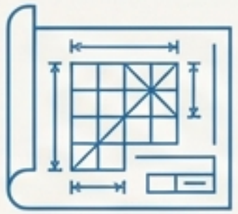
OpenCV Core: Deconstructing the Digital Image

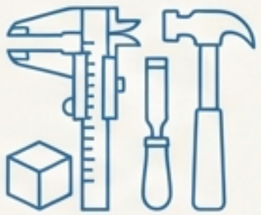
A foundational guide to image representation and manipulation.

Our Journey: From Pixel to Mastery



The Blueprint - Images as Data

Understanding the digital representation of images: NumPy arrays, coordinate systems, and color order.



The Toolkit - Forging the Tools

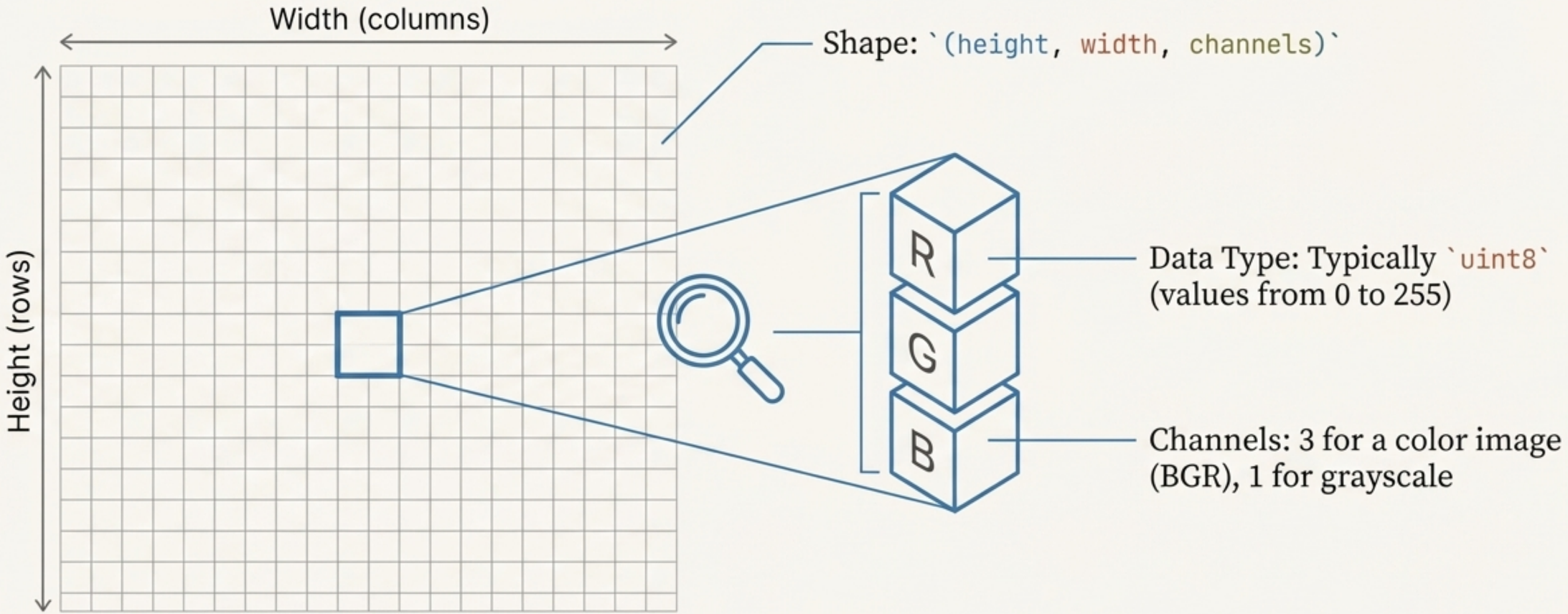
Introducing the primary operations for manipulation: Arithmetic, Bitwise Logic, and Channel Splitting.

The Craft - Precision & Application

Applying the tools with skill: defining Regions of Interest and handling image borders with precision.

The Blueprint: An Image is a Grid of Numbers

In Python, OpenCV represents images as NumPy arrays.
This structure is the key to all manipulation.



