

Repasso

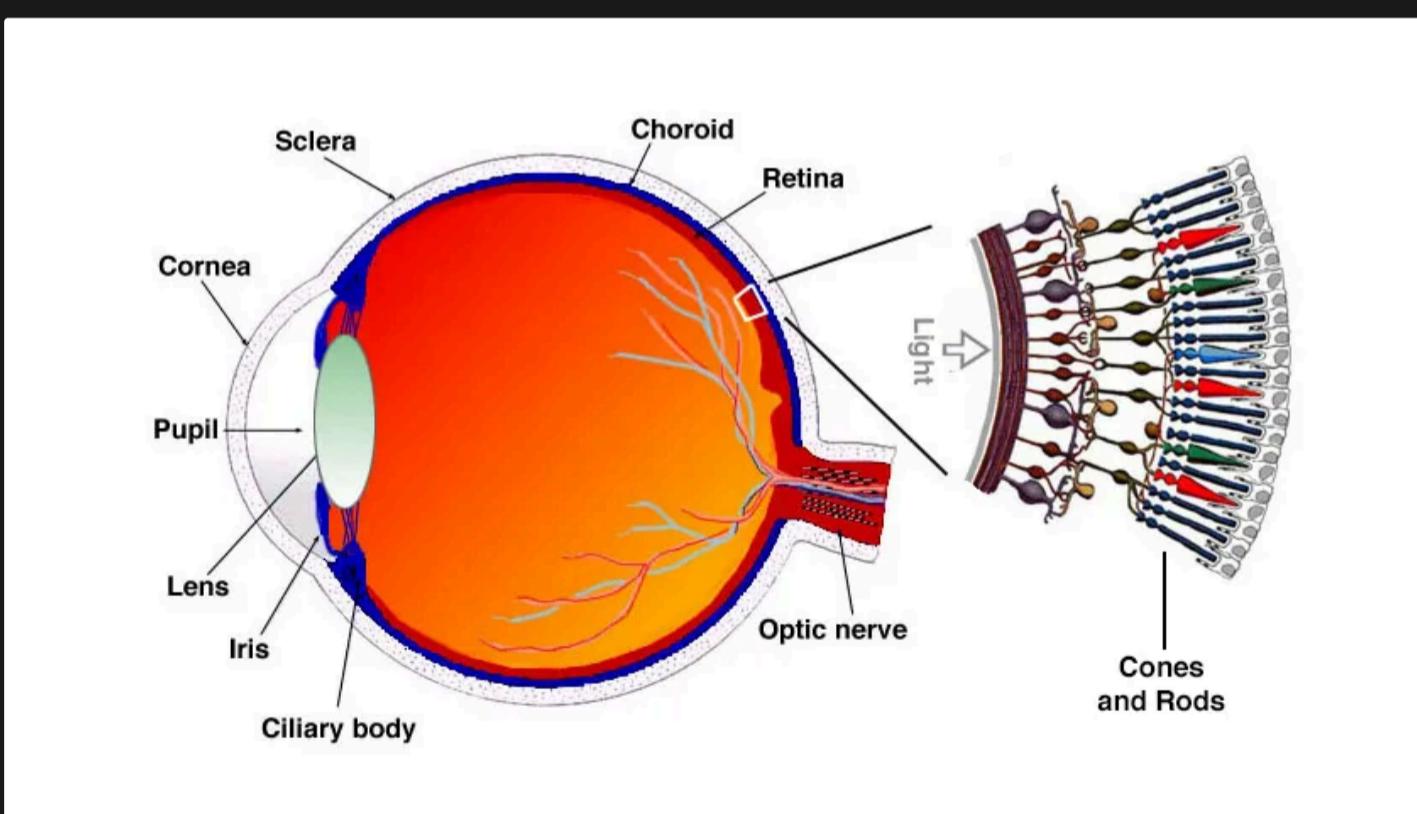
Representación digital de imágenes

Las pantallas y el RGB

Las tecnologías de pantalla modernas, incluyendo las de televisores, monitores de computadora, teléfonos inteligentes y tablets, se basan en el modelo de color RGB debido a cómo los seres humanos perciben el color. Este modelo utiliza los colores rojo (R), verde (G) y azul (B) como colores primarios de luz, que, al combinarse en diferentes intensidades, pueden crear una amplia gama de colores. Esta característica es fundamental en la forma en que las imágenes digitales son manipuladas y representadas en las computadoras y dispositivos electrónicos. A continuación veremos los aspectos más importantes de este proceso.

