NUMPY: Numerical Computing with Python

El lenguaje Python es una excelente herramienta de programación de propósito general, con una sintaxis muy legible, tipos de datos ricos y potentes, y una filosofía conocida como el "Zen de Python". Sin embargo, no fue diseñado específicamente para la computación matemática y científica.

En particular, las listas de Python son contenedores muy flexibles, pero no son adecuadas para representar de manera eficiente construcciones matemáticas comunes como vectores y matrices. Esto puede llevar a problemas de rendimiento en aplicaciones que requieren cálculos numéricos intensivos.

Afortunadamente, existe el paquete (módulo) **numpy**, que es una biblioteca fundamental para la computación científica en Python. Proporciona estructuras de datos de alto rendimiento para vectores, matrices y estructuras de datos de dimensiones superiores en Python.

Numpy está implementado en lenguajes de bajo nivel como C y Fortran, lo que permite que los cálculos vectorizados (es decir, operaciones que se realizan en vectores y matrices) tengan un rendimiento excelente. Esta optimización es crucial para la eficiencia en cálculos numéricos, y es por eso que **Numpy** se utiliza en prácticamente todos los cálculos numéricos en Python.

¿Por qué no simplemente usar listas de Python para los cálculos en lugar de arrays?

Hay varias razones:

- Las listas de Python son muy generales. Pueden contener cualquier tipo de objeto y son de tipado dinámico. No soportan directamente operaciones matemáticas como multiplicaciones de matrices o productos punto. Implementar estas funciones para listas de Python no sería eficiente debido a su tipado dinámico, lo que puede llevar a errores en tiempo de ejecución.
- Los arrays de Numpy tienen tipado estático y son homogéneos. El tipo de los elementos se define al crear el array, lo que optimiza el uso de la memoria y mejora el rendimiento. Esto permite que Numpy realice operaciones de forma más rápida y eficiente.
- Numpy es eficiente en términos de uso de memoria. Los arrays son compactos y no almacenan más información de la necesaria, lo que resulta en un menor uso de memoria en comparación con listas de Python.
- Gracias al tipado estático y a su implementación en lenguajes compilados, **Numpy** permite realizar operaciones matemáticas como la multiplicación de matrices, suma y otras funciones matemáticas de manera mucho más rápida que con listas de Python.

Aplicaciones de Numpy:

- **Ciencia de datos**: Numpy es fundamental en el campo de la ciencia de datos, donde se utilizan grandes volúmenes de datos. Proporciona herramientas para la manipulación y análisis de datos, lo que facilita el trabajo con datasets complejos.
- **Aprendizaje automático**: Muchas bibliotecas de aprendizaje automático, como TensorFlow y Scikit-Learn, utilizan Numpy para operaciones matemáticas y manipulación de datos. Su eficiencia permite realizar cálculos rápidos y escalables en modelos de machine learning.
- **Simulación**: En campos como la física y la ingeniería, Numpy se utiliza para realizar simulaciones que requieren cálculos precisos y rápidos, desde simulaciones de sistemas físicos hasta el modelado de procesos estocásticos.

En resumen, **Numpy** es una herramienta poderosa y esencial para cualquier persona que trabaje con cálculos numéricos y matemáticos en Python, ofreciendo un rendimiento superior y una amplia gama de funcionalidades que van más allá de lo que las listas de Python pueden proporcionar.

¿Qué es NumPy?

NumPy, abreviatura de "Numerical Python", es una potente biblioteca para el lenguaje de programación Python. Está diseñada específicamente para facilitar la creación y manipulación de grandes matrices multidimensionales y arrays, junto con una extensa colección de funciones matemáticas de alto nivel para operar en estos arrays. NumPy es ampliamente utilizado en aplicaciones científicas, análisis de datos, aprendizaje automático y visualización de datos debido a sus eficientes capacidades de computación numérica.

Los orígenes de NumPy se remontan a la biblioteca "Numeric", desarrollada inicialmente por Jim Hugunin, con contribuciones de varios otros desarrolladores. Con el tiempo, NumPy evolucionó y amplió sus capacidades. En 2005, Travis Oliphant jugó un papel crucial en la fusión de características de una biblioteca competidora llamada Numarray en Numeric, resultando en el nacimiento de NumPy. Desde entonces, NumPy ha ganado tremenda popularidad y se ha convertido en una parte integral del ecosistema Python. Es un proyecto de código abierto con una vibrante comunidad de contribuyentes que continúa mejorando y ampliando sus funcionalidades.

En el corazón del paquete NumPy yace el objeto `ndarray` (abreviatura de "array n-dimensional"). Esta estructura de datos fundamental permite a los usuarios crear arrays con múltiples dimensiones, todos conteniendo tipos de datos homogéneos. Los arrays de NumPy son más eficientes que las listas de Python en términos de memoria y velocidad, lo que es crucial para manejar grandes volúmenes de datos.

Una de las principales fortalezas de NumPy es su capacidad para ejecutar muchas operaciones de manera eficiente mediante el uso de código compilado para mejorar el rendimiento. Esto permite que los usuarios realicen cálculos complejos en grandes conjuntos de datos sin comprometer la velocidad. NumPy soporta una amplia gama de operaciones, incluidas aritméticas, estadísticas, álgebra lineal y transformadas de Fourier.

Además, NumPy ofrece herramientas para integrar con otras bibliotecas populares como Pandas, Matplotlib y SciPy, lo que permite a los usuarios realizar análisis de datos avanzados y visualizaciones gráficas. La interoperabilidad entre estas bibliotecas hace que el ecosistema Python sea muy potente para la ciencia de datos y la investigación científica.

Existen varias diferencias notables entre los arrays de NumPy y las secuencias estándar de Python.

numpy.png

Por ejemplo:

- **Tipado homogéneo**: Los arrays de NumPy deben contener elementos del mismo tipo, lo que permite una mejor optimización en términos de memoria y velocidad de cálculo.
- **Eficiencia en memoria**: Los arrays de NumPy ocupan menos espacio en memoria en comparación con las listas de Python, ya que están diseñados para almacenar datos de manera compacta.
- **Operaciones vectorizadas**: NumPy permite realizar operaciones en arrays enteros de forma simultánea, en lugar de hacerlo elemento por elemento, lo que acelera significativamente los cálculos.
- **Funciones matemáticas avanzadas**: NumPy incluye una variedad de funciones matemáticas integradas, desde funciones estadísticas hasta transformadas, lo que facilita la realización de cálculos complejos.

En conclusión, **NumPy** no solo mejora la eficiencia en el manejo de datos y cálculos numéricos, sino que también ofrece una base sólida para el desarrollo de proyectos científicos y de análisis de datos. Con su creciente popularidad y una comunidad activa, NumPy sigue siendo una herramienta esencial en el arsenal de cualquier científico de datos, ingeniero o investigador.

Trabajar con datos numéricos

En el ámbito del análisis de datos, los "datos" suelen hacer referencia a información numérica, como mediciones científicas, precios de acciones, datos de sensores, o resultados de encuestas. Para manejar eficientemente estos cálculos numéricos en Python, la biblioteca NumPy ofrece estructuras de datos especializadas como los arrays y una gran cantidad de funciones matemáticas que facilitan los cálculos sobre grandes conjuntos de datos. Su capacidad para manipular arrays multidimensionales y realizar cálculos vectorizados lo convierte en una herramienta esencial para la ciencia de datos, la ingeniería y la investigación científica.

NumPy también es extremadamente eficiente en términos de rendimiento debido a que realiza operaciones en arrays de manera más rápida que las listas nativas de Python. Estas operaciones incluyen matemáticas elementales, álgebra lineal, estadística y transformaciones avanzadas, lo que lo hace perfecto para aplicaciones científicas y de big data.

A continuación, vamos a explorar un ejemplo que demuestra cómo NumPy facilita trabajar con datos numéricos en un contexto práctico. Consideremos un escenario donde queremos evaluar la idoneidad de varias regiones para el cultivo de manzanas utilizando datos climáticos históricos como la temperatura, la precipitación y la humedad relativa.

Caso práctico: Evaluación de regiones para el cultivo de manzanas

Imaginemos que estamos analizando si una región en particular es adecuada para el cultivo de manzanas basándonos en ciertos factores climáticos: temperatura media anual, precipitación anual y humedad relativa. Podemos modelar esta relación mediante una ecuación lineal que correlacione estas variables con el rendimiento del cultivo (medido en toneladas por hectárea).

La ecuación general sería:

`rendimiento_de_manzanas = w1 * temperatura + w2 * precipitación + w3 * humedad`

Aquí, los coeficientes `w1`, `w2` y `w3` representan los pesos que determinan la importancia relativa de cada factor climático en el rendimiento de las manzanas. Aunque la relación real podría ser más compleja y no estrictamente lineal, este modelo proporciona una estimación inicial útil.

Estimación de coeficientes

A partir de datos históricos y análisis estadístico, podemos determinar valores aproximados para los coeficientes `w1`, `w2` y `w3`, que podrían variar según la región y las condiciones específicas. Un análisis de regresión lineal podría proporcionarnos una buena estimación de estos valores, basados en datos históricos de producción agrícola.

Por ejemplo, supongamos que después de un análisis estadístico obtenemos los siguientes valores para los coeficientes:

dat_np.png

Predicción del rendimiento utilizando NumPy

Usando estos coeficientes estimados, ahora podemos predecir el rendimiento de las manzanas en diferentes regiones basándonos en los datos climáticos de cada área. Para hacer el cálculo más sencillo y eficiente, podemos usar arrays de NumPy. Los datos climáticos de cada región se pueden representar como un vector que contiene los valores de temperatura, precipitación y humedad.

Asignemos los coeficientes:

```
w1, w2, w3 = 0.3, 0.2, 0.5
```

Entonces, podemos predecir el rendimiento del cultivo en una región específica, como Kanto, usando los siguientes datos climáticos (temperatura = 73°F, precipitación = 67 mm, humedad = 43%):

```
yield_of_apples = w1 * 73 + w2 * 67 + w3 * 43
```

Ventajas de usar NumPy para cálculos numéricos

Al realizar estos cálculos utilizando NumPy, podemos aprovechar el poder de la computación vectorizada, lo que permite realizar operaciones sobre arrays completos de datos sin necesidad de escribir bucles explícitos. Esto no solo hace el código más conciso y legible, sino que también mejora el rendimiento, especialmente cuando se trabaja con grandes conjuntos de datos o múltiples regiones.

A continuación, veamos cómo podríamos implementar este cálculo usando arrays de NumPy:

Definimos los pesos como un array de NumPy

```
import numpy

weights = numpy.array([w1, w2, w3])
weights
```

```
array([0.3, 0.2, 0.5])
```

Ahora, los datos climáticos de Kanto también se pueden representar como un array

```
kanto = numpy.array([73, 67, 43])
```

Usamos el producto punto para calcular el rendimiento

```
yield_kanto = numpy.dot(weights, kanto)
print(yield_kanto)
```

```
56.8
```

Cálculo para múltiples regiones

Uno de los principales beneficios de NumPy es que facilita realizar cálculos para múltiples regiones de manera simultánea. Por ejemplo, si tuviéramos los datos climáticos de varias regiones, podríamos almacenarlos en un array bidimensional y usar NumPy para calcular los rendimientos de todas las regiones en un solo paso:

```
# Definimos los datos climáticos para varias regiones
regiones = numpy.array([
    [73, 67, 43],  # Kanto
    [91, 88, 64],  # Johto
    [87, 85, 73],  # Hoenn
    [102, 43, 38],  # Sinnoh
])

# Calculamos el rendimiento de manzanas para todas las regiones
rendimientos = numpy.dot(regiones, weights)
print(rendimientos)
```

```
[56.8 76.9 79.6 58.2]
```

En este ejemplo, NumPy realiza el cálculo del producto punto para cada región de forma eficiente. Esta capacidad de manejar cálculos en paralelo para conjuntos de datos grandes es una de las razones por las que NumPy es tan valioso para científicos de datos y profesionales que trabajan con grandes volúmenes de datos.

En conclusión, NumPy no solo facilita cálculos numéricos complejos, sino que también permite que el análisis de grandes conjuntos de datos sea más rápido y eficiente. Para cualquier proyecto que involucre análisis numéricos, NumPy es una herramienta indispensable.

Pasar de listas de Python a arrays de NumPy

Trabajar con listas de Python es adecuado para pequeñas colecciones de datos, pero cuando tratamos con grandes volúmenes de datos o cuando queremos realizar operaciones matemáticas y algebraicas eficientes, las listas pueden resultar limitadas. Aquí es donde **NumPy** entra en juego, proporcionando arrays n-dimensionales (`ndarray`) que son más rápidos y ocupan menos espacio en memoria. Además, los arrays de NumPy permiten realizar operaciones matemáticas avanzadas con una sintaxis clara y expresiva.

¿Por qué convertir listas de Python en arrays de NumPy?

Las listas de Python son estructuras de datos muy versátiles, capaces de almacenar diferentes tipos de objetos. Sin embargo, estas no están optimizadas para realizar cálculos numéricos. Algunas de las razones por las que podríamos querer convertir listas a arrays de NumPy incluyen:

1. **Mayor eficiencia de memoria**: Los arrays de NumPy son más compactos porque almacenan datos de manera continua en memoria, lo que optimiza el acceso a ellos.
2. **Velocidad**: Las operaciones con arrays de NumPy son mucho más rápidas porque aprovechan la optimización de código en lenguajes de bajo nivel como C y Fortran.

3. **Operaciones vectorizadas**: NumPy permite realizar operaciones matemáticas sobre arrays completos sin necesidad de escribir bucles explícitos, lo que simplifica el código y lo hace más legible.
4. **Funciones especializadas**: NumPy ofrece una amplia gama de funciones matemáticas y de álgebra lineal, como el producto punto, transformadas de Fourier, operaciones estadísticas, y mucho más.