Navigating the Grid: The Coordinate System

Accessing pixels uses a `[row, col]` indexing system, which corresponds to `[y, x]` coordinates. The origin `(0,0)` is at the **top-left corner**.

x (columns) →

y (rows) →									
(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	(7,0)	(8,0)	(9,0)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(6,1)	(7,1)	(8,1)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)	(8,1)	(9,1)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(6,1)	(7,1)	(8,1)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)	(8,1)	(9,1)

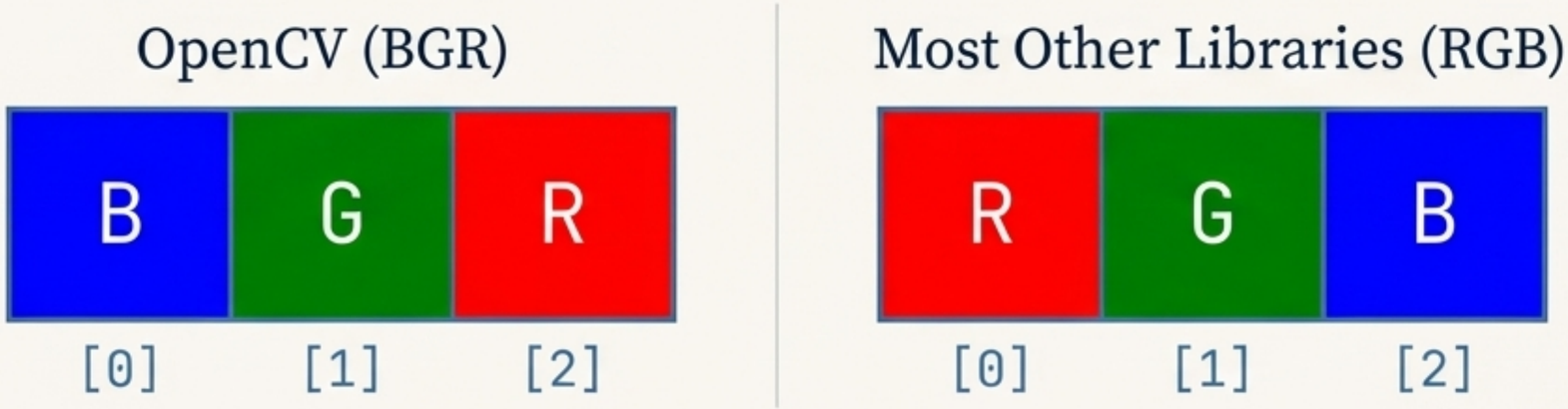
```
pixel_value = image[row, col]
```

Key Detail

Remember: It's `image[y, x]`. Rows (height) come before columns (width). This is a common source of bugs!

A Crucial Distinction: OpenCV's BGR Color Order

Unlike most image processing libraries that use RGB (Red, Green, Blue), OpenCV stores color channels in BGR (Blue, Green, Red) order by default.



Example: A pure red pixel

In OpenCV:`[0, 0, 255]`

In Matplotlib/PIL:`[255, 0, 0]`

The Alchemist’s Tools: Modifying Pixels with Arithmetic

Image arithmetic allows you to combine or alter images. OpenCV’s functions are designed to handle pixel values (0-255) correctly.