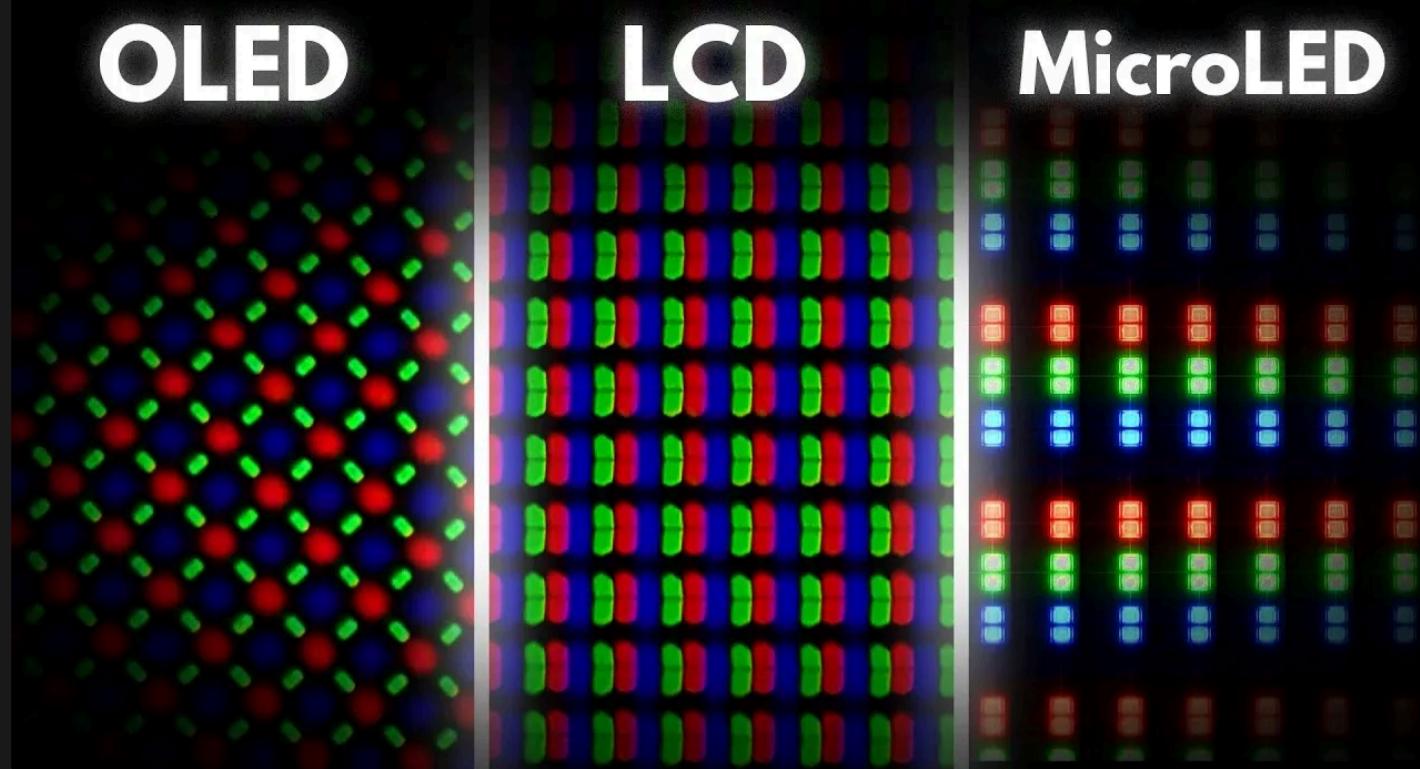
Percepción Humana del Color

El modelo RGB se basa en la teoría de la visión del color tricromática, que sugiere que el ojo humano tiene tres tipos de receptores de color (conos) sensibles a las longitudes de onda de la luz que corresponden aproximadamente a los colores rojo, verde y azul. Al estimular estos receptores en diferentes proporciones, podemos percibir una amplia gama de colores. Las tecnologías de pantalla explotan esta característica al usar píxeles compuestos por subpíxeles rojos, verdes y azules para simular el espectro completo de colores visibles.



Tecnología de Pantalla RGB

En las pantallas, cada píxel es en realidad un conjunto de tres subpíxeles: uno rojo, uno verde y uno azul. La intensidad de cada subpíxel se puede ajustar de forma independiente. Al variar la intensidad de estos subpíxeles, las pantallas pueden crear millones de colores diferentes. Por ejemplo, para producir el color blanco puro, los subpíxeles rojo, verde y azul se iluminan al máximo, mientras que para generar negro, todos se apagan. La mezcla de estos tres colores en proporciones específicas genera el espectro completo de colores que percibimos en una pantalla.



Colores primarios aditivos y sustractivos

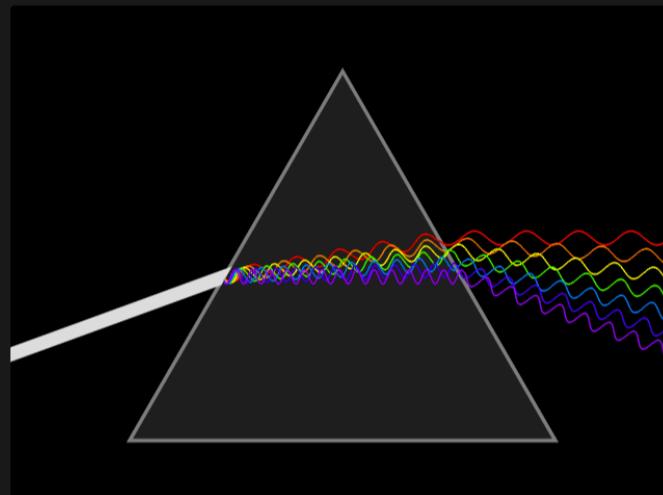
Los colores primarios se pueden clasificar en dos categorías principales según cómo se combinan para crear otros colores:

Colores Primarios Aditivos (RGB)

Los colores primarios aditivos son aquellos que se utilizan en dispositivos **que emiten luz**, como pantallas y proyectores. Se llaman aditivos porque al combinar diferentes cantidades de luz de estos colores, se suman para crear otros colores:

- Rojo + Verde = Amarillo
- Verde + Azul = Cian
- Azul + Rojo = Magenta
- Rojo + Verde + Azul = Blanco

La ausencia de estos colores produce negro.



Dispersión refractiva (Isaac Newton):
https://es.wikipedia.org/wiki/Dispersión_refractiva

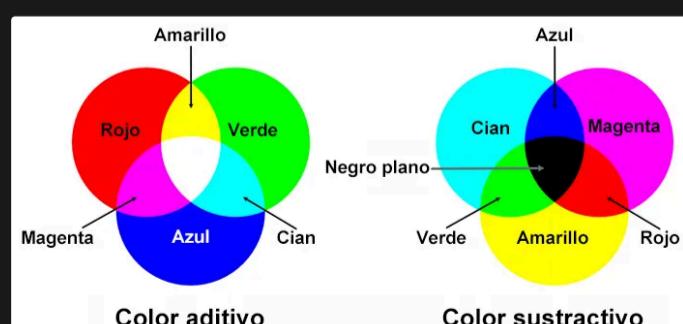
Colores Primarios Sustractivos (CMY/CMYK)

Los colores primarios sustractivos son aquellos que se utilizan en la impresión y en pigmentos físicos. Se llaman sustractivos porque cada pigmento sustrae (absorbe) ciertas longitudes de onda de la luz y refleja otras:

- Cian + Magenta = Azul
- Magenta + Amarillo = Rojo
- Amarillo + Cian = Verde

En la impresión se agrega el negro (K) para lograr una mejor definición y ahorro de tinta, formando el sistema CMYK. La ausencia de estos colores produce blanco (el color del papel), mientras que la combinación de todos ellos produce negro.

En resumen, en la siguiente figura podemos ver los colores primarios y secundarios de cada categoría "Aditiva" y "Sustractiva":



💡 ¿Y los colores primarios tradicionales: rojo, azul y amarillo?

