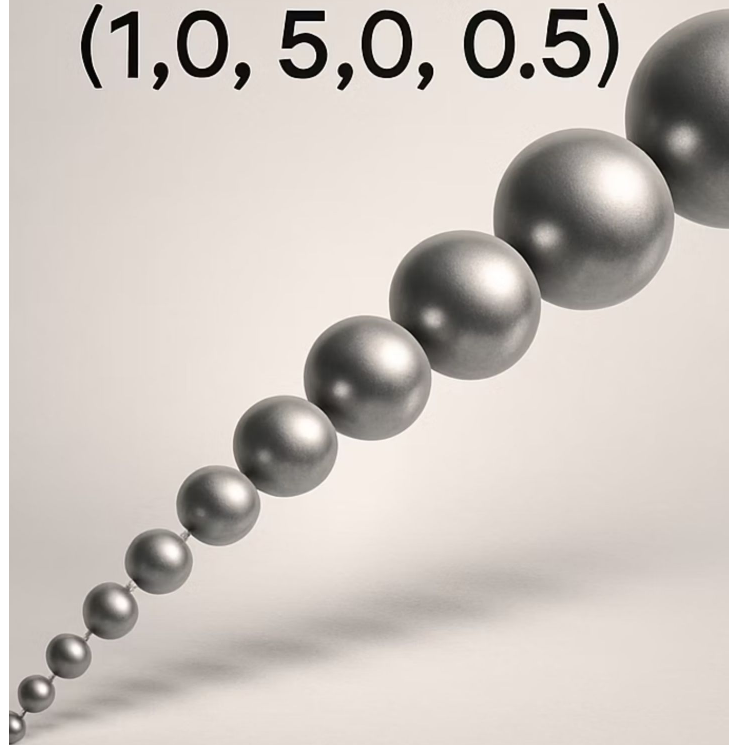
NumPy ofrece diversas funciones para crear arreglos con patrones específicos, facilitando la inicialización de estructuras de datos para diferentes aplicaciones.

Arreglo con valores (arange)

La función `np.arange()` genera secuencias de números con intervalos definidos. Es una alternativa eficiente a la función `range()` de Python.

```
secuencia = np.arange(0, 10, 2) # Values de 0 a 10  
print(secuencia)
```

np.arange (1,0, 5,0, 0.5)



Aplicaciones de np.arange()

Generación de datos

Esta función es especialmente útil en la generación de datos de prueba para algoritmos y modelos.

Simulaciones

Se utiliza en simulaciones matemáticas donde se necesitan secuencias de valores con intervalos regulares.

Visualización

Facilita la creación de ejes para gráficos y visualizaciones de datos.

Matrices de ceros y unos

NumPy permite la creación de matrices de ceros y unos mediante `np.zeros()` y `np.ones()`, respectivamente.

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

# Matriz de unos 2x4
ones_matrix = np.ones((2, 4))
```

Aplicaciones de matrices de ceros y unos

Inicialización

Estas matrices se utilizan frecuentemente en la inicialización de pesos en redes neuronales.

Algoritmos de optimización

Son útiles en algoritmos de optimización donde se necesita partir de valores conocidos.

Máscaras

Las matrices de ceros y unos pueden servir como máscaras para operaciones de filtrado en procesamiento de imágenes.

Vector con distribución de puntos

La función `linspace()` genera un vector con valores equidistantes en un rango dado.

```
# 5 puntos equidistantes entre 0 y 1
points = np.linspace(0, 1, 5)
# Output: [0.    0.25 0.5   0.75 1.   ]
```

Aplicaciones de linspace()

Gráficos

Este tipo de vector es utilizado en la generación de datos para gráficos con puntos equidistantes en los ejes.

Modelos matemáticos

Es útil en modelos matemáticos donde se necesita evaluar funciones en intervalos regulares.

Interpolación

Facilita la interpolación de valores entre puntos conocidos en análisis numérico.

Matriz identidad

La matriz identidad es un concepto clave en álgebra lineal y se puede crear con `np.eye()`.