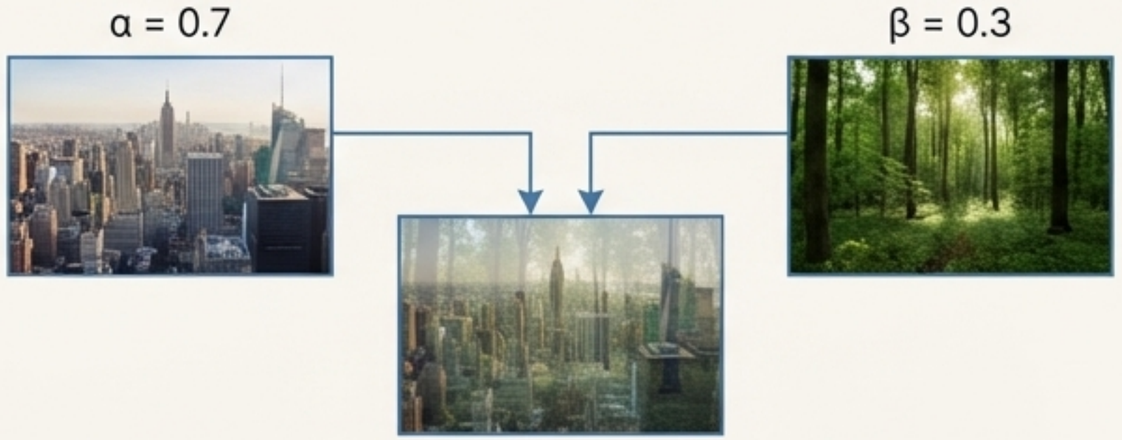
Addition for Brightening

Adding a constant value to every pixel increases the overall brightness of an image.

```
cv2.add(image, value_matrix)
```

Weighted Addition for Blending (Alpha Blending)

Combine two images with different weights to create a transparent overlay or fade effect.




```
dst = α * src1 + β * src2 + γ
```

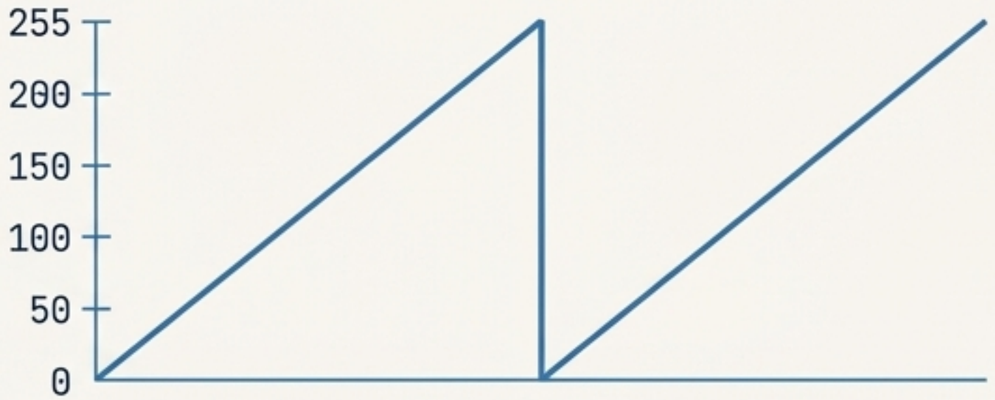
Critical Concept: Saturation vs. Wrap-Around

What happens when a pixel value goes above 255?

NumPy Addition (Wrap-around)

Values ‘wrap around’ using modulo arithmetic.

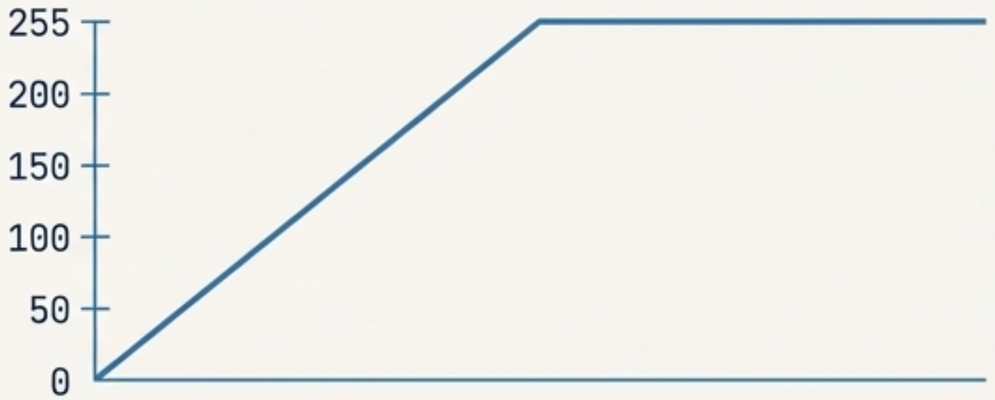
`200 + 100 = 300`, then `300 % 256 = 44`.
A bright pixel incorrectly becomes dark.