Históricamente, en el arte y la enseñanza, se adoptaron el **rojo**, **azul** y **amarillo** como colores primarios sustractivos porque eran una simplificación práctica y funcional para los pintores y artistas desde hace siglos. Este modelo tiene raíces en la observación empírica más que en una ciencia precisa:

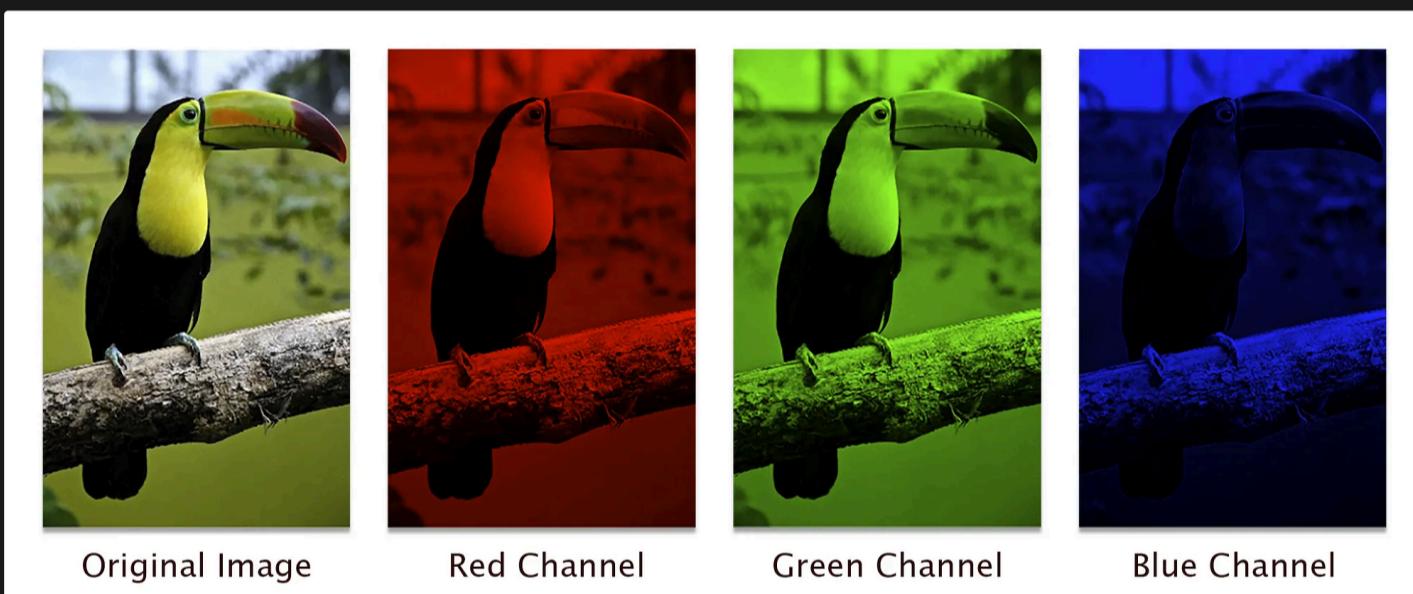
- Los artistas notaron que con estos tres colores podían mezclar una amplia variedad de tonos (aunque no todos).
- Era un sistema intuitivo: el rojo, el azul y el amarillo son colores "puros" y perceptualmente distintos para el ojo humano, lo que los hacía fáciles de enseñar y usar.
- Sin embargo, este modelo no es científicamente exacto ni óptimo para reproducir todos los colores posibles en la mezcla sustractiva.

CMY: el modelo sustractivo moderno

En el ámbito técnico y científico (como en la impresión moderna), los verdaderos colores primarios sustractivos son **cian**, **magenta** y **amarillo** (CMY). ¿Por qué? Porque estos colores son más eficientes y precisos para absorber longitudes de onda específicas de luz.

Manipulación Digital de Imágenes

Cuando manipulamos imágenes digitales en computadoras, esencialmente estamos ajustando los valores de los píxeles en términos de sus componentes RGB. Cada píxel de una imagen digital tiene un valor específico para rojo, verde y azul, representado comúnmente como un número de 0 a 255 (en el caso de imágenes de 8 bits por canal). La manipulación de estos valores a nivel de software permite cambiar los colores, la luminosidad, el contraste y otros atributos visuales de la imagen. En una imagen de 8 bits por 3 canales, podemos combinar un total de 16.777.216 colores ($2^{(8*3)} = 2^{24}$).



Aquí podemos interactivamente trabajar con la mezcla de colores aditivos:

RGB Color Slider Tool

<http://www.cknuckles.com/rbgsiders.html>

Importancia en el Diseño de Software y Hardware

El modelo RGB no solo influye en cómo se construyen y diseñan las pantallas, sino también en cómo se desarrollan los software de procesamiento de imágenes y los sistemas gráficos de las computadoras. Los formatos de archivo de imagen, las API gráficas y los algoritmos de procesamiento de imágenes están diseñados teniendo en cuenta el modelo RGB. Esto asegura que las imágenes se muestren de manera consistente a través de diferentes dispositivos y plataformas, y que las herramientas de edición de imágenes puedan manipular estos archivos de manera efectiva.

Imágenes como Numpy Arrays

Cuando trabajamos con imágenes en Python utilizando OpenCV, las imágenes se almacenan en memoria como arrays multidimensionales de Numpy. Este enfoque aprovecha la eficiencia y la flexibilidad de Numpy para manipular los datos de la imagen, permitiendo realizar una amplia variedad de operaciones de procesamiento de imágenes de manera eficiente.