```
# Matriz identidad 3x3
identity = np.eye(3)
# Output:
# [[1. 0. 0.]
#  [0. 1. 0.]
#  [0. 0. 1.]]
```

Aplicaciones de la matriz identidad

Transformaciones Lineales

La matriz identidad representa la transformación que no altera un vector, manteniendo su dirección y magnitud.

Sistemas de Ecuaciones

Es fundamental en la resolución de sistemas de ecuaciones lineales y en la inversión de matrices.

Regularización

Se utiliza en técnicas de regularización en machine learning para estabilizar modelos.

Matriz Aleatoria

Las matrices aleatorias se generan con `np.random.rand()` o `np.random.randint()`, dependiendo de si se necesitan valores flotantes o enteros.

```
matriz_aleatoria = np.random.rand(3, 3)
print(matriz_aleatoria)
```

Aplicaciones de matrices aleatorias

Inicialización de Modelos

Las matrices aleatorias son cruciales para inicializar pesos en redes neuronales y evitar la simetría.

Simulaciones

Se utilizan en simulaciones de Monte Carlo y otros métodos estocásticos.

Pruebas

Son útiles para generar datos de prueba aleatorios en el desarrollo de algoritmos.

Redimensionado de un arreglo

La función `reshape()` permite cambiar la estructura de un arreglo sin modificar sus valores.