Instalar y configurar NumPy

Para empezar a trabajar con NumPy, necesitas instalar la biblioteca si aún no lo has hecho. Puedes hacerlo fácilmente con el gestor de paquetes `pip3`:

```
pip3 install numpy
```

Una vez instalado, podemos importar la biblioteca en nuestro código. Es común utilizar el alias `np` para importar NumPy, lo cual es una convención en la comunidad de Python:

```
import numpy as np
```

Conversión de listas a arrays

A continuación, veremos cómo convertir una lista de Python en un array de NumPy. Empecemos con una lista de Python que representa, por ejemplo, datos climáticos de una región:

```
kanto_as_list = [73, 67, 43]  # Temperatura, precipitación y humedad
```

Verificamos el tipo de la lista:

```
type(kanto_as_list)

list
```

Para convertir esta lista en un array de NumPy, usamos la función `np.array()`:

```
kanto_array = np.array(kanto_as_list)
kanto_array

array([73, 67, 43])
```

Ahora, si verificamos el tipo del objeto `kanto_array`, observaremos que es un `ndarray` de NumPy:

```
type(kanto_array)

numpy.ndarray
```

Operaciones matemáticas con arrays de NumPy

Una vez que tenemos nuestros datos en formato de array, podemos aprovechar las capacidades de NumPy para realizar operaciones matemáticas eficientes. Por ejemplo, podemos multiplicar arrays elemento a elemento, realizar sumas, productos punto, y muchas otras operaciones sin necesidad de bucles.

Supongamos que tenemos un array que representa los pesos asociados a cada factor climático:

```
weights_array = np.array([0.3, 0.2, 0.5])  # Pesos para temperatura, precipitación y humedad
weights_array
```

```
array([0.3, 0.2, 0.5])
```

Podemos multiplicar los dos arrays de manera eficiente:

```
kanto_array * weights_array
```

```
array([21.9, 13.4, 21.5])
```

Esta operación multiplica elemento a elemento los dos arrays. Si ambos arrays tienen el mismo tamaño, el resultado será un nuevo array que contiene los productos individuales de cada par de elementos.

Suma de elementos

Si deseamos sumar los productos, simplemente usamos el método `sum()` de NumPy:

```
np.sum(kanto_array * weights_array)
```

```
56.8
```

Esta operación calcula la suma de todos los elementos del array resultante. La ventaja de NumPy es que estas operaciones están optimizadas y pueden manejar grandes volúmenes de datos con mucha mayor rapidez que las listas de Python.

Producto punto con NumPy

Para realizar un cálculo más formal del producto punto entre dos vectores (una operación muy común en álgebra lineal), NumPy proporciona la función `np.dot()`. Esta función realiza el producto escalar entre dos arrays:

```
np.dot(weights_array, kanto_array)
```

```
56.8
```

Esta es la forma más eficiente y directa de calcular el producto punto en NumPy, y es particularmente útil cuando trabajamos con grandes matrices o vectores en problemas de ciencia de datos o aprendizaje automático.

Otras operaciones matemáticas avanzadas con NumPy

NumPy no solo facilita las operaciones de producto punto, sino que también incluye una amplia gama de otras funciones matemáticas avanzadas que pueden ser útiles en diversos dominios, como:

1. **Álgebra lineal**: Operaciones como multiplicación de matrices (`np.matmul`), descomposiciones matriciales (SVD, LU, etc.), y resolución de sistemas de ecuaciones lineales.
2. **Transformadas de Fourier**: Para aplicaciones en procesamiento de señales, la función `np.fft` permite aplicar transformadas rápidas de Fourier.
3. **Estadísticas**: NumPy tiene funciones para cálculos estadísticos como media (`np.mean`), desviación estándar (`np.std`), mediana (`np.median`), y muchas otras.
4. **Generación de números aleatorios**: NumPy incluye un módulo para generar números aleatorios (`np.random`) que es muy utilizado en simulaciones, análisis estocásticos y en la creación de datasets de prueba.

Diferencias de rendimiento entre listas y arrays

Para ilustrar la diferencia de rendimiento entre listas de Python y arrays de NumPy, considera la siguiente comparación en la ejecución de una operación simple como la suma de grandes cantidades de elementos. NumPy suele ser considerablemente más rápido, ya que las operaciones se realizan a nivel de bajo nivel (en C) y están optimizadas.

```
import time

# Usando listas de Python
size = 1000000
lista1 = list(range(size))
lista2 = list(range(size))

start_time = time.time()
resultado_lista = [x + y for x, y in zip(lista1, lista2)]
print(f"Tiempo con listas: {time.time() - start_time} segundos")

# Usando arrays de NumPy
array1 = np.arange(size)
array2 = np.arange(size)

start_time = time.time()
resultado_array = array1 + array2
print(f"Tiempo con arrays: {time.time() - start_time} segundos")

Tiempo con listas: 0.10295629501342773 segundos
Tiempo con arrays: 0.0059680938720703125 segundos
```

Esta comparación demuestra cómo NumPy ofrece una optimización significativa al manejar cálculos numéricos a gran escala.

En resumen, convertir listas de Python en arrays de NumPy no solo mejora la eficiencia y velocidad de las operaciones matemáticas, sino que también permite acceder a una amplia gama de herramientas matemáticas avanzadas. NumPy es esencial para la ciencia de datos, la investigación científica, la ingeniería y cualquier campo que implique cálculos numéricos intensivos.

Ventajas de usar Arrays de NumPy

Los arrays de NumPy ofrecen varias ventajas sobre las listas de Python, especialmente cuando se trabaja con datos numéricos en análisis de datos y computación científica. A continuación, exploraremos algunas de estas ventajas en detalle.

Facilidad de Uso

Una de las ventajas principales de los arrays de NumPy es su facilidad de uso para realizar operaciones matemáticas. Puedes escribir expresiones concisas e intuitivas que operan en arrays completos, haciendo tu código más legible y reduciendo la necesidad de bucles explícitos.

Esto significa que puedes realizar operaciones matemáticas complejas con solo unas pocas líneas de código. Por ejemplo, supongamos que tienes dos arrays, `kanto` y `weights`, que representan datos climáticos y los pesos correspondientes. Puedes calcular fácilmente el rendimiento de las manzanas para la región de Kanto de la siguiente manera:

```
yield_of_apples = (kanto * weights).sum()
yield_of_apples
```

```
56.8
```

Aquí, utilizamos la multiplicación elemento a elemento y luego aplicamos la función `sum()` para obtener el rendimiento total, simplificando significativamente el proceso.

Rendimiento

Otra ventaja significativa de los arrays de NumPy es su rendimiento. Las operaciones y funciones de NumPy están implementadas internamente en C o C++, lo que las hace considerablemente más rápidas que usar declaraciones y bucles de Python equivalentes que se interpretan en tiempo de ejecución. Esto resulta en una mejora notable en el tiempo de procesamiento, especialmente cuando se trabaja con grandes conjuntos de datos.

A continuación, vamos a medir el tiempo que se tarda en calcular el rendimiento usando listas de Python en comparación con arrays de NumPy:

```
import time
import numpy as np

# Primero, definimos los pesos y los datos climáticos para la región
de Kanto
weights = [0.3, 0.2, 0.5]  # Pesos para temperatura, precipitación y
humedad
kanto_as_list = [73, 67, 43]  # Datos climáticos para Kanto como lista
```

```
# Función para calcular el rendimiento de manzanas manualmente usando listas de Python
def apples_per_region(lst_):
    counter = 0
    for j, k in zip(weights, lst_):
        counter += j * k
    return counter

start_time = time.time()  # Registrar el tiempo de inicio
result = apples_per_region(kanto_as_list)  # Calcular rendimiento usando listas de Python
end_time = time.time()  # Registrar el tiempo de finalización

elapsed_time = end_time - start_time  # Calcular el tiempo transcurrido
print(f"Yield calculated using Python lists: {result}")
print(f"Time taken: {elapsed_time} seconds")
```

```
Yield calculated using Python lists: 56.8
Time taken: 9.512901306152344e-05 seconds
```

```
# Ahora, realizamos el mismo cálculo usando arrays de NumPy
kanto_array = np.array([73, 67, 43])  # Datos climáticos para Kanto como array de NumPy
weights_array = np.array([0.3, 0.2, 0.5])  # Pesos como array de NumPy

# Medir el tiempo que tarda en calcular el rendimiento usando arrays de NumPy
start_time = time.time()  # Registrar el tiempo de inicio
result = np.dot(weights_array, kanto_array)  # Calcular rendimiento usando arrays de NumPy
end_time = time.time()  # Registrar el tiempo de finalización

elapsed_time = end_time - start_time  # Calcular el tiempo transcurrido
print(f"Yield calculated using NumPy arrays: {result}")
print(f"Time taken: {elapsed_time} seconds")
```

```
Yield calculated using NumPy arrays: 56.8
Time taken: 0.0006070137023925781 seconds
```

La ligera diferencia en el tiempo de ejecución, donde NumPy parece tardar un poco más, se debe principalmente a la sobrecarga adicional introducida por las operaciones internas de NumPy para manejar arrays. Aunque en cálculos simples NumPy puede parecer más lento, su eficiencia se vuelve más evidente en operaciones más complejas.

Rendimiento en grandes conjuntos de datos

Es crucial destacar que el rendimiento de NumPy se vuelve extremadamente beneficioso en escenarios que involucran grandes conjuntos de datos. Por ejemplo, al realizar operaciones en

matrices de miles o millones de elementos, NumPy puede reducir drásticamente el tiempo de ejecución. Esto se debe a su capacidad para ejecutar cálculos en paralelo y aprovechar la optimización de hardware moderno, como las instrucciones SIMD (Single Instruction, Multiple Data).

Complejidad de cálculos

En el caso de operaciones más complejas, como la multiplicación de matrices, el uso de NumPy permite realizar estas tareas con una notación sencilla y eficiente. Con NumPy, puedes realizar transformaciones lineales, cálculos estadísticos y manipulación de datos con facilidad, gracias a sus funciones altamente optimizadas.

Versatilidad y Funciones Adicionales

NumPy no solo se limita a operaciones básicas. La biblioteca incluye una variedad de funciones adicionales, tales como:

- **Manipulación de matrices**: Transposiciones, reshapes y concatenaciones de matrices son fáciles de realizar.
- **Funciones matemáticas avanzadas**: Desde funciones trigonométricas hasta exponenciales, NumPy proporciona una gama completa de funciones matemáticas.
- **Funcionalidades de álgebra lineal**: Descomposición SVD, eigenvectores y eigenvalores, y mucho más.

En resumen, aunque en cálculos simples NumPy pueda parecer un poco más lento, su conveniencia, facilidad de uso y rendimiento sobresaliente en operaciones complejas y grandes conjuntos de datos hacen de NumPy una herramienta invaluable en el análisis de datos y la computación científica. Al aprovechar estas ventajas, puedes escribir código más limpio, más expresivo y lograr un mejor rendimiento en tus tareas de análisis de datos y computación científica.

Crear Arrays

En análisis de datos y computación científica, generar datos aleatorios es una tarea común. NumPy ofrece herramientas poderosas para crear arrays llenos de números aleatorios, que pueden ser particularmente útiles para tareas como simular experimentos, generar conjuntos de datos sintéticos o probar algoritmos.

Números Aleatorios

Random Randint vs. Random Random

En esta sección, exploraremos dos funciones esenciales para generar números aleatorios en NumPy: **`numpy.random.randint`** y **`numpy.random.random`**.

- **`numpy.random.randint`**: Esta función genera números enteros aleatorios dentro de un rango especificado. Es útil para crear arrays de números enteros cuando se simulan escenarios que involucran valores discretos. Puedes especificar el rango inferior, el rango superior y la cantidad de números que deseas generar. Por ejemplo:

```
import numpy as np

# Generar un array de 5 números enteros aleatorios entre 0 y 10
random_integers = np.random.randint(0, 10, size=5)
print("Array de Números Enteros Aleatorios:")
print(random_integers)
```

```
Array de Números Enteros Aleatorios:
[5 3 1 4 5]
```

- **`numpy.random.random`**: Esta función produce números de punto flotante aleatorios entre 0 y 1. Es adecuada para tareas que requieren variables aleatorias continuas, como modelar la incertidumbre en experimentos científicos. También puedes especificar la forma del array que deseas generar. Por ejemplo:

```
# Generar un array de 10 números de punto flotante aleatorios entre 0 y 1
random_numbers = np.random.random(10)
print("Array de Números Aleatorios:")
print(random_numbers)
```

```
Array de Números Aleatorios:
[0.3541322  0.15083313 0.22700565 0.09899985 0.6522254  0.25388205
 0.6915714  0.1342997  0.51061752 0.08070081]
```

Entendiendo estas dos funciones, ganarás la capacidad de generar conjuntos diversos de datos aleatorios adaptados a tus necesidades específicas. Ya sea que estés realizando simulaciones estadísticas, probando algoritmos o explorando distribuciones de probabilidad, las capacidades de generación de números aleatorios de NumPy serán invaluables.