Almacenamiento de Imágenes en Memoria

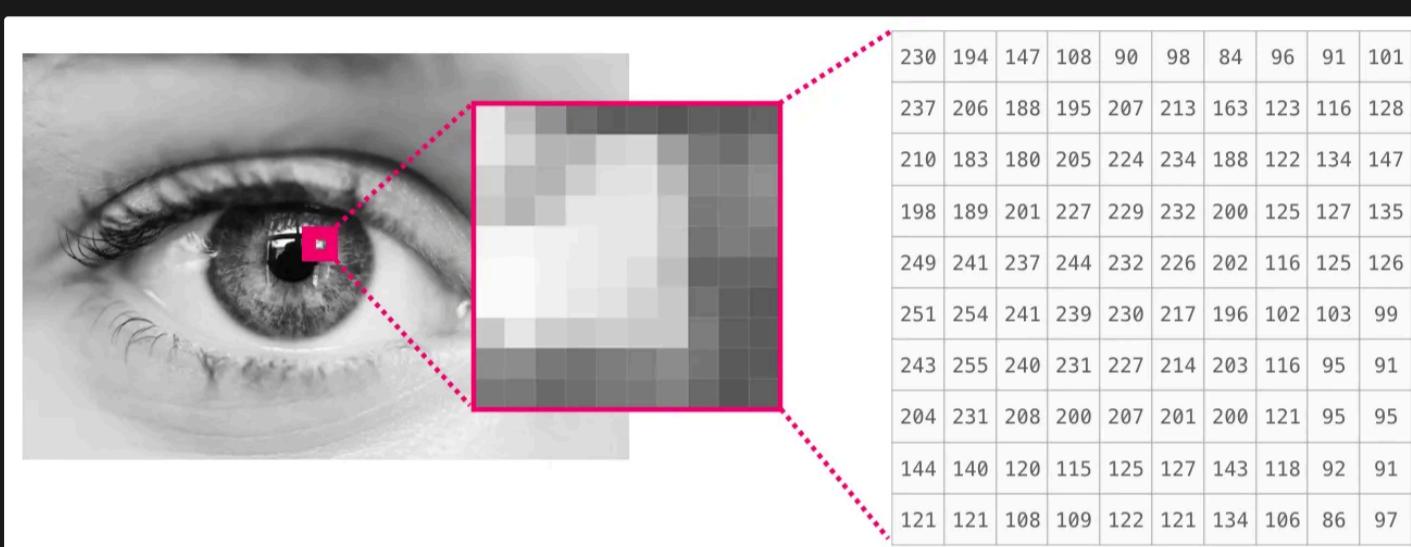
Una imagen digital puede considerarse como una matriz de píxeles, donde cada píxel representa el color y la intensidad en un punto específico de la imagen. En el caso de imágenes en escala de grises, cada píxel se representa típicamente por un único valor de intensidad (un solo canal), mientras que en imágenes a color, cada píxel se representa mediante una combinación de intensidades de diferentes canales de color, comúnmente rojo, verde y azul (RGB).

OpenCV, por defecto, lee imágenes en el formato BGR (azul, verde, rojo) en lugar del convencional RGB. Esto es solamente un cambio en el orden de los canales, diferente al convencional. Significa que cada píxel se representa por un conjunto de tres valores, indicando la intensidad de azul, verde y rojo respectivamente.

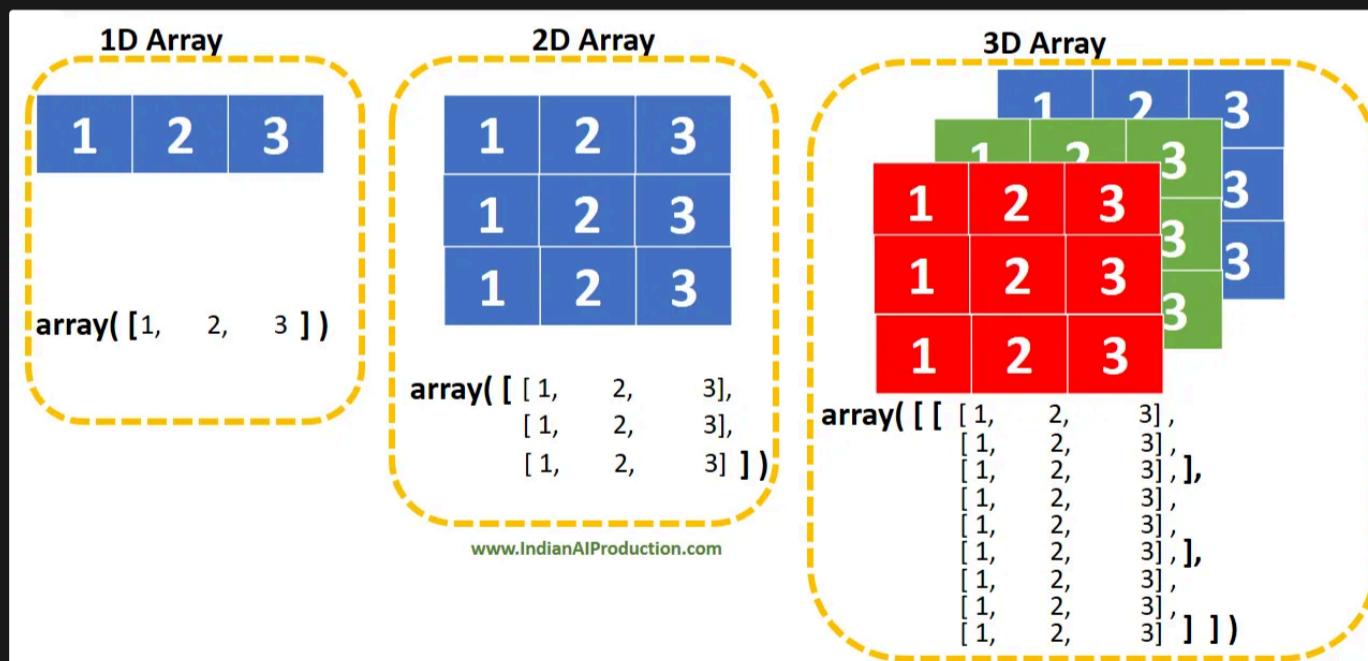
Representación con Numpy

Numpy es una librería de Python que proporciona un objeto array multidimensional, el cual es una colección de elementos (normalmente números), todos del mismo tipo, indexados por una tupla de enteros positivos. En el contexto de las imágenes:

- **Imágenes en Escala de Grises:** Se representan como un array 2D de Numpy, donde la forma (`shape`) del array es `(altura, anchura)`. Cada elemento del array representa la intensidad de un píxel en la imagen. En este caso tenemos solamente un canal o plano de color.



- **Imágenes a Color:** Se representan como un array 3D de Numpy, donde la forma del array es `(altura, anchura, canales)`. Los "canales" se refieren al número de componentes de color por píxel, que en el caso de BGR o RGB es 3. Cada píxel se representa entonces por un subarray de 3 elementos.