```
# Vector de 12 elementos
arr = np.arange(12)
# Redimensionar a matriz 3x4
reshaped = arr.reshape(3, 4)
```

Aplicaciones del redimensionado

Preparación de datos

El redimensionado es esencial en la preparación de datos para algoritmos de machine learning.

Procesamiento de imágenes

Permite transformar imágenes entre diferentes representaciones (plana vs. matricial).

Reestructuración de datos

Facilita la reorganización de datos para análisis y visualización.

Live Coding

¿En qué consistirá la Demo?

Vamos a trabajar con funciones de creación avanzada de arreglos NumPy y realizar transformaciones estadísticas básicas sobre los datos simulados. Esto permitirá comprender cómo se puede simular un conjunto de datos y procesarlo para análisis exploratorio, sin necesidad de bucles ni librerías adicionales.

1. Importar NumPy
2. Crear un conjunto de datos aleatorios con `np.random.rand()`
3. Calcular la media por columna (promedio por característica)
4. Calcular la desviación estándar por columna
5. Estandarizar los datos (Z-score normalization)
6. Filtrar filas donde la primera característica sea mayor al promedio
7. Agregar una columna con el promedio por fila (nueva característica)
8. Obtener la matriz de correlación entre columnas

Tiempo: 20 minutos

#Momento de Preguntas...

3. Calcular la media por columna (promedio por característica)

Resumen rápido:

- `axis=0` : Operación por **columnas** (resultado tiene el mismo ancho que la matriz).
- `axis=1` : Operación por **filas** (resultado tiene la misma altura que la matriz).
- **Sin axis**: Promedio global de **todos** los números de la matriz.

5. Estandarizar los datos (Z-score normalization)

La estandarización o **Z-score** se usa para que todas tus variables estén en la misma escala.

Transforma tus datos de modo que tengan una **media de 0** y una **desviación estándar de 1**.

La fórmula que estás aplicando es:

$$z = \frac{x - \mu}{\sigma}$$

- x : El dato original.
- μ (`mean_col`): El promedio de la columna.
- σ (`std_col`): La desviación estándar de la columna.

#Momentode Preguntas...

6. Filtrar filas donde la primera característica sea mayor al promedio

1. La Selección: `data[:, 0]`

Aquí estamos extrayendo la información que queremos evaluar.

- El `:` significa "todas las filas".
- El `0` significa "la primera columna".

#Momento de Preguntas...

7. Agregar una columna con el promedio por fila (nueva característica)

1) `np.mean(data, axis=1)`

- `data` tiene forma **(10, 3)**: 10 filas, 3 columnas.
- `axis=1` significa: **promedia "a lo largo de las columnas"**, o sea, calcula el promedio de cada fila.

Resultado: un arreglo de forma **(10,)** (un vector) con 10 promedios, uno por fila.

Ejemplo mental:

- fila 0: promedio de sus 3 números
- fila 1: promedio de sus 3 números
- ...
- fila 9: promedio de sus 3 números

2) `.reshape(-1, 1)`

Esto cambia la forma del resultado para que deje de ser un vector **(10,)** y pase a ser una "columna"

(10, 1).

- El `-1` le dice a NumPy: **"calcula tú automáticamente cuántas filas deben ser"** según los datos y lo otro que le estoy imponiendo.
- El `1` le dice: **"quiero 1 columna"**.

Como hay 10 valores, NumPy concluye: "ok, entonces esto es (10 filas, 1 columna)".

#Momentode Preguntas...

8. Obtener la matriz de correlación entre columnas

la matriz de correlación es una forma ordenada de mirar cómo se relacionan varias variables entre sí, de a pares.

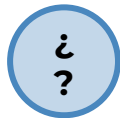
Cómo se interpreta (sin matemáticas duras)

- Valores **cercanos a 1**: cuando una variable aumenta, la otra suele aumentar también.
- Valores **cercanos a -1**: cuando una aumenta, la otra suele disminuir.
- Valores **cercanos a 0**: se mueven sin un patrón lineal claro.

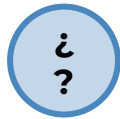
#Momentode Preguntas...



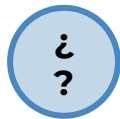
¿Para qué sirve la función reshape y en qué casos se usa?



¿Qué diferencia hay entre @ y np.dot()?



¿Qué ventajas tiene NumPy frente a listas tradicionales en Python?



¿Cómo afecta el uso de operaciones vectorizadas al rendimiento del código?



Ejercicio N° 1

Manipulando datos con NumPy sin bucles

Manipulando datos con NumPy sin bucles

Contexto: 🙌

Una empresa de análisis de datos necesita generar estructuras de prueba para un nuevo algoritmo de predicción. Para ello, debe crear arreglos numéricos representando variables simuladas y aplicar transformaciones sin utilizar bucles for, optimizando el rendimiento.

Consigna: 📝

Utilizando la librería NumPy, crea un vector con valores del 0 al 99. A partir de él, genera una matriz de 10x10. Luego:

- Calcula su transpuesta.
- Multiplica cada valor por 2 sin utilizar bucles.
- Obtén la diagonal principal.
- Crea una matriz identidad del mismo tamaño.
- Realiza una multiplicación matricial con @ entre la original y la identidad.

Paso a paso: ⚙️

- Importar la librería NumPy.
- Generar el vector con `np.arange(100)`.
- Transformarlo en matriz con `.reshape(10, 10)`.
- Realizar la transpuesta con `.T`.
- Multiplicar por 2 con operaciones vectorizadas.
- Extraer la diagonal con `np.diag()`.
- Crear la identidad con `np.eye(10)`.
- Multiplicar matrices con `@`.

Tiempo 🕒: 45 minutos



Momento:

Time-out!



5 -10 min.



La librería numpy

NumPy Indexing and Selection

array[2.3] →

array[1./3./2] →

1	2	3	4	4
5	6	7	8	8
5	6	9	11	12
13	14	15	15	16

Indexación y selección

NumPy ofrece métodos potentes para acceder y seleccionar elementos específicos dentro de arreglos, facilitando la manipulación de datos.

Selección de elementos de un arreglo

NumPy permite acceder a elementos individuales de un arreglo utilizando índices, similar a las listas en Python. En un vector unidimensional, el acceso se realiza indicando la posición del elemento.

```
import numpy as np
vector = np.array([10, 20, 30, 40, 50])
print(vector[2]) # Output: 30
```

Indexación en matrices

En arreglos bidimensionales (matrices), se deben especificar dos índices: el de la fila y el de la columna.