Profundicemos en cada una de estas funciones para ver cómo funcionan y cómo usarlas efectivamente.

```
# Importar la biblioteca NumPy como np
import numpy as np

# Generar un array de 10 números de punto flotante aleatorios entre 0 y 1
random_numbers = np.random.random(10)

# Imprimir el array generado
print("Array de Números Aleatorios:")
print(random_numbers)
```

```
Array de Números Aleatorios:
[0.30723639 0.26476293 0.82676288 0.59000145 0.3734221  0.44536644
 0.58407067 0.25824943 0.14323182 0.42565978]
```

FUERA DE TEMA

- **Aleatoriedad**: La aleatoriedad es un concepto relacionado con la imprevisibilidad y el azar en datos o eventos. La comprensión de la aleatoriedad es fundamental en muchas áreas, como estadística y teoría de probabilidades.
- **Tiempo en Segundos**: Medir el tiempo en segundos puede ser útil para cronometrar eventos o procesos. En contextos de programación, a menudo se utilizan para evaluar el rendimiento de algoritmos.
- **Coordenadas**: Las coordenadas representan puntos en el espacio o en un plano, a menudo utilizadas en geometría y cartografía. En programación, se utilizan para definir posiciones en gráficos o mapas.

```
# Crear una función para mostrar información sobre un array de NumPy
def numpy_info(arr):
    print("\nLa dimensión es:", arr.ndim)
    print("La forma es:", arr.shape)
    print("El tamaño es:", arr.size)
```

La función `numpy_info` está diseñada para proporcionar información sobre un array de NumPy. Toma un array de entrada `arr` y muestra tres características clave del array:

1. **Dimensión (`arr.ndim`)**: Este método devuelve el número de dimensiones en el array de entrada `arr`. Por ejemplo, si `arr` es un array unidimensional, devolverá `1` (vector). Si es un array bidimensional, devolverá `2` (matriz), y así sucesivamente.
2. **Forma (`arr.shape`)**: Este método devuelve una tupla que representa las dimensiones del array. La tupla contiene la longitud del array a lo largo de cada dimensión. Por ejemplo, si `arr` es un array unidimensional con 7 elementos, devolverá `(7,)`. Si es un array bidimensional con dimensiones 3x4, devolverá `(3, 4)`.
3. **Tamaño (`arr.size`)**: Este método devuelve el número total de elementos en el array. Es esencialmente el producto de las longitudes de todas las dimensiones. Por ejemplo, si `arr` es un array bidimensional con dimensiones 3x4, devolverá `12` porque hay un total de 12 elementos en el array.

El propósito de la función `numpy_info` es resumir rápidamente la estructura y el tamaño de un array de NumPy para una mejor comprensión y análisis.

Ejemplo de un Array Unidimensional (`1d array: vector`)

```
# Generar un array de 7 números de punto flotante aleatorios entre 0 y 1
vector = np.random.random(7)

# Llamar a la función numpy_info para mostrar información sobre el array 'vector'
print("\nInformación del Array:")
```

```
numpy_info(vector)

# Imprimir el vector generado
print("\nVector Aleatorio:")
print(vector)
```

```
Información del Array:

La dimensión es: 1
La forma es: (7,)
El tamaño es: 7

Vector Aleatorio:
[0.50222076 0.94798017 0.93133857 0.27557308 0.68592038 0.91175865
 0.84471648]
```

Ejemplo de un Array Bidimensional (`2d array: matriz`)

```
# Generar un array NumPy 2D aleatorio con dimensiones 3x4
matrix = np.random.random((3, 4)) # (filas, columnas)



# Llamar a la función numpy_info para mostrar información sobre el
array 'matrix'
print("\nInformación del Array:")
numpy_info(matrix)

# Imprimir la matriz generada
print("\nMatriz Aleatoria:")
print(matrix)
```

```
Información del Array:

La dimensión es: 2
La forma es: (3, 4)
El tamaño es: 12

Matriz Aleatoria:
[[0.51956851 0.05830975 0.13176167 0.52298225]
 [0.37412562 0.00720151 0.97610186 0.92076738]
 [0.40530797 0.39473668 0.23854654 0.80721374]]
```

Ejemplo de un Array Tridimensional (`3d array: tensor`)

```
# Generar un array NumPy 3D aleatorio con dimensiones 3 x 4 x 5
tensor = np.random.randint(3, size=(3, 4, 5)) # (bloques, filas,
```

```
columnas)


# Llamar a la función numpy_info para mostrar información sobre el
array 'tensor'
print("\nInformación del Array:")
numpy_info(tensor)

# Imprimir el tensor generado
print("\nTensor Aleatorio:")
print(tensor)
```

```
Información del Array:

La dimensión es: 3
La forma es: (3, 4, 5)
El tamaño es: 60

Tensor Aleatorio:
[[[2 1 2 2 1]
  [1 2 0 1 1]
  [0 0 0 2 0]
  [1 1 1 2 2]]

 [[0 2 1 2 0]
  [2 0 0 2 2]
  [1 2 2 1 2]
  [2 2 0 0 2]]

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

enter-the-matrix-10-638.png

NumPy también proporciona una variedad de funciones convenientes para crear arrays con formas específicas y que contienen valores fijos o aleatorios. Puedes explorar estas funciones de creación de arrays más a fondo consultando la documentación oficial o utilizando la función integrada `help` para obtener información más completa y ejemplos. Estas funciones simplifican el proceso de inicializar arrays para satisfacer tus necesidades específicas de manipulación de datos.

Rango

Puedes usar NumPy para crear fácilmente un rango de valores utilizando la función `np.arange()`. Esta función toma tres argumentos: `start`, `stop` y `step`, y devuelve un array de valores que comienza desde `start`, hasta (pero sin incluir) `stop`, con incrementos de `step`.

He aquí un ejemplo:

```
# Hace lo mismo que la función range() de Python incorporada
range_of_values = np.arange(0, 100)
```

Esto generará un array que incluye los números del 0 al 99. La función `np.arange()` es especialmente útil cuando necesitas crear secuencias de números con pasos específicos, lo que resulta ideal para configuraciones de bucles o análisis de datos.

Espaciado Uniforme

La función `linspace` de NumPy es una herramienta versátil para crear secuencias de valores espaciados uniformemente dentro de un intervalo especificado. Te permite definir un punto de inicio y un punto final para un rango dado y determinar el número total de valores espaciados uniformemente que deseas dentro de ese intervalo. Es importante destacar que esta secuencia incluye tanto el punto de inicio como el de finalización.

Así es como puedes usar la función `np.linspace()`:

```
np.linspace(0, 100, 50)
```

```
array([  0.        ,   2.04081633,   4.08163265,   6.12244898,
         8.16326531,  10.20408163,  12.24489796,  14.28571429,
        16.32653061,  18.36734694,  20.40816327,  22.44897959,
        24.48979592,  26.53061224,  28.57142857,  30.6122449 ,
        32.65306122,  34.69387755,  36.73469388,  38.7755102 ,
        40.81632653,  42.85714286,  44.89795918,  46.93877551,
        48.97959184,  51.02040816,  53.06122449,  55.10204082,
        57.14285714,  59.18367347,  61.2244898 ,  63.26530612,
        65.30612245,  67.34693878,  69.3877551 ,  71.42857143,
        73.46938776,  75.51020408,  77.55102041,  79.59183673,
        81.63265306,  83.67346939,  85.71428571,  87.75510204,
        89.79591837,  91.83673469,  93.87755102,  95.91836735,
        97.95918367, 100.        ])
```

En esta función, `start` representa el inicio del intervalo, `end` es el punto final y `num` es el número total de valores que deseas en la secuencia. NumPy generará una secuencia de valores que están distribuidos uniformemente entre `start` y `end`, incluyendo ambos puntos finales.

La función `linspace` es particularmente útil para tareas como definir valores a lo largo de un eje continuo para trazar gráficos o crear intervalos espaciados uniformemente para cálculos numéricos.

Para más detalles y opciones, puedes referirte a la documentación oficial de NumPy `linspace`.

Redondeo

Permite redondear los arrays a los decimales que le indiquemos 🙃

```
np.round(np.linspace(0, 100, 50), 3)
```

```
array([  0.   ,   2.041,   4.082,   6.122,   8.163,  10.204,  12.245,
        14.286,  16.327,  18.367,  20.408,  22.449,  24.49 ,  26.531,
        28.571,  30.612,  32.653,  34.694,  36.735,  38.776,  40.816,
        42.857,  44.898,  46.939,  48.98 ,  51.02 ,  53.061,  55.102,
        57.143,  59.184,  61.224,  63.265,  65.306,  67.347,  69.388,
        71.429,  73.469,  75.51 ,  77.551,  79.592,  81.633,  83.673,
        85.714,  87.755,  89.796,  91.837,  93.878,  95.918,  97.959,
       100.   ])
```

Esto redondeará los valores generados a tres decimales.

Arrays desde datos

En NumPy, puedes crear fácilmente arrays a partir de estructuras de datos existentes como listas. Esta sección explora varias formas de convertir entre listas de Python y arrays de NumPy.

Convertir una Lista en un Array de NumPy

Supongamos que tienes una lista de Python con algunos valores:

```
a = [90, 50, 0]
```

Puedes convertir esta lista en un array de NumPy usando `np.array()`:

```
a = np.array(a)
a

array([90, 50,  0])
```

Ahora, la variable `a` contiene un array de NumPy con los mismos valores:

Convertir un Array en una Lista: Si tienes un array de NumPy y quieres convertirlo de nuevo a una lista de Python, tienes algunas opciones:

1. Usando el método `tolist()`:

```
b = a.tolist()
b

[90, 50, 0]
```

Esto creará una nueva lista `b` que contiene los valores del array de NumPy.

1. Usando el constructor `list()`:

Puedes convertir directamente el array de NumPy a una lista de Python usando el constructor `list()`.

```
list(a)

[90, 50, 0]
```

Arrays Multidimensionales: NumPy también puede manejar arrays multidimensionales con facilidad. Por ejemplo, si quieres crear un array 3D con enteros aleatorios:

```
a = np.random.randint(10, size=(1, 2, 3))
a

array([[[7, 5, 3],
        [5, 7, 9]]])
```

Esto resultará en un array 3D de NumPy. Ten cuidado al convertir tales arrays en listas, ya que la estructura puede volverse anidada y menos intuitiva.

```
c_badly_done = list(a)
c_badly_done

[array([[7, 5, 3],
        [5, 7, 9]])]
```

En el ejemplo anterior, `c_badly_done` será una lista anidada con la estructura que refleja las dimensiones del array original.

Conclusión: Entender cómo convertir datos entre listas de Python y arrays de NumPy es esencial para trabajar con datos de manera eficiente. Dependiendo de tus necesidades, puedes cambiar fácilmente entre estas estructuras de datos mientras te aseguras de manejar las dimensiones adecuadamente.

Siempre ten en cuenta la estructura de datos con la que estás trabajando, especialmente al manejar arrays multidimensionales, ya que convertirlos en listas puede resultar en estructuras anidadas que requieren manejo adicional.

Array de CEROS

`np.zeros` se utiliza para crear el array donde todos los elementos son 0. Su sintaxis es:

```
shape = (3, 4)
array = np.zeros(shape, dtype=int)
array

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

Donde:

- El `shape` es el tamaño del array, y puede ser 1-D, 2-D o de múltiples dimensiones.
- El `dtype` es `float64` por defecto, pero puede ser configurado a cualquier tipo de datos en NumPy.

```
np.zeros((2, 5, 10))
```

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

       [[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]]])
```

Este ejemplo creará un array 3D con dimensiones 2x5x10, donde todos los elementos son 0.

Array de UNOS

De manera similar a crear arrays de ceros, puedes crear un array donde todos los elementos estén establecidos en 1 usando `np.ones()`. La sintaxis y los parámetros de `np.ones()` son idénticos a los de `np.zeros()`:

```
np.ones((2, 5, 10))
```

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

       [[1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]]])
```

Esto generará un array 3D de unos con las mismas dimensiones que el anterior.

Array Diagonal

La función `np.eye()` en NumPy se utiliza para generar un array bidimensional con unos a lo largo de la diagonal principal y ceros en otros lugares. Esto puede ser especialmente útil cuando necesitas crear matrices identidad, que son matrices cuadradas con unos en la diagonal y ceros en todas partes.

Así es como puedes usar `np.eye()` con sus parámetros:

- **N**: Un entero que especifica el número de filas en el array resultante. Esto determina el tamaño de la matriz cuadrada.
- **M** (opcional): Un entero que especifica el número de columnas en el array. Por defecto, se establece en `None`, lo que significa que será igual a `N`, resultando en una

matriz cuadrada. Sin embargo, puedes establecer `M` en un valor diferente para crear una matriz rectangular.

- **k** (opcional): Un entero, cuyo valor por defecto es 0. Determina la posición de la diagonal. Cuando `k` es 0, coloca unos en la diagonal principal. Si `k` es positivo, desplaza la diagonal hacia arriba, creando una diagonal superior con el desplazamiento de `k`. Por el contrario, si `k` es negativo, desplaza la diagonal hacia abajo, formando una diagonal inferior con el desplazamiento de `-k`.
- **dtype** (opcional): El tipo de datos de los elementos del array. Por defecto, se establece en flotante, pero puedes especificar otros tipos de datos según sea necesario.

```
np.eye(5, 5, dtype=int)
```

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

```
# Devuelve un array 2-D con unos en la diagonal y ceros en otros lugares.
np.eye(5, 5, 2)
```

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

Tipos de Datos (Dtypes)

NumPy proporciona varios tipos de datos (dtypes) para representar y trabajar con diferentes tipos de datos de manera eficiente. Estos tipos de datos son esenciales para especificar el tipo de datos que un array puede contener, y son un aspecto crucial de la funcionalidad de NumPy.

Aquí hay una lista de los tipos de datos comunes de NumPy:

- **int8, int16, int32, int64**: Tipos de enteros con signo con diferentes tamaños de bit.
- **uint8, uint16, uint32, uint64**: Tipos de enteros sin signo (no negativos) con diferentes tamaños de bit.
- **float16, float32, float64**: Tipos de punto flotante con variada precisión.
- **complex64, complex128**: Tipos de números complejos.
- **bool**: Tipo booleano.
- **object**: Tipo de objeto genérico de Python.
- **string_**: Tipo de cadena (cadenas ASCII de tamaño fijo).
- **unicode_**: Tipo de cadena Unicode (cadenas Unicode de tamaño fijo).
- **datetime64**: Tipo de fecha y hora.