OpenCV Addition (Saturation)

Values are ‘clamped’ or ‘saturated’ at the maximum.

`200 + 100 = 300`, then `min(300, 255) = 255`.
A very bright pixel correctly stays very bright.



Always use `cv2.add()` and `cv2.subtract()` for image arithmetic to prevent visual artifacts.

The Sculptor’s Chisel: Bitwise Operations

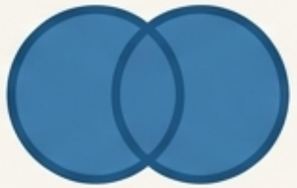
Bitwise operations act on the binary representation of each pixel’s value. They are essential for masking, selecting, and combining specific image regions.

AND

A	B	OUT
0	0	0
0	1	0
1	0	0
1	1	1

Masking / Intersection

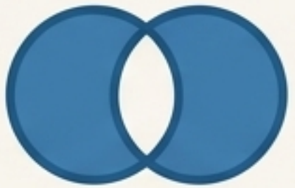
OR



A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	1

Combining / Union

XOR



A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	0

Finding Differences

NOT

IN	OUT
0	1
1	0

Inverting

Practical Application: Masking with `bitwise_and`

The `AND` operation is the standard way to apply a mask. Where the mask is white (value 255), the original image content is kept. Where the mask is black (value 0), the output is black.


```
masked_image = cv2.bitwise_and(original_image, original_image, mask=mask)
```

Working with Color: Splitting & Merging Channels

A color image can be deconstructed into its separate B, G, and R channels. These can be modified individually and then recombined.

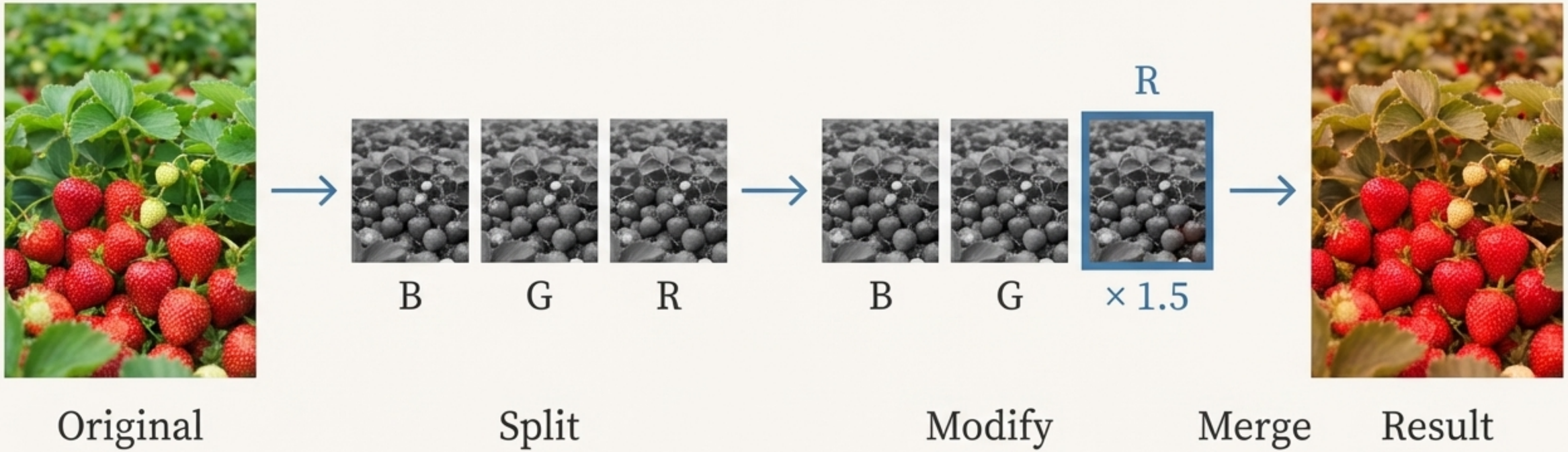
Blue Channel


```
b, g, r = cv2.split(image)
```

```
image = cv2.merge([b, g, r])
```

Example: Amplifying the Red Channel

By splitting channels, we can perform arithmetic on just one channel before merging them back together to alter the image's color balance.