```
# Acceder al elemento en la fila 1, columna 2
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
element = matrix[1, 2] # Valor: 6
```

Slicing en NumPy

Definición

También se pueden seleccionar porciones del arreglo utilizando el operador : (slicing).

Sintaxis

La sintaxis es similar a la de las listas de Python:
`array[inicio:fin:paso].`

Dimensiones

En matrices, se puede hacer slicing en ambas dimensiones:
`matrix[1:3, 0:2].`

Ejemplos de Slicing

```
# Vector
```

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

```
slice1 = arr[2:7] # [2, 3, 4, 5, 6]
```

```
slice2 = arr[::2] # [0, 2, 4, 6, 8]
```

```
# Matriz
```

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

```
rows_1_2 = matrix[1:3] # Filas 1 y 2
```

```
cols_0_1 = matrix[:, 0:2] # Columnas 0 y 1
```

Selección condicional de elementos

NumPy permite seleccionar elementos de un arreglo que cumplan ciertas condiciones, generando arreglos booleanos.

```
numeros = np.array([3, 7, 2, 8, 5])  
mayores_a_5 = numeros[numeros > 5]  
print(mayores_a_5) # Output: [7 8]
```

Combinación de condiciones

También se pueden combinar múltiples condiciones usando operadores lógicos como & (AND) y | (OR).

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

# Elementos mayores que 2 Y menores que 7
condition = (arr > 2) & (arr < 7)
result = arr[condition] # [3, 4, 5, 6]
```

Aplicaciones de la selección condicional

Filtrado de Datos

La selección condicional es fundamental para filtrar datos según criterios específicos.

Limpieza de Datos

Permite identificar y tratar valores atípicos o faltantes en conjuntos de datos.

Segmentación

Facilita la segmentación de datos para análisis específicos o visualizaciones.

Referencia y copia de arreglos

Cuando se asigna un arreglo a una nueva variable en NumPy, no se crea una copia, sino una referencia al mismo objeto en memoria.

```
original = np.array([1, 2, 3, 4])
copia_ref = original # No es una copia real
copia_ref[0] = 99
print(original) # Output: [99  2  3  4]
```

Creación de copias

Para crear una copia real, se debe usar `copy()`.

```
# Referencia (no copia)
arr = np.array([1, 2, 3, 4])
ref = arr # Referencia al mismo arreglo

# Copia real
copy_arr = arr.copy() # Nueva copia independiente
```

Implicaciones de referencias vs. copias

Modificaciones

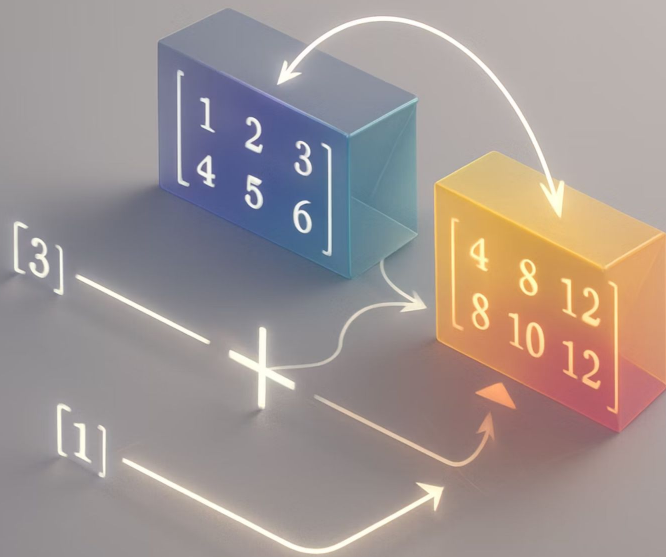
Los cambios en una referencia afectan al arreglo original, mientras que los cambios en una copia no.

Memoria

Las referencias son más eficientes en términos de memoria, pero pueden causar efectos secundarios no deseados.

Rendimiento

Crear copias consume más recursos pero garantiza la integridad de los datos originales.



Operaciones

NumPy proporciona una amplia gama de operaciones matemáticas optimizadas para trabajar con arreglos, permitiendo cálculos eficientes sin necesidad de bucles.