- **timedelta64**: Diferencias entre dos valores datetime.
- **void**: Tipo de datos en bruto (para arrays estructurados).
- **structured**: Tipos de datos estructurados definidos por el usuario.
- **unicodechar**: Un único carácter Unicode.

```
np.eye(5, 5) # tipo: float
```

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

```
np.eye(5, 5, dtype="int") # tipo: int
```

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

```
np.eye(5, 5, dtype="str") # tipo: str
```

```
array([['1', '', '', '', ''],
       ['', '1', '', '', ''],
       ['', '', '1', '', ''],
       ['', '', '', '1', ''],
       ['', '', '', '', '1']], dtype='<U1')
```

```
# si un dataframe tiene dtype=object -> es una cadena
np.eye(5, 5, dtype="object") # tipo: object
```

```
array([[1, 0, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 1, 0],
       [0, 0, 0, 0, 1]], dtype=object)
```

Estos tipos de datos te permiten trabajar con una amplia gama de datos de manera eficiente, haciendo de NumPy una biblioteca poderosa para la computación numérica y científica. Puedes explorar la documentación oficial de NumPy para obtener más información detallada sobre estos tipos de datos y su uso.

El Producto Punto y sus Aplicaciones en Ciencia de Datos

1. Definición del Producto Punto

El **producto punto** (también conocido como producto escalar) es una operación entre dos vectores que produce un escalar como resultado. Si tenemos dos vectores a y b en un espacio vectorial n-dimensional, el producto punto se define como:

$$a \cdot b = a_1 b_1 + a_2 b_2 + \ldots + a_n b_n = \sum_{i=1}^{n} a_i b_i$$

Donde:

- $a = \left[a_1, a_2, \ldots, a_n\right)$
- $b = \left[b_1, b_2, \ldots, b_n\right)$

El resultado es un número escalar que refleja la magnitud del paralelismo entre los dos vectores.

2. Aplicaciones del Producto Punto en Ciencia de Datos

En análisis de datos, el producto punto tiene múltiples aplicaciones, especialmente en problemas relacionados con álgebra lineal y geometría. Algunas de sus aplicaciones más comunes incluyen:

a) Similitud entre vectores

El producto punto se usa para medir la **similitud** entre dos vectores. Esto es fundamental en algoritmos de machine learning y sistemas de recomendación, donde se busca comparar la similitud entre puntos de datos representados como vectores.

Un caso especial es cuando los vectores están normalizados (es decir, tienen longitud 1), en cuyo caso el producto punto es equivalente al **coseno del ángulo** entre ellos:

$$\cos(\theta) = \frac{a \cdot b}{|a)|b)}$$

b) Regresión Lineal

En la regresión lineal, el producto punto entre un vector de características y un vector de coeficientes nos da la predicción para un modelo. Por ejemplo, para un vector de características x y un vector de pesos w, la predicción $\hat{y}$ es:

$$\hat{y} = w \cdot x$$

c) Optimización y Gradientes

El producto punto se utiliza de manera crucial en el cálculo de **gradientes** y la **optimización** de funciones objetivo en muchos algoritmos de machine learning, como el descenso por gradiente.

En el contexto del descenso por gradiente, el objetivo es minimizar una función de pérdida $L(x)$, que a menudo se expresa como una función cuadrática. Por ejemplo, para un modelo de regresión lineal, la función de pérdida se puede definir como:

$$L(x) = \frac{1}{2} \sum_{i=1}^{m} (y_i - w \cdot x_i)^2$$

Aquí, w son los pesos del modelo, y_i son las salidas reales y x_i son los vectores de características. La expresión $y_i - w \cdot x_i$ es la diferencia entre el valor real y la predicción, y el producto punto entre w y x_i da la predicción del modelo.

Para minimizar la función de pérdida, se calcula el gradiente de $L(x)$ con respecto a los pesos w, utilizando el producto punto:

$$\nabla L(x) = -\sum_{i=1}^{m} (y_i - w \cdot x_i) x_i$$

Aquí, el producto punto es esencial, ya que permite computar la dirección en la que se debe ajustar los pesos para reducir el error de predicción. Esta información se utiliza para actualizar los pesos en cada iteración del algoritmo de descenso por gradiente:

$$w \leftarrow w - \eta \nabla L(x)$$

Donde η es la tasa de aprendizaje.

3. Matrices de Correlación

El producto punto está estrechamente relacionado con las **matrices de correlación**, que son una herramienta fundamental en análisis de datos para comprender las relaciones entre variables.

La correlación entre dos variables es una medida de la fuerza y la dirección de su relación lineal. En términos del producto punto, la correlación entre dos vectores a y b (representando dos variables) se calcula como:

$$\text{correlación}(a, b) = \frac{a \cdot b}{|a||b|}$$

Esto es una extensión del coseno del ángulo mencionado anteriormente, normalizando los vectores por sus magnitudes.

Las matrices de correlación se construyen calculando la correlación entre cada par de variables en un conjunto de datos. Esto es útil para:

- Identificar relaciones lineales entre variables.
- Seleccionar características relevantes para modelos predictivos.

- Detectar multicolinealidad en los datos.

4. Transformación de Sistemas Lineales Ax = b

En álgebra lineal aplicada a ciencia de datos, uno de los problemas más comunes es la solución de sistemas lineales de la forma:

$$A\,x = b$$

Donde A es una matriz de coeficientes, x es el vector de incógnitas, y b es el vector de términos independientes.

En muchos casos, es deseable transformar este sistema a uno **ortogonal**, donde la matriz es más fácil de invertir. Esto se logra mediante la **descomposición ortogonal**, como la descomposición QR o la descomposición de valores singulares (SVD).

a) Sistemas Ortogonales

Un sistema ortogonal tiene una matriz A tal que $A^T = A^{-1}$, lo que simplifica la solución de $A\,x = b$. Esta propiedad reduce el costo computacional al resolver sistemas de ecuaciones en ciencia de datos, ya que la inversa de A se calcula simplemente tomando su transpuesta.

b) Descomposición QR

La **descomposición QR** es una técnica que transforma la matriz A en el producto de una matriz ortogonal Q y una matriz triangular superior R, de la siguiente forma:

$$A = Q\,R$$

Al resolver el sistema $A\,x = b$, se convierte en:

$$Q\,R\,x = b$$

Debido a que Q es ortogonal ($Q^T = Q^{-1}$), la solución se simplifica, optimizando el tiempo de cómputo en problemas de gran escala.

5. Ejemplo de Utilidad de Sistemas Ortogonales en Ciencia de Datos

Los sistemas ortogonales son útiles en la reducción de la dimensionalidad y el ajuste de modelos. Por ejemplo, en **análisis de componentes principales (PCA)**, se busca transformar un conjunto de variables correlacionadas en un conjunto de variables no correlacionadas (componentes principales) mediante una transformación ortogonal.

Ejemplo de una Matriz de 5x5

A continuación se muestra un ejemplo de cómo generar una matriz de 5×5 y calcular su descomposición QR utilizando Python:

```python
import numpy as np

# Generar una matriz aleatoria de 5x5
A = np.random.rand(5, 5)

# Realizar la descomposición QR
Q, R = np.linalg.qr(A)

# Mostrar resultados
print("Matriz A:")
print(A)
print("\nMatriz Q (Ortogonal):")
print(Q)
print("\nMatriz R (Triangular Superior):")
print(R)
```

```
Matriz A:
[[0.40186403 0.69145842 0.50023028 0.60919662 0.43457889]
 [0.64953349 0.47945585 0.36252956 0.81999958 0.38359487]
 [0.53225728 0.81381058 0.01552744 0.76444987 0.7790143 ]
 [0.64887369 0.60858311 0.59428608 0.19098333 0.0093412 ]
 [0.6601411  0.11766163 0.20965507 0.08423151 0.22408308]]

Matriz Q (Ortogonal):
[[-0.30610631  0.50681562  0.45867037  0.18489358  0.63629609]
 [-0.49476013 -0.12411592  0.02509444  0.7706375  -0.38117682]
 [-0.40542895  0.52048197 -0.73146248 -0.16907259 -0.0332112 ]
 [-0.49425755  0.06759995  0.47157791 -0.55447774 -0.4704343 ]
 [-0.50284011 -0.67250422 -0.17767577 -0.18947588  0.47688585]]

Matriz R (Triangular Superior):
[[-1.31282505 -1.13877958 -0.73793746 -1.03886208 -0.75594466]
 [ 0.          0.67651981  0.11579037  0.56112215  0.42803869]
 [ 0.          0.          0.47018211 -0.1840709  -0.3962742 ]
 [ 0.          0.          0.          0.49345561  0.19661565]
 [ 0.          0.          0.          0.          0.20689901]]
```

6. Resolución de Sistemas de la Forma Ax = b

Para resolver un sistema de la forma $A\,x = b$ usando una matriz ortogonal, primero aplicamos la descomposición QR. Esto nos permite reescribir el sistema en la forma:

$$Q\,R\,x = b$$

Multiplicando ambos lados por Q^T (la transpuesta de Q), obtenemos:

$$R\,x = Q^T\,b$$

Aquí, $Q^T\,b$ es un nuevo vector que se puede calcular fácilmente. Luego, el sistema se reduce a resolver un sistema triangular superior, lo que es computacionalmente eficiente.

Ejemplo en Python

A continuación se presenta un ejemplo donde se transforma un sistema $Ax=b$ a un sistema con una matriz ortogonal usando Python:

```python
import numpy as np

# Generar una matriz aleatoria de 5x5
A = np.random.rand(5, 5)

# Generar un vector b aleatorio
b = np.random.rand(5)

# Realizar la descomposición QR
Q, R = np.linalg.qr(A)

# Resolver el sistema original Ax = b
# Multiplicar ambos lados por Q^T
y = np.dot(Q.T, b)

# Resolver el sistema triangular superior R x = y
x = np.linalg.solve(R, y)

# Mostrar resultados
print("Matriz A:")
print(A)
print("\nVector b:")
print(b)
print("\nVector solución x:")
print(x)
```

```
Matriz A:
[[0.88809757 0.73705744 0.3409491  0.91580973 0.72325107]
 [0.84226027 0.52443745 0.75541929 0.717066   0.64338484]
 [0.95475598 0.16099476 0.19095743 0.40980075 0.78431096]
 [0.54562549 0.25219528 0.60526843 0.62028872 0.79569489]
 [0.91370628 0.69929963 0.85766542 0.08635919 0.54720287]]

Vector b:
[0.92880939 0.26892028 0.51413059 0.1173077  0.88784934]

Vector solución x:
[ 0.06637615  1.72853233 -0.90483155 -0.76972276  0.84238294]
```

Este código genera una matriz aleatoria A de tamaño 5×5 y un vector b aleatorio. Luego, realiza la descomposición QR de A, transforma el sistema usando la transpuesta de Q, y finalmente resuelve el sistema triangular superior resultante.

7. Optimización en Ciencia de Datos

El producto punto y su relación con las matrices ortogonales y su transpuesta son fundamentales en muchos algoritmos de optimización en ciencia de datos. Las transformaciones ortogonales permiten simplificar y acelerar el proceso de resolución de sistemas lineales. Algunas aplicaciones clave incluyen:

- **Descomposición QR**: La descomposición QR de una matriz permite expresar un sistema de ecuaciones lineales como $A\,x = b$ en términos de una matriz ortogonal Q y una matriz triangular superior R. Esto transforma el sistema en una forma que es más fácil de resolver, ya que se puede usar la transpuesta de la matriz ortogonal para simplificar el cálculo:

  $$Q^T A\,x = Q^T b$$

  Al resolver el sistema triangular $R\,x = Q^T b$, se aprovechan las propiedades de la ortogonalidad, lo que reduce el tiempo de cómputo.

- **Optimización de Funciones Cuadráticas**: En problemas de optimización, muchas veces se busca minimizar funciones cuadráticas de la forma $L(x) = \frac{1}{2}\,x^T Q\,x - b^T x$.

  Aquí, el uso de la transpuesta y de matrices ortogonales permite resolver el sistema de manera eficiente. La solución se obtiene usando la relación entre la matriz transpuesta y la inversa de matrices ortogonales:

  $$x = Q\left(Q^T Q\right)^{-1} Q^T b$$

  En este caso, la ortogonalidad asegura que $Q^T Q = I$, lo que simplifica aún más el cálculo.

- **Reducción Dimensional con PCA**: En el análisis de componentes principales (PCA), las transformaciones ortogonales son clave para identificar las direcciones principales de los datos y, por lo tanto, reducir su dimensionalidad. El proceso de PCA se puede desglosar en varios pasos:

  a. **Cálculo de la Matriz de Covarianza**: Para un conjunto de datos centrados (donde la media de cada característica es cero), la matriz de covarianza se calcula como:

  $$C = \frac{1}{n-1} X^T X$$

  Aquí, X es la matriz de datos, donde cada columna representa una característica y cada fila una observación. La transpuesta de X permite que el cálculo de covarianza considere todas las combinaciones de pares de características.

  a. **Cálculo de Valores y Vectores Propios**: A partir de la matriz de covarianza, se calculan los valores propios y los vectores propios. Los vectores propios de la matriz de covarianza representan las direcciones de máxima varianza

en los datos y son ortogonales entre sí. Se ordenan en función de sus valores propios, que indican la cantidad de varianza explicada por cada componente.

b. **Proyección de los Datos**: Para reducir la dimensionalidad, se seleccionan los primeros k vectores propios (componentes principales) que explican la mayor parte de la varianza. Los datos originales se proyectan en este nuevo espacio de características usando la transpuesta de la matriz de vectores propios:

$$Z = X\,V_k$$

Donde V_k es la matriz que contiene los primeros k vectores propios y Z es la matriz de datos transformada, que tiene una dimensión reducida.

Al utilizar transformaciones ortogonales y la transpuesta en la optimización de algoritmos, se mejora la estabilidad numérica y se acelera el tiempo de cómputo. En el caso de PCA, esto permite realizar la reducción de dimensionalidad de manera eficiente, manteniendo la mayor cantidad de información posible y facilitando la visualización y análisis posterior de los datos.

Transposición

En el análisis de datos y álgebra lineal, la transposición de una matriz es una operación fundamental que voltea la matriz sobre su diagonal. Esta operación intercambia las filas y columnas de la matriz, cambiando efectivamente la orientación de los datos.