Mastering the Craft: Defining a Region of Interest (ROI)

An ROI is a rectangular portion of an image selected for a specific operation. In NumPy, this is achieved through simple array slicing.


```
roi = image[y1:y2, x1:x2]
```

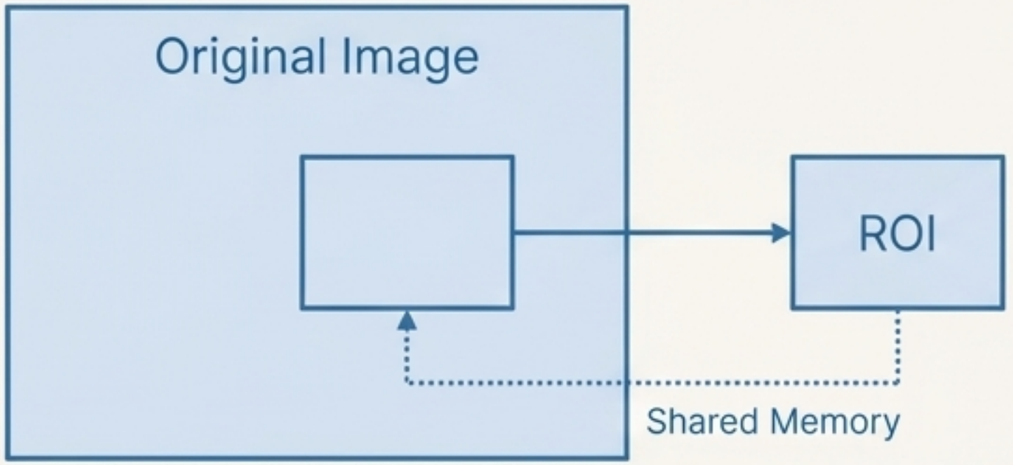
Gotcha
Notice the Order!

Slicing is [rows, columns], so the format is [y_start:y_end, x_start:x_end].

Critical Detail: An ROI is a View, Not a Copy

By default, slicing a NumPy array creates a “view” that shares memory with the original. To work independently, you must create an explicit copy.

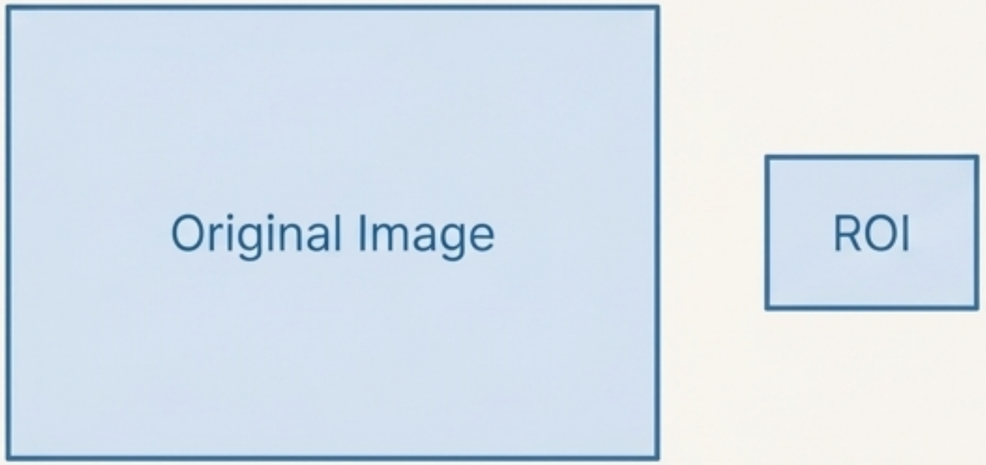
View (Default)




```
roi = image[y1:y2, x1:x2]
```

Modifying `roi` **changes** the original `image`.