Tipo de Datos (`dtype`)

El tipo de datos (`dtype`) de un array de Numpy que representa una imagen generalmente es `numpy.uint8`, indicando que cada valor de píxel es un entero sin signo de 8 bits (rango de 0 a 255). Sin embargo, dependiendo de la operación o el procesamiento aplicado a la imagen, otros tipos de datos pueden ser utilizados para representar diferentes rangos o precisión (como `float32` o `int16`).

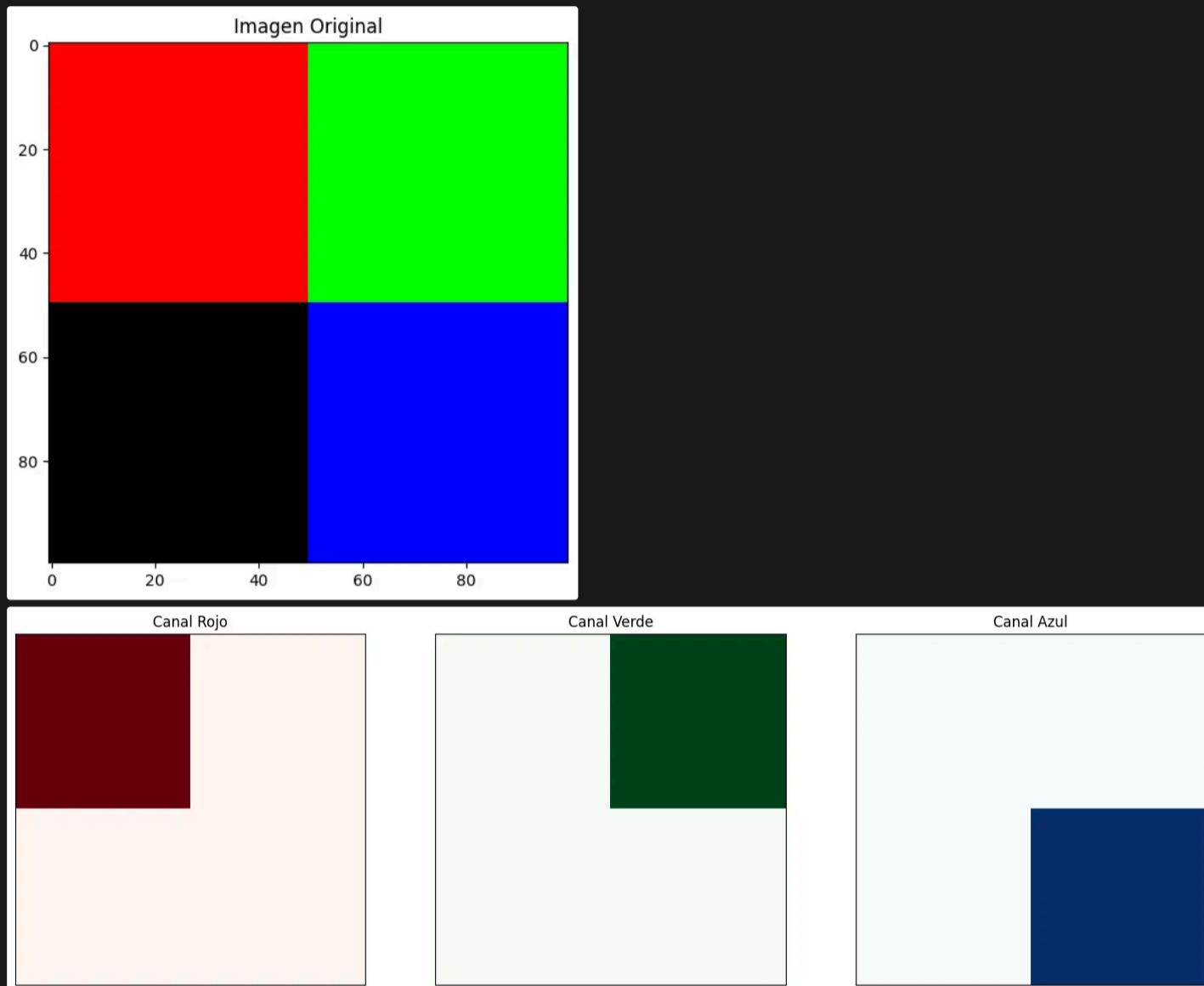
Una imagen se puede crear desde código, creando un array Numpy. Veamos un ejemplo:

```

import numpy as np import matplotlib.pyplot as plt # Paso 1: Crear una imagen de ejemplo
# Generar una imagen de 100x100 pixeles, cada píxel tiene 3 valores (R, G, B) # Rojo puro
para la mitad izquierda, verde puro para la mitad superior derecha, # y azul puro para la
mitad inferior derecha image = np.zeros((100, 100, 3), dtype=np.uint8) image[:50, :50] =
[255, 0, 0] # Rojo image[:50, 50:] = [0, 255, 0] # Verde image[50:, 50:] = [0, 0, 255] # Azul # Mostrar la imagen completa plt.figure(figsize=(6,6)) plt.imshow(image)
plt.title("Imagen Original") plt.show() # Paso 2: Numpy representa esta imagen en un
array. Su forma la podemos ver con `shape` print(f"Shape de la imagen: {image.shape}") # Paso 3: Separar la imagen en los canales de color R, G, B R, G, B = image[:, :, 0],
image[:, :, 1], image[:, :, 2] # Mostrar cada canal de color fig, axs = plt.subplots(1, 3,
figsize=(18, 6)) axs[0].imshow(R, cmap='Reds') axs[0].set_title('Canal Rojo')
axs[1].imshow(G, cmap='Greens') axs[1].set_title('Canal Verde') axs[2].imshow(B,
cmap='Blues') axs[2].set_title('Canal Azul') for ax in axs: ax.set_xticks([])
ax.set_yticks([]) plt.show()

```

La salida será:



Estas expresiones son equivalentes.

```
image[:50, :50] = [255, 0, 0] # Rojo image[:50, :50, :] = [255, 0, 0] # Rojo
```

La diferencia está en que:

- En la primera forma (`image[:50, :50]`), cuando omites el último índice en un array 3D, NumPy asume que quieres asignar valores a todas las posiciones en la tercera dimensión. Es una forma abreviada.
- En la segunda forma (`image[:50, :50, :]`), estás siendo explícito al usar `:` para indicar que quieres seleccionar todos los elementos de la tercera dimensión.