La transposición se utiliza comúnmente en varias operaciones y transformaciones matemáticas, lo que la convierte en un concepto crucial en la computación numérica.

Puedes pensar en la transposición como una forma de representar los mismos datos desde una perspectiva diferente, especialmente cuando se trata de arrays o matrices multidimensionales.

A menudo se utiliza para alinear los datos correctamente para operaciones matemáticas, incluyendo la multiplicación de matrices, la resolución de ecuaciones lineales y más.

En NumPy, la función `numpy.transpose()` te permite transponer arrays y matrices fácilmente, proporcionando una herramienta versátil para la manipulación y análisis de datos.

Transposición de Matriz

Ten en cuenta que la imagen anterior es una representación simplificada de la transposición de matrices, mostrando el cambio de orientación de filas a columnas y viceversa. En la práctica, la operación de transposición puede involucrar transformaciones de datos complejas, pero sirve como un componente fundamental para varias computaciones matemáticas y manipulaciones de datos.

Ejemplo de transposición

A continuación, se muestra un ejemplo de cómo transponer una matriz en NumPy:

```
import numpy as np
```

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

# Mostrar la matriz original
print("Matriz Original:")
print(matrix)

# Usar numpy.transpose() para transponer la matriz
transposed_matrix = np.transpose(matrix)

# Mostrar la matriz transpuesta
print("\nMatriz Transpuesta:")
print(transposed_matrix)

# Alternativamente, puedes usar el atributo T para la transposición
# Esto logra el mismo resultado que np.transpose()
transposed_matrix_alt = matrix.T
print("\nMatriz Transpuesta Alternativa:")
print(transposed_matrix_alt)

# Verificar si los dos métodos producen el mismo resultado
print("\n¿Son iguales las dos matrices transpuestas?")
print(np.array_equal(transposed_matrix, transposed_matrix_alt))
```

```
Matriz Original:
[[1 2 3]
 [4 5 6]
 [7 8 9]]

Matriz Transpuesta:
[[1 4 7]
 [2 5 8]
 [3 6 9]]

Matriz Transpuesta Alternativa:
[[1 4 7]
 [2 5 8]
 [3 6 9]]

¿Son iguales las dos matrices transpuestas?
True
```

Aplicaciones de la transposición

La transposición tiene múltiples aplicaciones en álgebra lineal, como en la solución de sistemas de ecuaciones lineales, donde se requieren matrices transpuestas para aplicar métodos como el

método de Gauss-Jordan. También es útil en el aprendizaje automático, donde la transposición se utiliza en algoritmos de optimización y redes neuronales para ajustar pesos y bias.

Cambiar la forma de los arrays en NumPy

Cambiar la forma de los arrays es una operación fundamental en la manipulación y análisis de datos. Implica cambiar las dimensiones o la forma de un array mientras se preservan sus datos originales. NumPy, una poderosa biblioteca para computación numérica en Python, proporciona una función versátil llamada `numpy.reshape()` para este propósito.

Cambiar la forma permite convertir arrays 1D en arrays 2D (y viceversa), reorganizar las dimensiones de arrays multidimensionales y preparar eficientemente los datos para diversas tareas de análisis de datos y aprendizaje automático.

En esta sección, exploraremos cómo cambiar la forma de los arrays en NumPy, incluyendo la transformación de arrays entre diferentes dimensiones y examinando las propiedades de los arrays con forma modificada.

Cambio de forma de array

Ejemplo de cambio de forma

A continuación se muestra un ejemplo de cómo cambiar la forma de un array en NumPy:

```
import numpy as np

# Crear un array 1D
array_1d = np.array([1, 2, 3, 4, 5, 6])

# Mostrar el array 1D original
print("Array 1D Original:")
print(array_1d)

# Cambiar la forma del array 1D a un array 2D con 2 filas y 3 columnas
array_2d = np.reshape(array_1d, (2, 3))

# Mostrar el array 2D reconfigurado
print("\nArray 2D Reconfigurado:")
print(array_2d)

# Comprobar la forma del array reconfigurado
print("\nForma del Array Reconfigurado:", array_2d.shape)

# Cambiar la forma del array 2D de vuelta a un array 1D
array_1d_reshaped = np.reshape(array_2d, -1)

# Mostrar el array 1D reconfigurado
print("\nArray 1D Reconfigurado:")
print(array_1d_reshaped)
```

```
Array 1D Original:
[1 2 3 4 5 6]

Array 2D Reconfigurado:
[[1 2 3]
 [4 5 6]]

Forma del Array Reconfigurado: (2, 3)

Array 1D Reconfigurado:
[1 2 3 4 5 6]
```

Consideraciones sobre el cambio de forma

Es importante tener en cuenta que al cambiar la forma, el número total de elementos debe permanecer constante. Por ejemplo, un array de 6 elementos puede cambiar su forma a 2 filas y 3 columnas, pero no puede cambiar a 2 filas y 4 columnas, ya que eso requeriría 8 elementos.

Actualizando el valor de un array de n dimensiones

En el análisis de datos y la computación numérica, a menudo es necesario modificar los valores de elementos individuales dentro de arrays multidimensionales. NumPy, una poderosa biblioteca para la manipulación de arrays en Python, proporciona varias técnicas para actualizar los valores de arrays n-dimensionales de manera eficiente.

En esta sección, exploraremos métodos y técnicas para actualizar los valores de arrays n-dim en NumPy. Ya sea que estés trabajando con limpieza de datos, transformación o cualquier tarea de manipulación de datos, saber cómo actualizar los valores de los arrays es una habilidad esencial.

1. Sobrescribiendo valores

```python
import numpy as np

# Crear un array 2D de ejemplo
original_array = np.array([[1, 2, 3],
                           [4, 5, 6],
                           [7, 8, 9]])

# Mostrar el array original
print("Array Original:")
print(original_array)

# Hacer una copia del array original
updated_array = original_array.copy()

# Actualizar un elemento específico sobrescribiéndolo en la copia
updated_array[1, 1] = 99

# Mostrar el array actualizado
```

```
print("Array Actualizado:")
print(updated_array)
```

```
Array Original:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
Array Actualizado:
[[ 1  2  3]
 [ 4 99  6]
 [ 7  8  9]]
```

2. Usando su índice

```
# Actualizar un elemento específico utilizando su índice
updated_array = original_array.copy()
original_array[0][-1] = 60

#Mostrar el array actualizado
print("Array Actualizado:")
print(original_array)
```

```
Array Actualizado:
[[ 1  2 60]
 [ 4  5  6]
 [ 7  8  9]]
```

3. Otra sintaxis

Crear un array 2D de NumPy array_new = np.array(([1, 2, 3], [2, 2, 0]))

Usamos indexación booleana para seleccionar elementos en `array_new` donde la condición `array_new == 2` es verdadera. Esta operación devuelve un array que contiene todos los elementos iguales a 2. array_new[array_new == 2]

Actualizamos los elementos en `array_new` donde la condición `array_new == 2` es verdadera. En este caso, establecemos esos elementos en 20. array_new[array_new == 2] = 20

Creamos una máscara booleana comparando cada elemento de `array_new` con 20. Esto resulta en un array booleano donde cada elemento es `True` si el elemento correspondiente en array_new es igual a 20, y `False` en caso contrario. array_new == 20 # máscara

Comparación de Arrays

Los arrays de NumPy no solo almacenan datos de manera eficiente, sino que también admiten una amplia gama de operaciones de comparación. Estas operaciones te permiten comparar elementos dentro de los arrays, resultando en nuevos arrays de valores booleanos.

Estos arrays booleanos sirven como máscaras, lo que te permite extraer o manipular elementos basados en condiciones específicas.

Ejemplo de comparación de arrays

```
import numpy as np

# Crear dos arrays de NumPy
array1 = np.array([1, 2, 3, 4, 5])
array2 = np.array([3, 4, 5, 6, 7])

# Comparación de igualdad
equal_result = (array1 == array2)
print("Resultado de Comparación de Igualdad:")
print(equal_result)

# Comparación de desigualdad
not_equal_result = (array1 != array2)
print("\nResultado de Comparación de Desigualdad:")
print(not_equal_result)

# Comparación mayor que
greater_than_result = (array1 > array2)
print("\nResultado de Comparación Mayor que:")
print(greater_than_result)

# Comparación menor que
less_than_result = (array1 < array2)
print("\nResultado de Comparación Menor que:")
print(less_than_result)
```

```
Resultado de Comparación de Igualdad:
[False False False False False]

Resultado de Comparación de Desigualdad:
[ True  True  True  True  True]

Resultado de Comparación Mayor que:
[False False False False False]

Resultado de Comparación Menor que:
[ True  True  True  True  True]
```

Indexación y Segmentación en NumPy

NumPy extiende la notación de indexación de listas de Python usando [] a múltiples dimensiones de una manera intuitiva. Con NumPy, puedes proporcionar una lista de índices o rangos separados por comas para seleccionar elementos específicos o subarrays, también conocidos como segmentos, de un array de NumPy.

Esta poderosa característica te permite acceder y manipular datos dentro de arrays multidimensionales de manera eficiente.

Ejemplo de indexación y segmentación

```
import numpy as np

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

# Accediendo a elementos individuales
element = arr[1, 2]  # Accede al elemento en la segunda fila y tercera columna
print("Elemento Individual:", element)

# Segmentación de subarrays
subarray = arr[0:2, 1:3]  # Selecciona un subarray que consiste en filas 0 a 1 y columnas 1 a 2
print("\nSubarray:")
print(subarray)

# Usando paso en segmentación
step_array = arr[:2, :2]  # Selecciona filas de 0 a 1 y columnas de 0 a 1
print("\nSegmentación con Paso:")
print(step_array)
```

```
Elemento Individual: 6

Subarray:
[[2 3]
 [5 6]]

Segmentación con Paso:
[[1 2]
 [4 5]]
```

Métodos de np.arrays

Los arrays de NumPy vienen equipados con una amplia gama de métodos incorporados que hacen que realizar varias operaciones en arrays sea más conveniente y eficiente.

Estos métodos proporcionan funcionalidades para tareas como operaciones matemáticas, agregación, estadísticas y manipulación de datos.

Para explorar la extensa lista de métodos disponibles para los arrays de NumPy, puedes referirte a la documentación oficial de NumPy.

Esta documentación proporciona información detallada sobre cada método junto con ejemplos de su uso.

Ejemplo de métodos de arrays

```
import numpy as np

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

# Método 1: Ordenar el array
sorted_arr = np.sort(arr)
print("Array Ordenado:", sorted_arr)
```

```
Array Ordenado: [1 2 3 4 5]
```

```
# Método 2: Encontrar los valores máximos y mínimos
max_value = np.max(arr)
min_value = np.min(arr)
print("Valor Máximo:", max_value)
print("Valor Mínimo:", min_value)
```

```
Valor Máximo: 5
Valor Mínimo: 1
```

```
# Método 3: Calcular la suma y la media
sum_values = np.sum(arr)
mean_value = np.mean(arr)
print("Suma de Valores:", sum_values)
print("Media de Valores:", mean_value)
```

```
Suma de Valores: 15
Media de Valores: 3.0
```

```
# Método 4: Cambiar la forma del array
reshaped_arr = arr.reshape(1, 5)
print("Array Reconfigurado:", reshaped_arr)
```

```
Array Reconfigurado: [[3 1 2 5 4]]
```

```
# Método 5: Calcular la raíz cuadrada
sqrt_arr = np.sqrt(arr)
print("Raíz Cuadrada del Array:", sqrt_arr)
```

```
Raíz Cuadrada del Array: [1.73205081 1.         1.41421356 2.23606798
2.        ]
```

```
# Método 6: Aplicar una función matemática elemento por elemento
exp_arr = np.exp(arr)
print("Exponencial del Array:", exp_arr)
```

```
Exponencial del Array: [ 20.08553692   2.71828183   7.3890561
148.4131591   54.59815003]
```

```
# Método 7: Encontrar valores únicos y sus conteos
unique_values, counts = np.unique(arr, return_counts=True)
```

```
print("Valores Únicos:", unique_values)
print("Conteos:", counts)
```

```
Valores Únicos: [1 2 3 4 5]
Conteos: [1 1 1 1 1]
```

Cosas divertidas que nos permite hacer Numpy

Numpy es una biblioteca versátil que ofrece una amplia gama de funcionalidades más allá del cálculo numérico. Abre oportunidades para diversas tareas de manipulación de datos, incluido el procesamiento de imágenes. En esta sección, exploraremos algunas cosas emocionantes que puedes hacer con Numpy, desde analizar imágenes hasta otras aplicaciones divertidas.

Explorando la Versatilidad de Numpy

Para descubrir la miríada de funcionalidades que Numpy ofrece, puedes usar la función `dir(object)`, que muestra todos los posibles métodos que se pueden aplicar a un objeto de Numpy. Esto te permite explorar las extensas capacidades de Numpy para diferentes tareas de manipulación de datos.

Procesamiento de Imágenes con Pillow y scikit-image

Además de los datos numéricos, Numpy se puede utilizar para trabajar con imágenes a través de bibliotecas como Pillow y scikit-image:

- Pillow: Pillow, la Biblioteca de Imágenes de Python, proporciona capacidades completas de procesamiento de imágenes para tu intérprete de Python. Puedes abrir y manipular imágenes con facilidad utilizando funciones como `Image.open("ruta")`.
- scikit-image: Scikit-image extiende tus capacidades de procesamiento de imágenes en Python. Con funciones como `io.imread("ruta")`, puedes leer y procesar imágenes de manera eficiente para diversas aplicaciones.

En las próximas secciones, profundizaremos en ejemplos prácticos y exploraremos el lado divertido de Numpy, demostrando su versatilidad y utilidad en diversas tareas relacionadas con datos.

¡Podemos procesar imágenes con Numpy!