Operaciones entre arreglos

NumPy permite realizar operaciones aritméticas directamente entre arreglos del mismo tamaño.

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

# Operaciones elemento a elemento
suma = a + b # [5, 7, 9]
resta = a - b # [-3, -3, -3]
producto = a * b # [4, 10, 18]
division = a / b # [0.25, 0.4, 0.5]
```

Broadcasting

Definición

Cuando las dimensiones son compatibles, NumPy aplica broadcasting para operar arreglos de distinto tamaño sin usar bucles ni duplicar datos.

Reglas

Las dimensiones deben ser iguales o una de ellas debe ser 1 para que el broadcasting funcione.

Eficiencia

El broadcasting permite operaciones eficientes sin necesidad de crear copias físicas de los datos.

Ejemplo de Broadcasting

```
# Vector y escalar
arr = np.array([1, 2, 3, 4])
result = arr + 10 # [11, 12, 13, 14]

# Matriz y vector
matrix = np.array([[1, 2, 3], [4, 5, 6]])
vector = np.array([10, 20, 30])
result = matrix + vector
# [[11, 22, 33], [14, 25, 36]]
```

Operaciones con Escalares

Definición

Cuando se realizan operaciones con un escalar, esta se aplica a todos los elementos del arreglo.

Ejemplos

Multiplicación por escalar, suma de constante, elevación a potencia.

Eficiencia

Estas operaciones son altamente optimizadas en NumPy, mucho más rápidas que los bucles en Python.

Ejemplos de Operaciones con Escalares

```
arr = np.array([1, 2, 3, 4])

# Operaciones con escalares
suma = arr + 5 # [6, 7, 8, 9]
producto = arr * 2 # [2, 4, 6, 8]
potencia = arr ** 2 # [1, 4, 9, 16]
division = arr / 2 # [0.5, 1, 1.5, 2]
```

Aplicando funciones a un arreglo

NumPy proporciona funciones matemáticas para aplicar a cada elemento de un arreglo sin necesidad de bucles.

```
array = np.array([1, 4, 9, 16])
raiz_cuadrada = np.sqrt(array)
logaritmo = np.log(array)
seno = np.sin(array)

print(raiz_cuadrada) # Output: [1. 2. 3. 4.]
print(logaritmo) # [0. 1.38629436 2.19722458 2.77258872]
print(seno) # [0.84147098 -0.7568025 0.41211849 -0.28790332]
```

Funciones matemáticas en NumPy

Trigonométricas

`np.sin()`, `np.cos()`, `np.tan()`,
`np.arcsin()`, `np.arccos()`,
`np.arctan()`

Exponenciales y Logarítmicas

`np.exp()`, `np.log()`, `np.log10()`,
`np.sqrt()`

Redondeo

`np.round()`, `np.floor()`, `np.ceil()`

Ejemplos de funciones matemáticas

```
arr = np.array([0, np.pi/4, np.pi/2, np.pi])

# Aplicar funciones
senos = np.sin(arr) # [0, 0.7071, 1, 0]
logaritmos = np.log(np.array([1, 2, 3, 4]))
raices = np.sqrt(np.array([1, 4, 9, 16])) # [1, 2, 3, 4]
```

Funciones Estadísticas

NumPy ofrece un conjunto robusto de funciones estadísticas diseñadas para realizar análisis y resumir datos de manera eficiente. Permiten calcular métricas clave para entender la distribución y las propiedades de los arreglos numéricos.

Medidas de Tendencia Central

Incluyen funciones para calcular la media (`np.mean()`), la mediana (`np.median()`) y la moda de un arreglo, útiles para identificar el valor central de un conjunto de datos.

Medidas de Dispersión

Funciones como la desviación estándar (`np.std()`) y la varianza (`np.var()`) ayudan a entender la dispersión de los datos alrededor de la media, indicando qué tan lejos se extienden los valores.

Funciones de Agregación Básicas

Permiten obtener el valor mínimo (`np.min()`), el valor máximo (`np.max()`) y la suma total (`np.sum()`) de los elementos de un arreglo, facilitando un resumen rápido de los datos.

Ejemplos de Funciones Estadísticas

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

# Funciones estadísticas
media = np.mean(arr) # 3.0
mediana = np.median(arr) # 3.0
desviacion = np.std(arr) # ~1.41
minimo = np.min(arr) # 1
maximo = np.max(arr) # 5
suma = np.sum(arr) # 15
```

Funciones de Agregación

Por Eje

Las funciones de agregación pueden aplicarse a lo largo de un eje específico en matrices multidimensionales.