NumPy hace broadcasting automáticamente del valor `[255, 0, 0]` a todos los píxeles seleccionados en ambos casos. Desde el punto de vista del resultado, no hay diferencia - los primeros 50×50 píxeles de la imagen serán rojos en ambos casos. La forma abreviada es comúnmente utilizada por ser más concisa, pero la forma explícita puede ser más clara para entender exactamente qué está sucediendo, especialmente para quienes están aprendiendo.

Veamos ahora un ejemplo de cómo cargar una imagen desde una URL, y ver cada uno de sus canales:

```

import cv2 import numpy as np import matplotlib.pyplot as plt import requests from io
import BytesIO # Función auxiliar para leer una imagen desde una URL con OpenCV def
read_image_from_url(url): response = requests.get(url) # Convertir la respuesta a un
array de bytes. Obtenemos el JPG en memoria. image_bytes =
np.array(bytarray(response.content), dtype=np.uint8) # Decodificar los bytes codificados
como JPG en una imagen numpy image = cv2.imdecode(image_bytes, cv2.IMREAD_COLOR) return
image # URL de la imagen que deseas descargar image_url =
"https://raw.githubusercontent.com/jpmanson/tuia-unr/main/images/big_buck_bunny.jpg" #
Leer la imagen image = read_image_from_url(image_url) # OpenCV maneja imágenes en formato
BGR, convertir a RGB image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB) # Mostramos la
información de la imagen print(f"Shape de la imagen: {image_rgb.shape}.")
print(f"Columnas (Ancho): {image_rgb.shape[1]}") print(f"Filas (Altura): {image_rgb.shape[0]}")
print(f"Canales: {image_rgb.shape[2]}") print(f"Tipo de dato del array: {image_rgb.dtype}")
# Descomponer la imagen en los canales R, G, B R, G, B =
image_rgb[:, :, 0], image_rgb[:, :, 1], image_rgb[:, :, 2] # Mostrar cada canal de color fig,
axs = plt.subplots(1, 3, figsize=(15, 5)) axs[0].imshow(R, cmap='Reds')
axs[0].set_title('Canal Rojo') axs[0].axis('off') axs[1].imshow(G, cmap='Greens')
axs[1].set_title('Canal Verde') axs[1].axis('off') axs[2].imshow(B, cmap='Blues')
axs[2].set_title('Canal Azul') axs[2].axis('off') plt.show()

```

La salida será como la siguiente:

Shape de la imagen: (1080, 1920, 3). Columnas (Ancho): 1920 Filas (Altura): 1080 Canales: 3 Tipo de dato del array: uint8



En el ejemplo anterior, hemos convertido la imagen cargada en memoria originalmente con una disposición BGR (Azul, Verde, Rojo) a una disposición RGB (Rojo, Verde y Azul), usando la línea:

image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

Es habitual este tipo de conversiones cuando trabajamos con imágenes, y es importante saber siempre que forma (shape), tipo de dato y la disposición de los canales de colores tiene la imagen con la cual estamos trabajando.

Librería Pillow

Otra forma de cargar una imagen en memoria, es a través de la librería [pillow](#) (). Veamos un ejemplo:

```

import requests from PIL import Image import numpy as np import matplotlib.pyplot as plt
from io import BytesIO # URL de la imagen que deseas descargar image_url =
"https://raw.githubusercontent.com/jpmanson/tuia-unr/main/images/big_buck_bunny.jpg" #
Paso 2: Descargar la imagen response = requests.get(image_url) # Descarga de la URL image
= Image.open(BytesIO(response.content)) # Aquí se decodifica de JPG a imagen utilizable #
Paso 3: Convertir la imagen en un array de Numpy image_np = np.array(image) print(f"Shape
de la imagen: {image_np.shape}.") print(f"Columnas (Ancho): {image_np.shape[1]}")
print(f"Filas (Altura): {image_np.shape[0]}") print(f"Canales: {image_np.shape[2]}")
print(f"Tipo de dato del array: {image_np.dtype}") # Paso 4: Descomponer la imagen en los
canales R, G, B R, G, B = image_np[:, :, 0], image_np[:, :, 1], image_np[:, :, 2] # Paso 5:
Mostrar cada canal de color fig, axs = plt.subplots(1, 3, figsize=(15, 5))
axs[0].imshow(R, cmap='Reds') axs[0].set_title('Canal Rojo') axs[0].axis('off')
axs[1].imshow(G, cmap='Greens') axs[1].set_title('Canal Verde') axs[1].axis('off')
axs[2].imshow(B, cmap='Blues') axs[2].set_title('Canal Azul') axs[2].axis('off')
plt.show()

```

Escalas de grises

Convertir imágenes a escala de grises es una práctica común en el procesamiento de imágenes y la visión por computadora, especialmente cuando se trabaja con OpenCV, por varias razones clave:

Simplificación de la Información

Las imágenes en color contienen tres canales (R, G, B o B, G, R en el caso de OpenCV), lo que significa que cualquier operación de procesamiento de imágenes debe tener en cuenta tres veces la cantidad de datos en comparación con una imagen en escala de grises, que solo tiene un canal. Convertir una imagen a escala de grises reduce la complejidad de los datos, facilitando y acelerando el procesamiento.

Reducción del Procesamiento Computacional

Dado que las imágenes en escala de grises contienen menos datos (un solo canal en lugar de tres), el procesamiento de estas imágenes requiere menos recursos computacionales. Esto es particularmente importante para aplicaciones en tiempo real, como el seguimiento de objetos o el reconocimiento facial en video, donde el tiempo de procesamiento es crítico.

Mejora en el Análisis de Imágenes

Muchas técnicas de análisis y procesamiento de imágenes, como la detección de bordes, el seguimiento de características, y el reconocimiento de patrones, no requieren información de color para ser efectivas. En estos casos, trabajar con imágenes en escala de grises puede simplificar los algoritmos sin comprometer su eficacia. Además, algunas operaciones, como el cálculo de gradientes y la binarización, son más sencillas y directas en imágenes en escala de grises.

Conservación de la Luminancia

La conversión a escala de grises conserva la luminancia (o brillo) de la imagen mientras descarta la información de color. Esto es útil en aplicaciones donde el color no aporta información significativa, y el contraste entre las regiones claras y oscuras de la imagen es más relevante.