```
# Importar la biblioteca de scikit-image
from skimage import io

# Leer una imagen utilizando skimage
photo = io.imread("img/img_prueba.png")
```

Al leer la imagen, la variable `photo` contendrá una representación numérica de la imagen, donde cada píxel está representado por valores de color RGB.

```
photo
```

```
array([[[255, 255, 255],
        [255, 255, 255],
        [255, 255, 255],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]],

       [[255, 255, 255],
        [255, 255, 255],
        [255, 255, 255],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]],

       [[255, 255, 255],
        [255, 255, 255],
        [255, 255, 255],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]],

       ...,

       [[255, 255, 255],
        [255, 255, 255],
        [255, 255, 255],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]],

       [[255, 255, 255],
        [255, 255, 255],
        [255, 255, 255],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]],

       [[255, 255, 255],
        [255, 255, 255],
        [255, 255, 255],
        ...,
        [255, 255, 255],
```

```
        [255, 255, 255],
        [255, 255, 255]]], dtype=uint8)
```

Información sobre la imagen

La imagen digital se puede representar como un array tridimensional donde cada píxel es un valor de 0 a 255, siendo 0 negro y 255 blanco.

```
# Mostrar información sobre la imagen
numpy_info(photo)
```

```
La dimensión es: 3
La forma es: (557, 640, 3)
El tamaño es: 1069440
```

Cada píxel en la imagen tiene un formato de color como sigue:

- `pixel`: cuadrado con color
- `color`: [r, g, b] (valores de rojo, verde y azul)
- Ejemplo: [0, 255, 89]

`Explorando la imagen numéricamente`

Podemos obtener las dimensiones de la imagen utilizando `photo.shape`, donde:

- `photo.shape[0]`: primera dimensión (altura, eje Y)
- `photo.shape[1]`: segunda dimensión (longitud, eje X)
- `photo.shape[2]`: tercera dimensión (profundidad del color, RGB)

```
# Dimensiones de la imagen
print(f'Primera dimensión: {photo.shape[0]}, Segunda dimensión:
{photo.shape[1]}, Tercera dimensión: {photo.shape[2]}')
```

```
Primera dimensión: 557, Segunda dimensión: 640, Tercera dimensión: 3
```

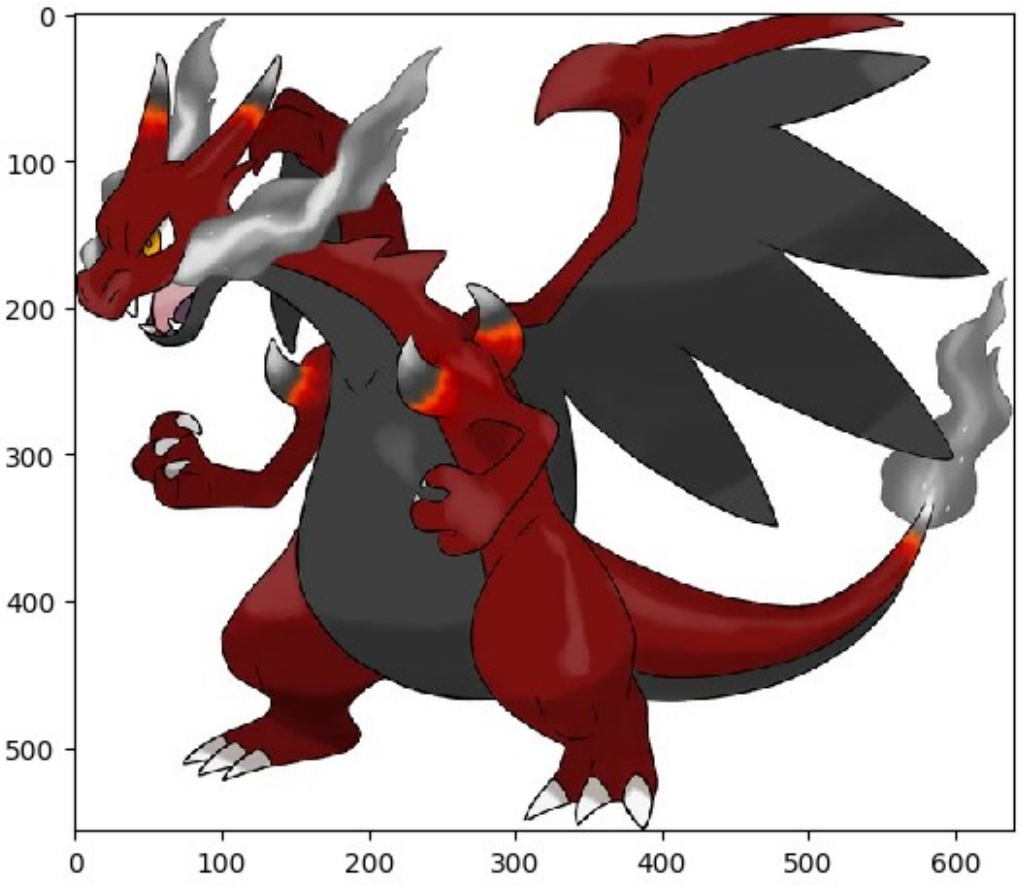
Gradación de Color

Podemos acceder a los valores de color de un píxel específico. Por ejemplo, el píxel en la posición (0, 0):

```
print("RED: ", photo[0][0][0])
print("GREEN: ", photo[0][0][1])
print("BLUE: ", photo[0][0][2])
```

```
RED:  255
GREEN:  255
BLUE:  255
```

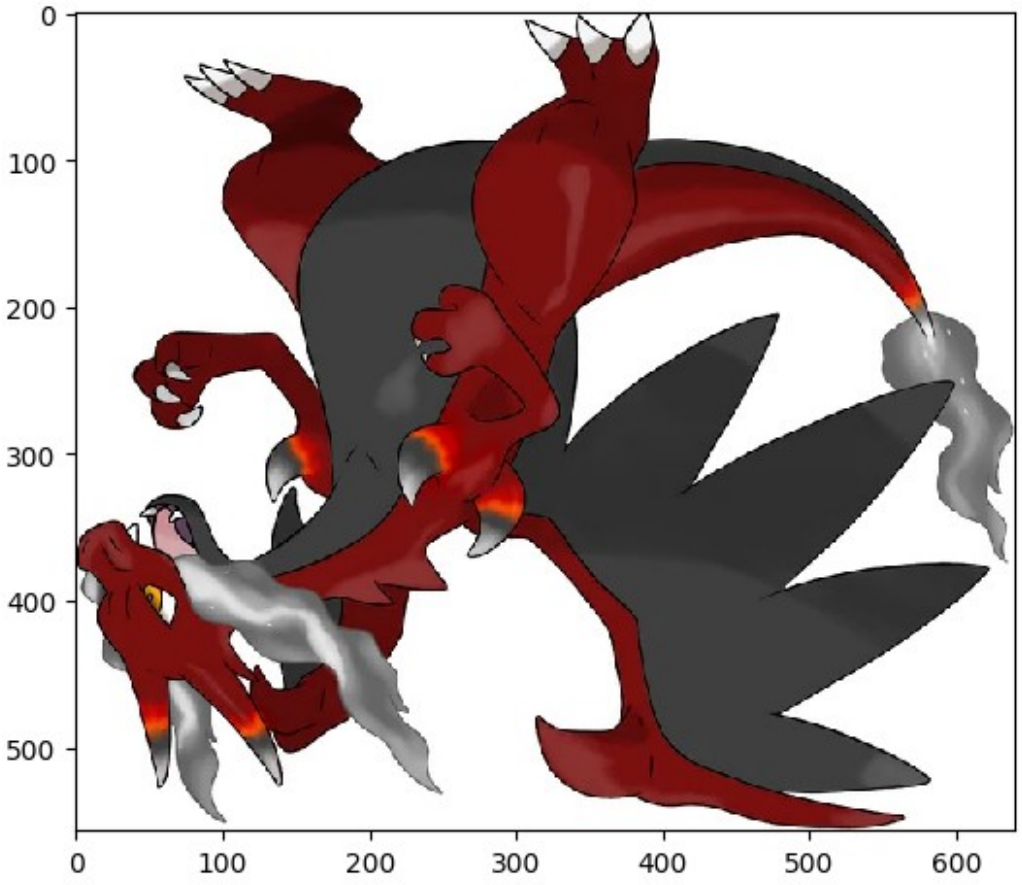
```
#Mostrar la imagen
pic = io.imshow(photo)
```



1. Invirtiendo una imagen

Recuerda que podemos invertir un array con `array[::-1]`. Intentémoslo:

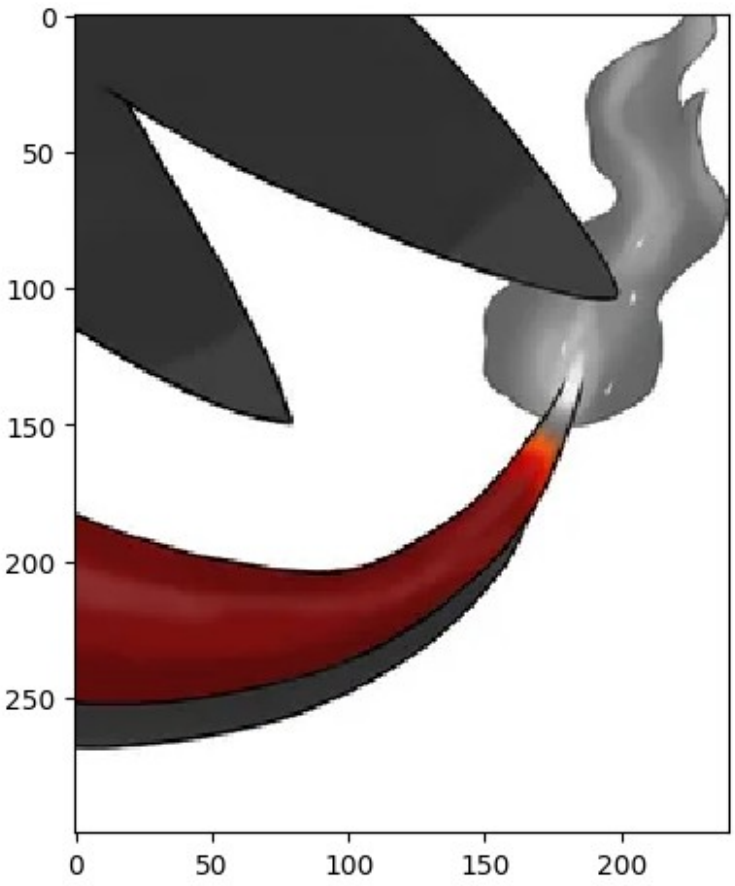
```
# Invertir la imagen
reversed_pic = io.imshow(photo[::-1])
```



2. Recortando una imagen

También podemos seleccionar una parte específica de la imagen:

```
# Recortar la imagen
cathedral = io.imshow(photo[200:500, 400:800])
```

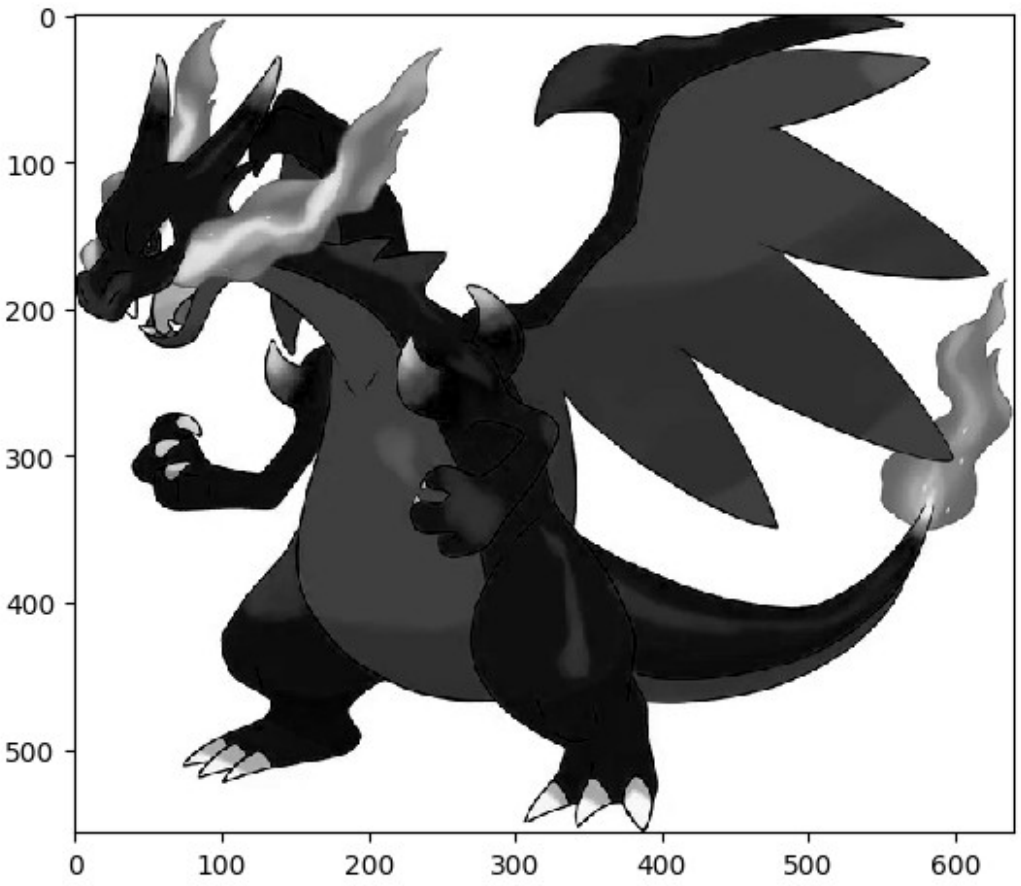


3. Cambiando el color

Para obtener solo el canal azul, podemos seleccionar todas las filas y columnas, y únicamente el canal azul:

```
# Mostrar solo el canal azul
io.imshow(photo[:, :, 2])
```

```
<matplotlib.image.AxesImage at 0x7f1f903020e0>
```



4. Usando `np.where`

La función `np.where` es útil para aplicar condiciones en arrays.

Aquí te mostramos cómo funciona:

- Si el valor es mayor que 100, ese valor se cambia a 255.
- Si no es mayor, el valor se establece en 0.