Ejemplos

`np.sum(matrix, axis=0)` suma columnas, `np.sum(matrix, axis=1)` suma filas.

Aplicaciones

Útiles para calcular estadísticas por grupos o categorías en análisis de datos.

Ejemplos de Agregación

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

```
# Suma por filas (axis=1)
```

```
row_sums = np.sum(matrix, axis=1) # [6, 15, 24]
```

```
# Suma por columnas (axis=0)
```

```
col_sums = np.sum(matrix, axis=0) # [12, 15, 18]
```

```
# Media por filas
```

```
row_means = np.mean(matrix, axis=1) # [2, 5, 8]
```

Funciones Universales (ufuncs)

Definición

Las ufuncs de NumPy son funciones matemáticas que se aplican elemento a elemento sobre arreglos, sin que tú tengas que escribir un for

Ventajas

Son mucho más rápidas que las funciones equivalentes en Python puro, ya que están implementadas en C.

Tipos

Existen ufuncs unarias (un solo input) y binarias (dos inputs).

Ejemplos de ufuncs

```
# Unarias
```

```
arr = np.array([-1, 0, 1, 2])
```

```
abs_values = np.abs(arr) # [1, 0, 1, 2]
```

```
squared = np.square(arr) # [1, 0, 1, 4]
```

```
# Binarias
```

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

```
b = np.array([4, 5, 6])
```

```
maximum = np.maximum(a, b) # [4, 5, 6]
```

```
power = np.power(a, b) # [1, 32, 729]
```

Álgebra Lineal con NumPy

Operaciones Matriciales

NumPy proporciona funciones para operaciones avanzadas de álgebra lineal como determinantes, inversas y descomposiciones.

Módulo linalg

El submódulo `np.linalg` contiene funciones especializadas para álgebra lineal.

Aplicaciones

Estas operaciones son fundamentales en machine learning, procesamiento de señales y optimización.

Ejemplos de Álgebra Lineal

```
from numpy import linalg as LA
```

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

```
# Determinante
```

```
det_A = LA.det(A) # -2.0
```

```
# Inversa
```

```
inv_A = LA.inv(A)
```

```
# Autovalores
```

```
eigenvalues = LA.eigvals(A)
```

```
# Norma
```

```
norm = LA.norm(A)
```

Operaciones Avanzadas

Transformada de Fourier

NumPy incluye funciones para calcular transformadas de Fourier, útiles en procesamiento de señales e imágenes.

Operaciones con Máscaras

Permite aplicar operaciones solo a elementos que cumplen ciertas condiciones mediante `np.where()` y `np.select()`.

Manipulación de Ejes

Funciones como `np.transpose()`, `np.swapaxes()` y `np.moveaxis()` permiten reorganizar las dimensiones de un arreglo.

Integración con Otras Bibliotecas



Pandas

NumPy se integra perfectamente con Pandas para análisis de datos estructurados y series temporales.



Matplotlib

Los arreglos de NumPy son la base para la visualización de datos con Matplotlib.



Scikit-learn

Los algoritmos de machine learning en Scikit-learn operan sobre arreglos NumPy.

Optimización de Rendimiento

Consejos para Mejorar el Rendimiento

- Evitar bucles en Python, usar operaciones vectorizadas
- Utilizar vistas en lugar de copias cuando sea posible
- Aprovechar las funciones de agregación con el parámetro axis
- Usar dtypes apropiados para optimizar memoria
- Considerar bibliotecas como Numba para código crítico

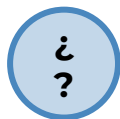
Live Coding

¿En qué consistirá la Demo?