Base para Otras Transformaciones

En muchos flujos de trabajo de visión por computadora, convertir una imagen a escala de grises es un paso previo antes de aplicar otras transformaciones o análisis, como la umbralización (thresholding), la detección de bordes, o la extracción de características. Esto se debe a que operar en un solo canal puede hacer que estas técnicas sean más efectivas y menos propensas a ser influenciadas por las variaciones de color en la imagen original.

Eficiencia en el Almacenamiento y la Transmisión

Las imágenes en escala de grises ocupan menos espacio de almacenamiento y pueden transmitirse más rápidamente que las imágenes en color, lo cual es ventajoso en situaciones donde el ancho de banda o el espacio de almacenamiento son limitados.

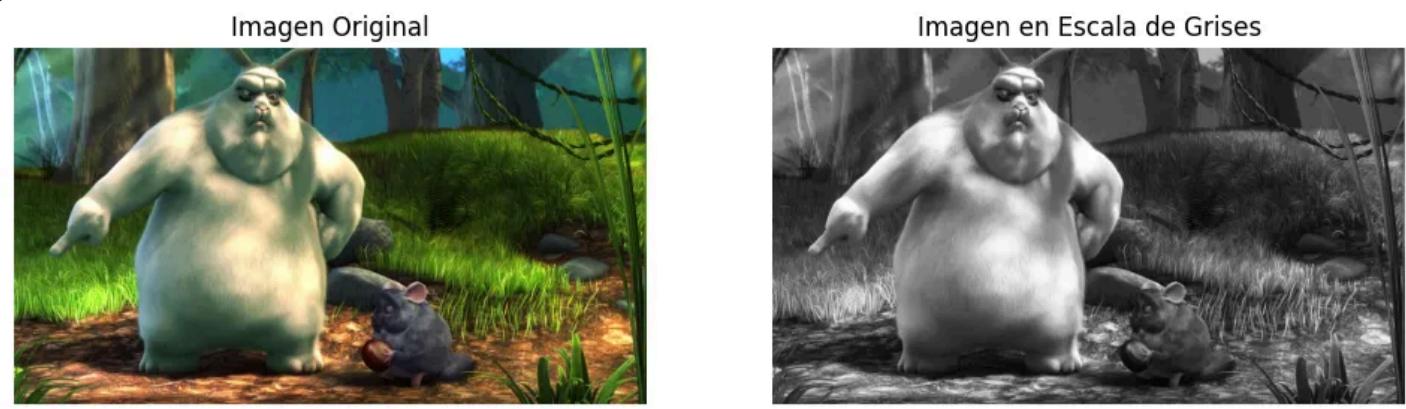
Con la función `cvtColor`, podemos convertir imágenes entre diferentes espacios de color, incluida la conversión de imágenes en color a escala de grises.

Veamos como hacerlo:

```
import cv2 import numpy as np import matplotlib.pyplot as plt import requests from io
import BytesIO # Función para leer una imagen desde una URL def read_image_from_url(url):
    response = requests.get(url) image = np.array(bytarray(response.content),
    dtype=np.uint8) image = cv2.imdecode(image, cv2.IMREAD_COLOR) # Convertir bytes a imagen
    return image # URL de la imagen que vamos a convertir a escala de grises image_url =
"https://raw.githubusercontent.com/jpmanson/tuia-unr/main/images/big_buck_bunny.jpg" #
    Leer la imagen image_color = read_image_from_url(image_url) # Convertir de BGR a RGB para
    mostrar correctamente con Matplotlib image_rgb = cv2.cvtColor(image_color,
    cv2.COLOR_BGR2RGB) # Convertir la imagen a escala de grises image_gray =
cv2.cvtColor(image_color, cv2.COLOR_BGR2GRAY) # Imprimir el shape y el tipo de dato del
    array de la imagen original y en escala de grises print(f"Shape de la imagen original
(RGB): {image_rgb.shape}") print(f"Tipo de dato del array de la imagen original:
{image_rgb.dtype}") print(f"Shape de la imagen en escala de grises: {image_gray.shape}") #
    print(f"Tipo de dato del array de la imagen en escala de grises: {image_gray.dtype}") #
    Mostrar ambas imágenes, original y en escala de grises plt.figure(figsize=(12, 6)) #
    Imagen original plt.subplot(1, 2, 1) plt.imshow(image_rgb) plt.title("Imagen Original")
    plt.axis('off') # Imagen en escala de grises plt.subplot(1, 2, 2) plt.imshow(image_gray,
    cmap='gray') plt.title("Imagen en Escala de Grises") plt.axis('off') plt.show()
```

Y la salida será:

```
Shape de la imagen original (RGB): (1080, 1920, 3) Tipo de dato del array de la imagen
original: uint8 Shape de la imagen en escala de grises: (1080, 1920) Tipo de dato del
array de la imagen en escala de grises: uint8
```