```
# Aplicando np.where
filter_2 = np.where(photo > 100, 255, 0)
filter_2
```

```
array([[[255, 255, 255],
        [255, 255, 255],
        [255, 255, 255],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]],

       [[255, 255, 255],
        [255, 255, 255],
```

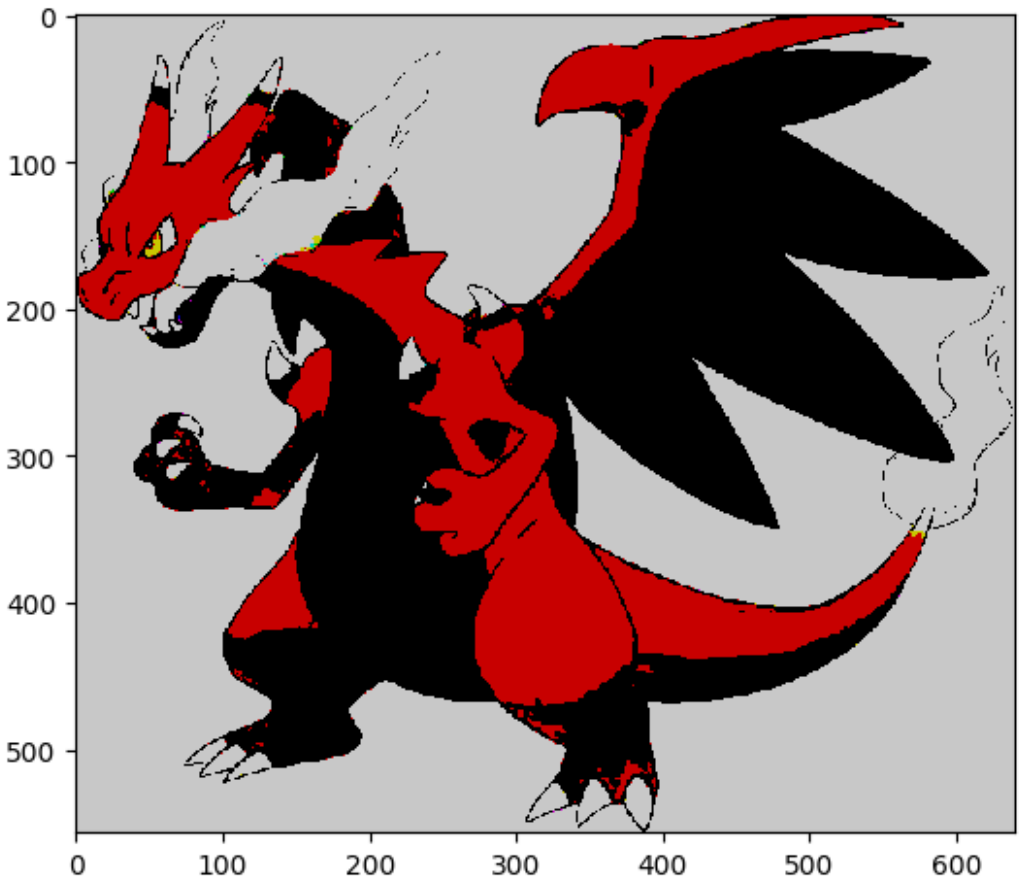
```
        [255, 255, 255],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]],

       [[255, 255, 255],
        [255, 255, 255],
        [255, 255, 255],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]],

       ...,

       [[255, 255, 255],
        [255, 255, 255],
        [255, 255, 255],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]],

       [[255, 255, 255],
        [255, 255, 255],
        [255, 255, 255],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]],

       [[255, 255, 255],
        [255, 255, 255],
        [255, 255, 255],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]]])
```

4.1. Brillo: Alto

Probemos con un valor de brillo alto de 200:

```
# Ajustar brillo alto
brightness_high = np.where(photo > 100, 200, 0).astype(np.uint8)
io.imshow(brightness_high)
```

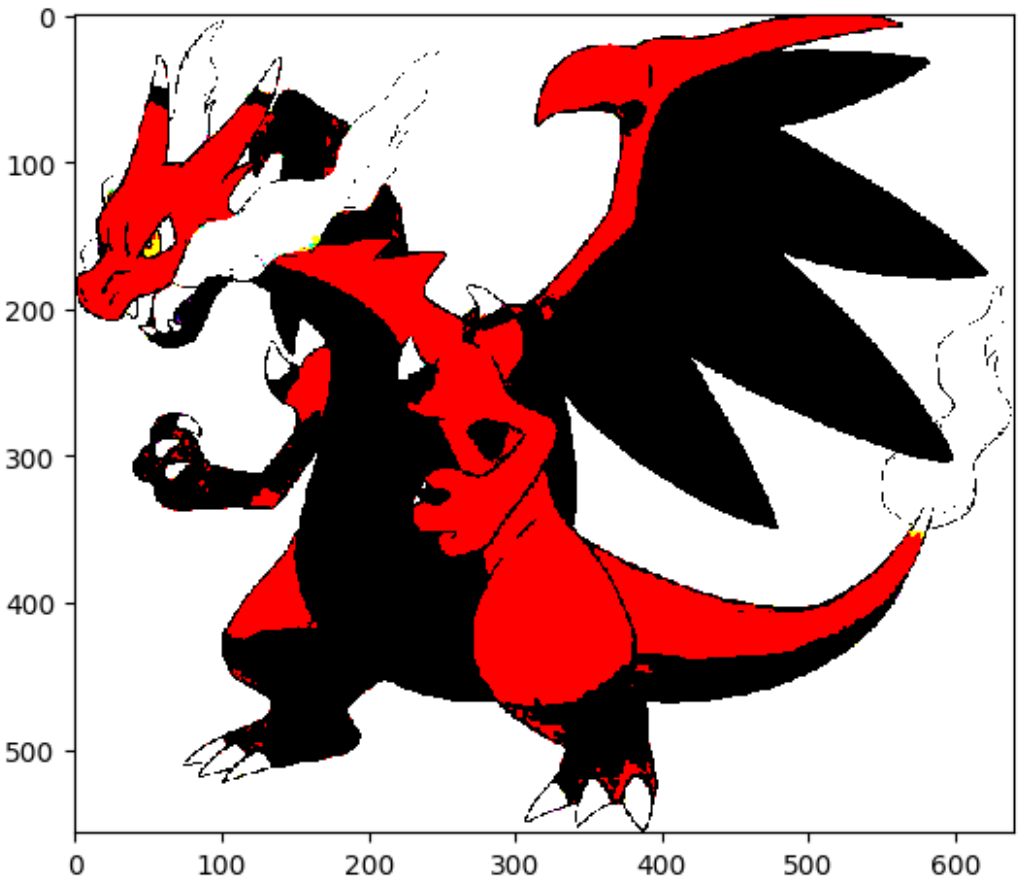
```
<matplotlib.image.AxesImage at 0x7f1f902e1000>
```



Probemos con un brillo de 255:

```
# Ajustar brillo a 255
brightness_high = np.where(photo > 100, 255, 0).astype(np.uint8)
io.imshow(brightness_high)
```

```
<matplotlib.image.AxesImage at 0x7f1f901f5780>
```

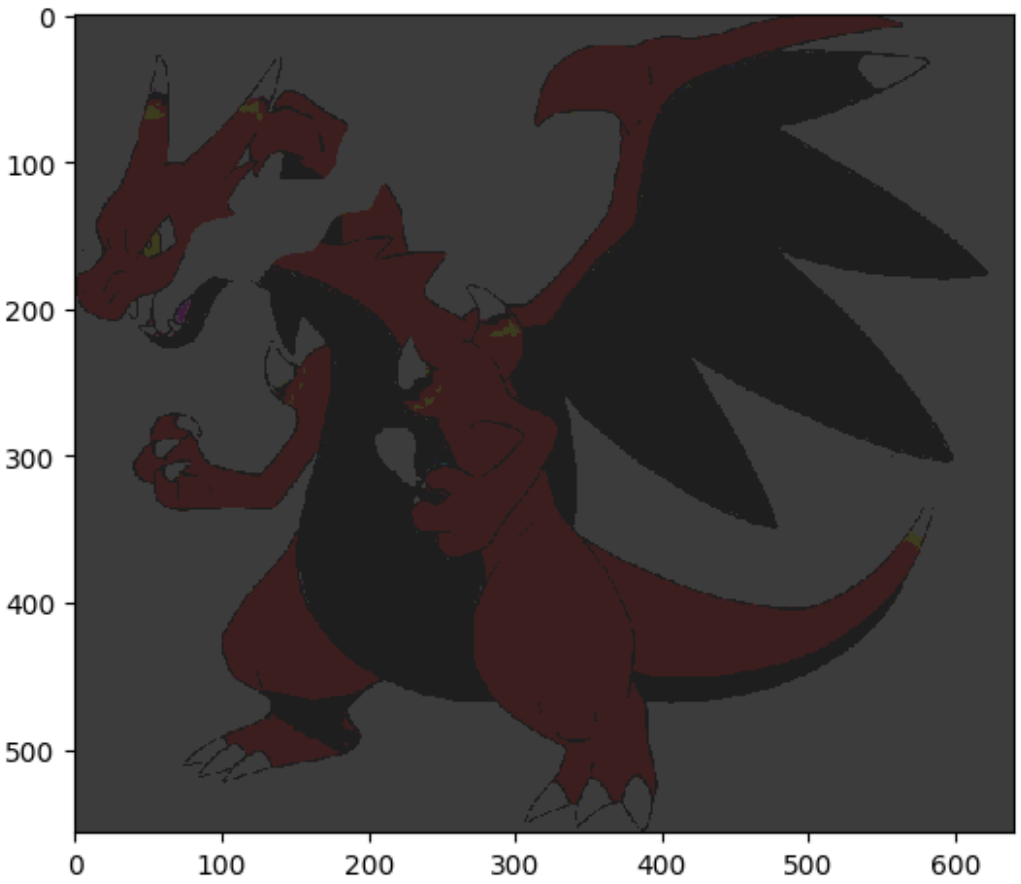


4.2. Brillo: Bajo

Probemos con un brillo bajo de 60:

```
# Ajustar brillo bajo
brightness_low = np.where(photo > 70, 60, 30).astype(np.uint8)
io.imshow(brightness_low)
```

```
<matplotlib.image.AxesImage at 0x7f1f9026afb0>
```



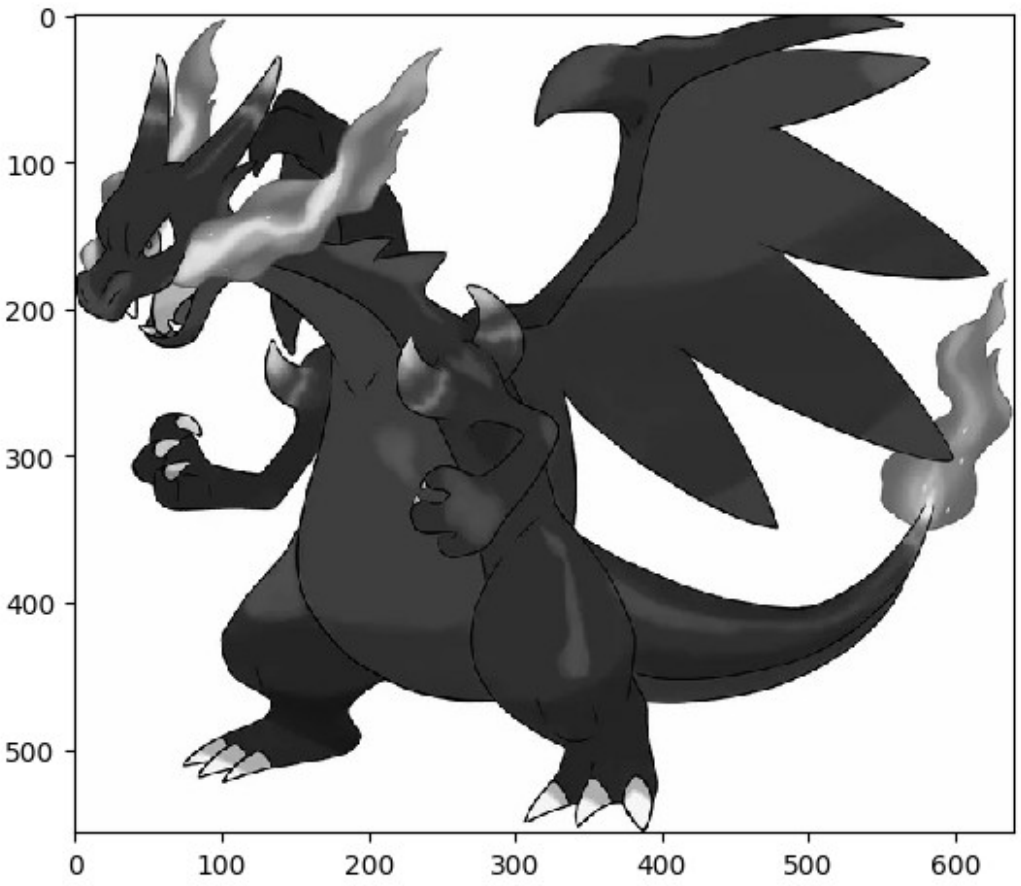
Una imagen digital se descompone en nuestro ordenador como un array tridimensional donde cada píxel es un valor de 0 a 255, siendo 0 negro y 255 blanco.

5. Convertir a Escala de Grises

Puedes convertir una imagen en color a escala de grises utilizando una fórmula que combina los valores RGB:

```
# Convertir a escala de grises
gray_photo = (0.2989 * photo[:, :, 0] + 0.5870 * photo[:, :, 1] +
0.1140 * photo[:, :, 2]).astype(np.uint8)
io.imshow(gray_photo, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7f1f900f4eb0>
```

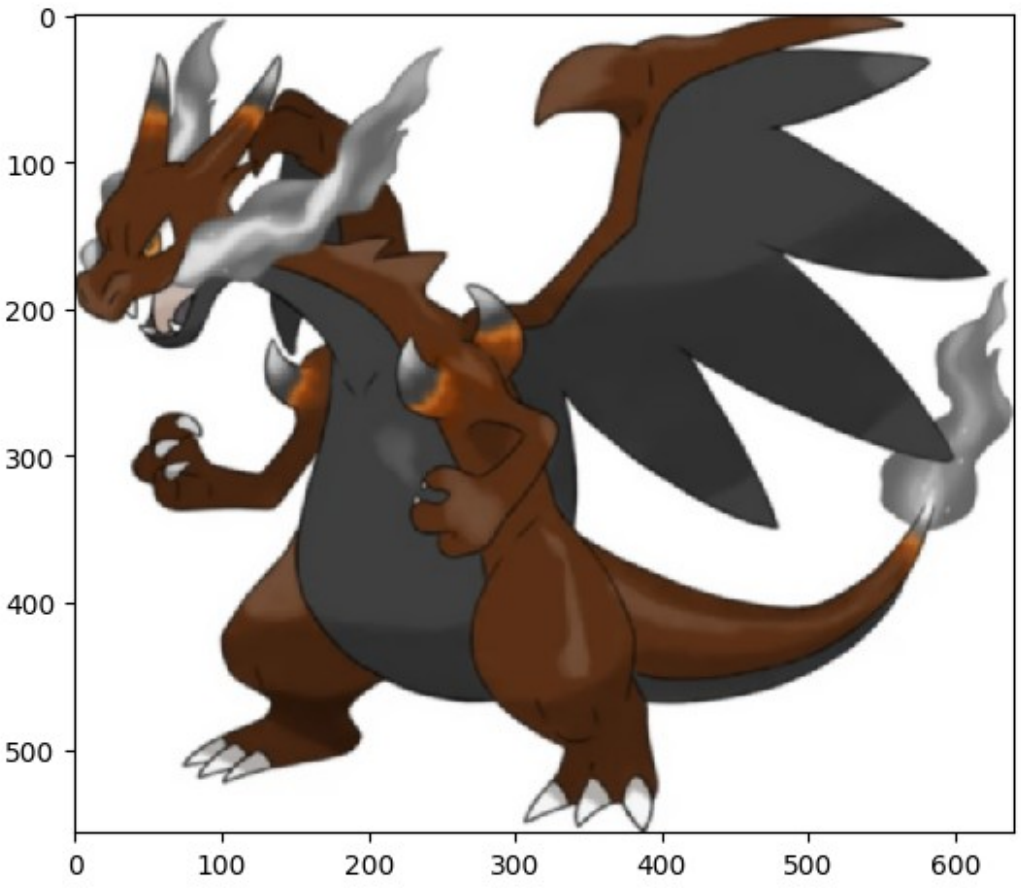


6. Aplicar un Filtro de Desenfoque

Puedes aplicar un filtro de desenfoque utilizando un kernel para suavizar la imagen:

```
# Importar filtros de scikit-image
from skimage.filters import gaussian

# Aplicar un filtro de desenfoque
blurred_photo = gaussian(photo, sigma=1)
io.imshow(blurred_photo)
```

```
<matplotlib.image.AxesImage at 0x7f1f8ada4340>
```

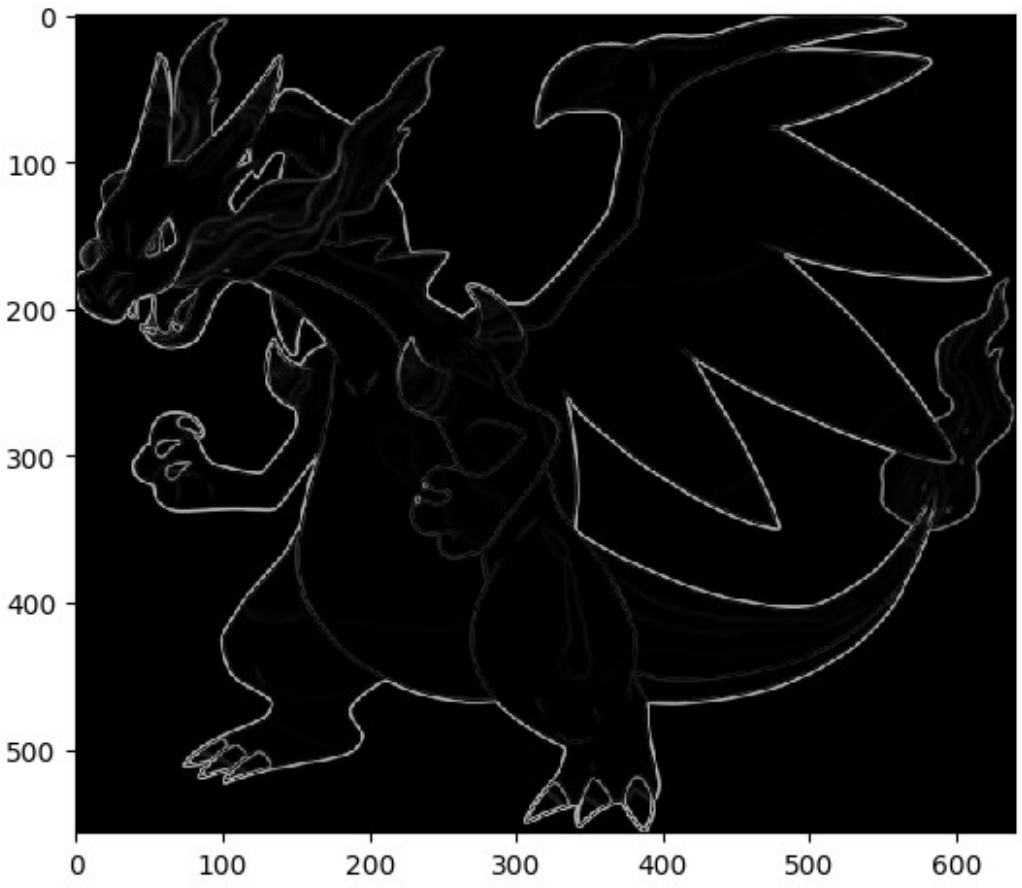


7. Detección de Bordes

Puedes usar filtros para detectar bordes en la imagen, como el filtro Sobel:

```
# Importar el filtro Sobel
from skimage.filters import sobel

# Aplicar detección de bordes
edges = sobel(gray_photo)
io.imshow(edges, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7f1f8abc0220>
```

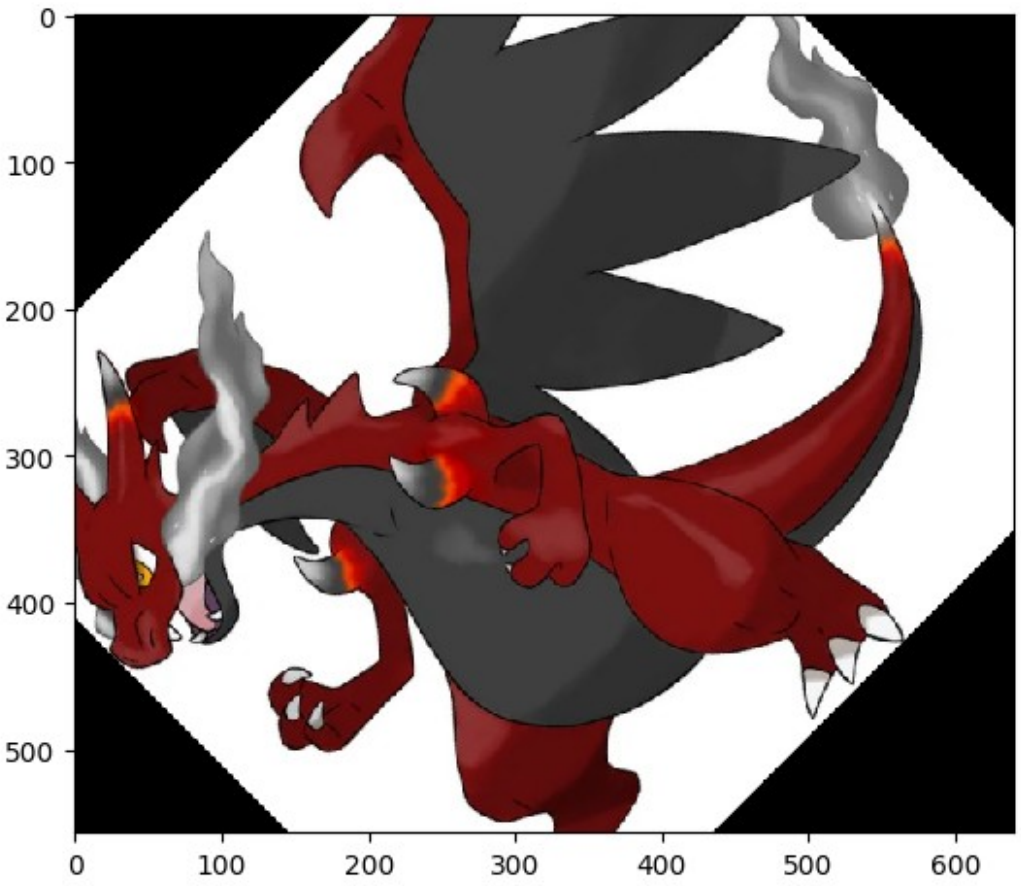


8. Rotar la Imagen

También puedes rotar la imagen en 90 grados o cualquier ángulo:

```
# Importar la función rotate
from skimage.transform import rotate

# Rotar la imagen 45 grados
rotated_photo = rotate(photo, angle=45)
io.imshow(rotated_photo)
```

```
<matplotlib.image.AxesImage at 0x7f1f8a62f6d0>
```

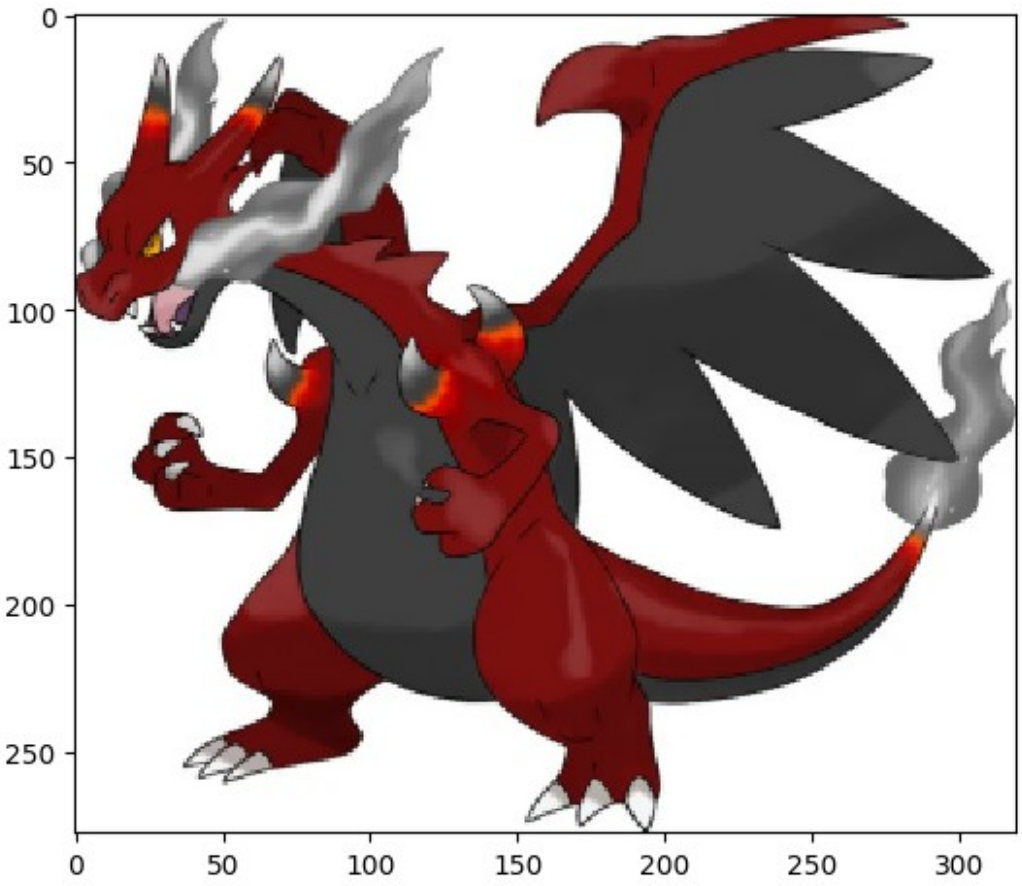


9. Escalar la Imagen

Puedes cambiar el tamaño de la imagen utilizando la función resize:

```
# Importar la función resize
from skimage.transform import resize

# Escalar la imagen a la mitad de su tamaño
resized_photo = resize(photo, (photo.shape[0] // 2, photo.shape[1] //
2))
io.imshow(resized_photo)
```

```
<matplotlib.image.AxesImage at 0x7f1f8a485360>
```



10. Guardar la Imagen Modificada

Finalmente, puedes guardar la imagen modificada en tu sistema:

```
# Importar la función imsave
from skimage.io import imsave

# Guardar la imagen en un nuevo archivo
imsave("img/photo_modificada.png", gray_photo)
```

Cada una de estas operaciones abre nuevas posibilidades para explorar y manipular imágenes utilizando Numpy y otras bibliotecas de Python.

Algunos métodos:

- Multiplicar
- Reformar
- Transponer

Material extra

- NumPy Cheatsheet

- Master Numpy
- Numpy Tricks
- 101 Numpy exercises