En esta demo trabajaremos con datos simulados, usando NumPy para mostrar cómo se pueden limpiar, analizar y procesar matrices numéricas, de forma similar a lo que ocurre en un problema básico de machine learning.

1. Generar datos simulados en una matriz
Representan mediciones simples (por ejemplo, ciudades y días).
2. Aplicar reglas de limpieza con umbrales
Detectar valores altos y corregir valores bajos.
3. Calcular promedios básicos
Obtener resúmenes por fila y por columna.
4. Estimar un valor de salida (y_{hat})
Usar los promedios como una predicción sencilla.
5. Evaluar el resultado con un error simple (MSE)

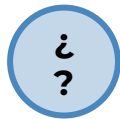
#Momentode Preguntas...



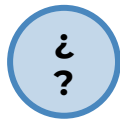
¿Qué diferencia hay entre `.copy()` y una simple asignación?



¿Qué hace `np.where()` y en qué se diferencia de `np.select()`?



¿Cómo mejora el rendimiento el uso de ufuncs respecto a bucles?



¿Qué ocurre si intento multiplicar dos matrices incompatibles en dimensiones?



Ejercicio N° 1

Análisis y transformación de una matriz con selección condicional

Análisis y transformación de una matriz con selección condicional

Contexto: 🙌

Trabajas con datos de temperaturas diarias de 5 ciudades durante una semana. Estos datos deben limpiarse, analizarse y procesarse para generar un reporte con estadísticas útiles para el equipo de analítica.

Consigna: ✍️

Utilizando NumPy, deberás:

- Simular una matriz 5x7 con temperaturas aleatorias entre 10°C y 40°C
- Identificar las temperaturas que superan los 30°C
- Reemplazar los valores inferiores a 15°C por el valor 15
- Calcular la media de temperaturas por ciudad (por fila) y por día (por columna)
- Determinar cuál es la ciudad con la mayor temperatura promedio

Tiempo 🕒: 45 min

Análisis y transformación de una matriz con selección condicional

Paso a paso: ⚙️

1. Crea una matriz de 5 filas y 7 columnas con valores enteros aleatorios entre 10 y 40
2. Selecciona las temperaturas mayores a 30°C utilizando operadores lógicos
3. Establece un umbral mínimo: los valores menores a 15 deben ser reemplazados por 15.
4. Calcula promedios por ciudad y por día
5. Compara los promedios por ciudad y encuentra cuál es la ciudad con el promedio más alto.

¿Alguna consulta?





Resumen

¿Qué logramos en esta clase?



- ✓ Entendimos qué es NumPy y por qué es fundamental
- ✓ Creamos vectores y matrices utilizando NumPy
- ✓ Aplicamos operaciones matemáticas sin bucles
- ✓ Usamos funciones como reshape, dot, eye, linspace, arange, etc.
- ✓ Exploramos aplicaciones reales en machine learning y álgebra lineal



Resumen

¿Qué logramos en esta clase?



- ✓ **Aprendimos a usar slicing e indexación avanzada**
- ✓ **Aplicamos selección condicional sobre arreglos**
- ✓ **Diferenciamos referencias y copias**
- ✓ **Usamos `np.mean`, `np.std`, `np.where`, `np.linalg`**
- ✓ **Realizamos álgebra lineal sin bucles y de forma optimizada**



¡Ponte a prueba!

Momento de ejercitación

Te invitamos a aprovechar esta última sección del espacio sincrónico para realizar de manera individual las **actividades disponibles en la plataforma**. Estas propuestas son clave para afianzar lo trabajado y **forman parte obligatoria del recorrido de aprendizaje**.

👉 **Análisis de caso** ————— 👉 **Selección Múltiple**

👉 **Comprensión lectora**

Si al resolverlas surge alguna duda, compartela o tráela al próximo encuentro sincrónico.

< **¡Muchas gracias!** >