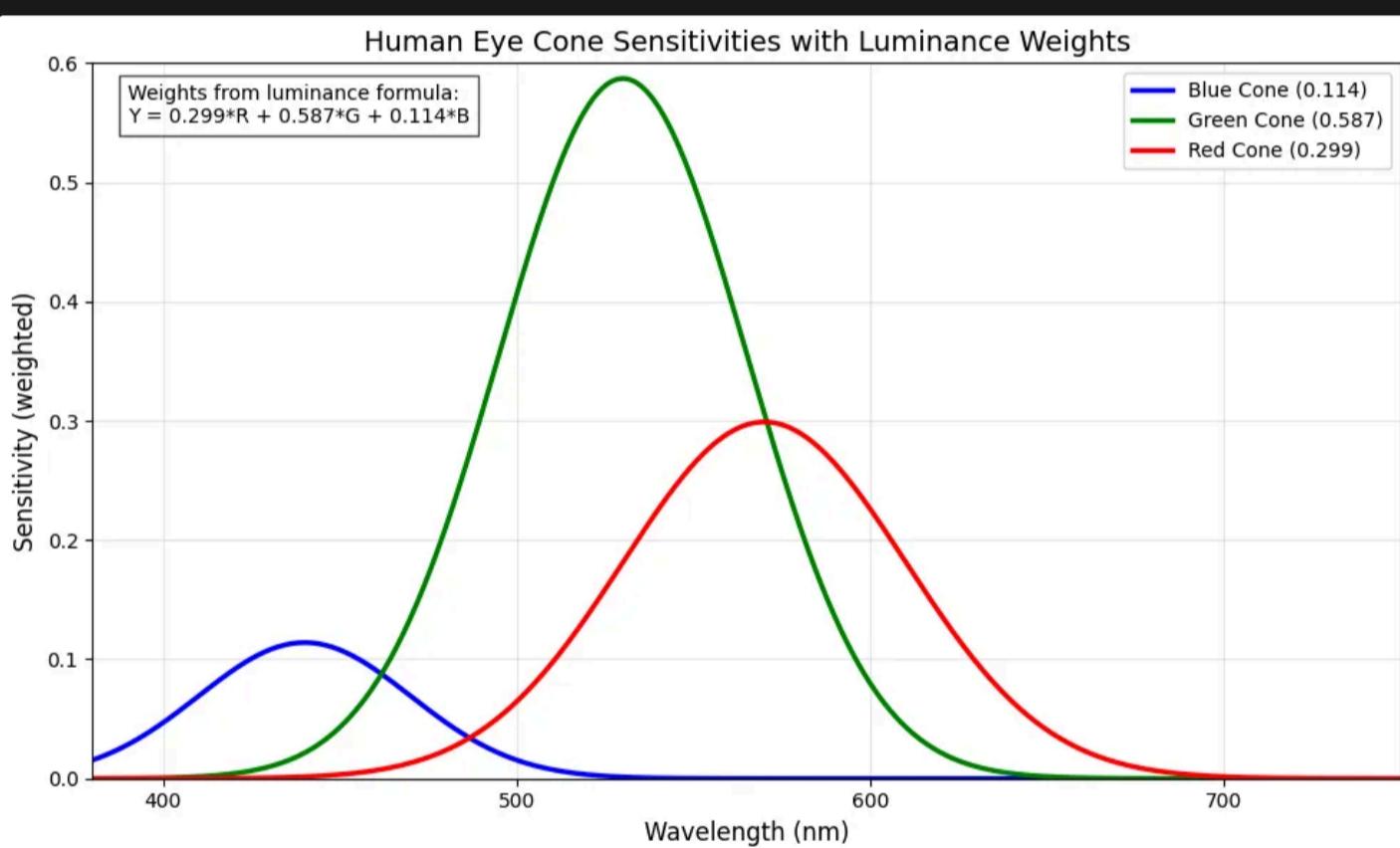
La fórmula que OpenCV utiliza para convertir una imagen a color en escala de grises es una ponderación específica de los canales de color RGB (Rojo, Verde, Azul). Esta fórmula está basada en la percepción del ojo humano de la luminosidad de diferentes colores. Aquí está la fórmula:

$$Y = 0.299 * R + 0.587 * G + 0.114 * B$$

Donde:

- Y es el valor resultante en escala de grises
- R es el valor del canal rojo
- G es el valor del canal verde
- B es el valor del canal azul

Esta ponderación se realiza para que la imagen en escala de grises resultante, refleje cómo el ojo humano percibe el brillo relativo de diferentes colores. Los coeficientes (0.299, 0.587, 0.114) representan la sensibilidad relativa del ojo humano a cada color. El ojo humano es más sensible al verde, luego al rojo, y menos sensible al azul, lo que se refleja en los pesos asignados.



En el siguiente ejemplo, vemos como realizar esta conversión de canales, realizando cálculos directamente con Numpy:

```
import numpy as np import matplotlib.pyplot as plt import requests import cv2 def
read_image_from_url(url): response = requests.get(url) image =
np.array(bytarray(response.content), dtype=np.uint8) image = cv2.imdecode(image,
cv2.IMREAD_COLOR) # Convertir de BGR a RGB (OpenCV usa BGR por defecto) image =
cv2.cvtColor(image, cv2.COLOR_BGR2RGB) return image def rgb_to_gray(rgb_image): #
Asegurarse de que la imagen esté en formato float rgb_image =
rgb_image.astype(np.float32) # Aplicar la fórmula de conversión gray_image =
np.dot(rgb_image[..., :3], [0.299, 0.587, 0.114]) # Asegurarse de que los valores estén
en el rango [0, 255] gray_image = np.clip(gray_image, 0, 255) # Convertir de vuelta a
uint8 return gray_image.astype(np.uint8) # URL de la imagen que vamos a convertir a
escala de grises image_url = "https://raw.githubusercontent.com/jpmanson/tuia-
unr/main/images/big_buck_bunny.jpg" # Leer la imagen rgb_image =
read_image_from_url(image_url) # Convertir a escala de grises gray_image =
rgb_to_gray(rgb_image) # Visualizar los resultados fig, (ax1, ax2) = plt.subplots(1, 2,
figsize=(15, 7)) ax1.imshow(rgb_image) ax1.set_title('Imagen RGB Original')
ax1.axis('off') ax2.imshow(gray_image, cmap='gray') ax2.set_title('Imagen en Escala de
Grises') ax2.axis('off') plt.tight_layout() plt.show()
```

La función `rgb_to_gray()` toma una imagen RGB como entrada y luego realiza las siguientes operaciones:

- Convierte la imagen a tipo float32 para realizar operaciones de punto flotante.

- Utiliza `np.dot()` (producto escalar de vectores) para aplicar la fórmula de conversión $(0.299R + 0.587G + 0.114B)$ a cada píxel. `rgb_image[..., :3]` representa nuestra imagen RGB. El `...` mantiene todas las dimensiones existentes (altura y anchura), y `:3` selecciona los tres canales de color. `[0.299, 0.587, 0.114]` es nuestro vector de pesos para la conversión a escala de grises. `np.dot()` multiplica cada pixel (que es un vector de 3 elementos [R, G, B]) por el vector de pesos, efectivamente calculando:
$$0.299R + 0.587G + 0.114*B$$
 para cada pixel de la imagen. Esto da como resultado un único valor escalar para cada pixel, que es precisamente lo que queremos para una imagen en escala de grises.
- Limita los valores resultantes al rango [0, 255] con `np.clip()`.
- Convierte la imagen resultante de vuelta a `uint8` para compatibilidad con la mayoría de las funciones de visualización de imágenes.

| Link a ejemplos en Colab

Google Colab

 <https://colab.research.google.com/drive/1xa35-nCPbyJU0hAxK1CEWTaMnB...>