Copy (Explicit)

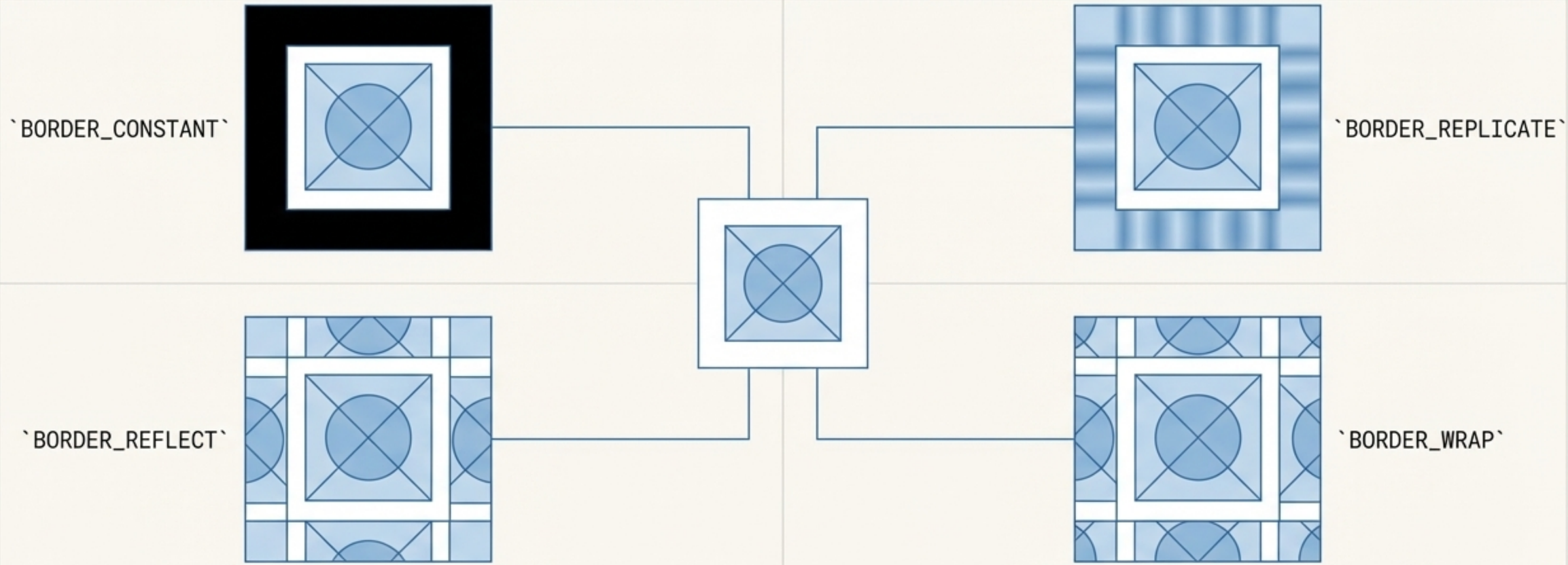



```
roi = image[y1:y2, x1:x2].copy()
```

Modifying `roi` **does not change** the original `image`.

Handling the Edges: Border Padding Techniques

When operations need to access pixels beyond the image boundary, we must define how to "invent" those pixels. OpenCV provides several intelligent methods for this.




```
bordered_image = cv2.copyMakeBorder(src, top, bottom, left, right, borderType)
```

Your Core OpenCV Toolkit: A Reference Guide

Image Creation & Access

`np.zeros((h,w,c), ...)`	Create black image.
`image[row, col]`	Access pixel at (y, x).
`image[y1:y2, x1:x2]`	Select Region of Interest (ROI).

cv2.add()	Masking and waege otition.
Saturating addition (for brightness).	Blend two images with transparency.

Arithmetic Operations

cv2.add()	Saturating addition (for brightness).
cv2.addWeighted()	Blend two images with transparency.

Bitwise & Logical Operations

cv2.bitwise_and()	Masking and finding intersection.
cv2.bitwise_or() cv2.bitwise_xor() cv2.bitwise_not()	Other logical operations (union, difference, inversion).

Channel & Border Operations

cv2.split() / cv2.merge()	Manipulate B, G, R color channels.
cv2.copyMakeBorder()	Add padding to image edges.