



LAB 3: WORKING WITH IMAGE DATA

University of Washington
ECE 241

Author: Jimin Kim (jk55@uw.edu)
Version: v1.7.0

OUTLINE

Part 1: Image formats

- How are images represented in a computer
- Grayscale image
- Color image
- Reading and Writing images in Python
- Converting Color to Grayscale

Part 2: Basic Operations on images: Grayscale

- Analyzing an image with pixel histogram
- Manipulating image pixels
- Image flipping
- Image down-sampling
- Image blending

Part 3: Basic Operations on images: Color

- Indexing 3D image arrays
- Constructing 3D image from 2D arrays
- Expanding image operations to color

Part 4: Lab Assignments

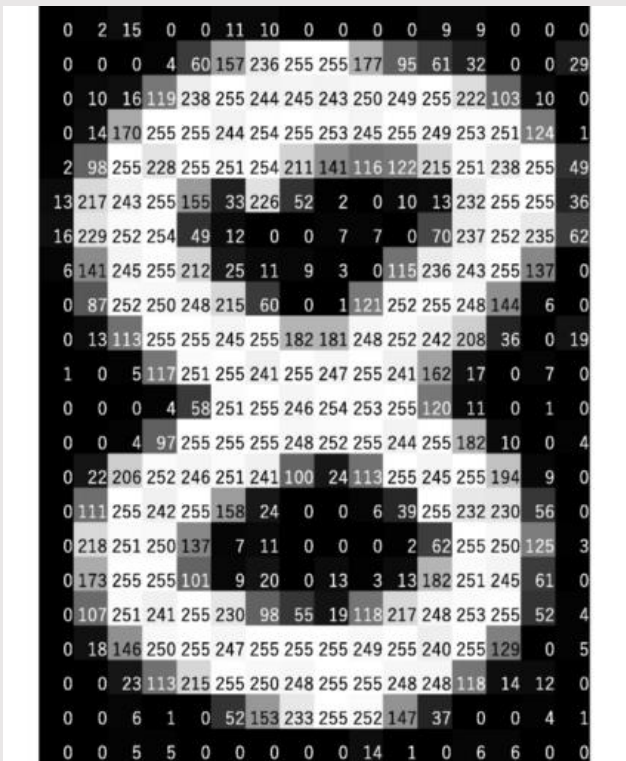
- Exercise 1 – 5

IMAGE FORMATS

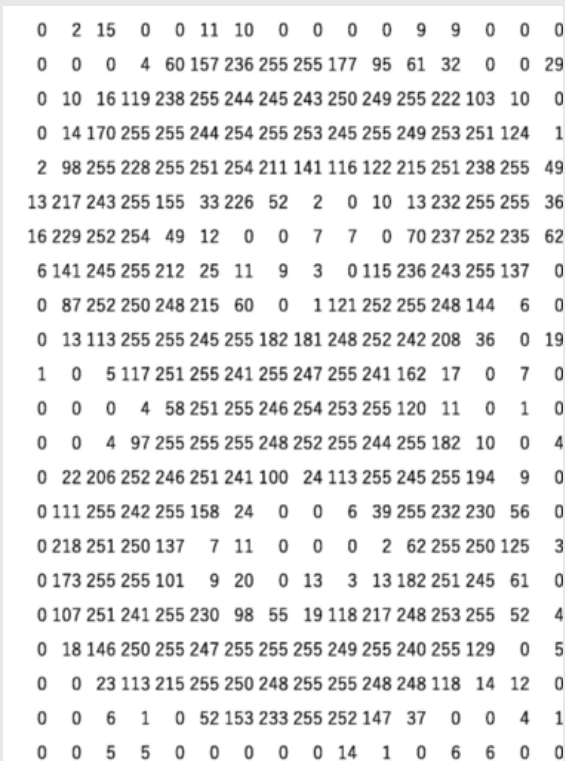
HOW ARE IMAGES REPRESENTED IN A COMPUTER



Original Image



Pixels as Numbers

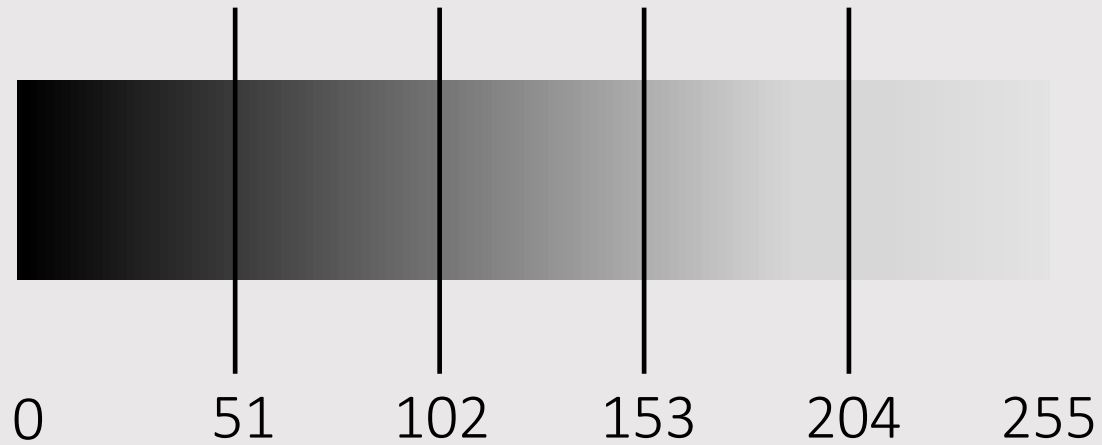


Array form of the image

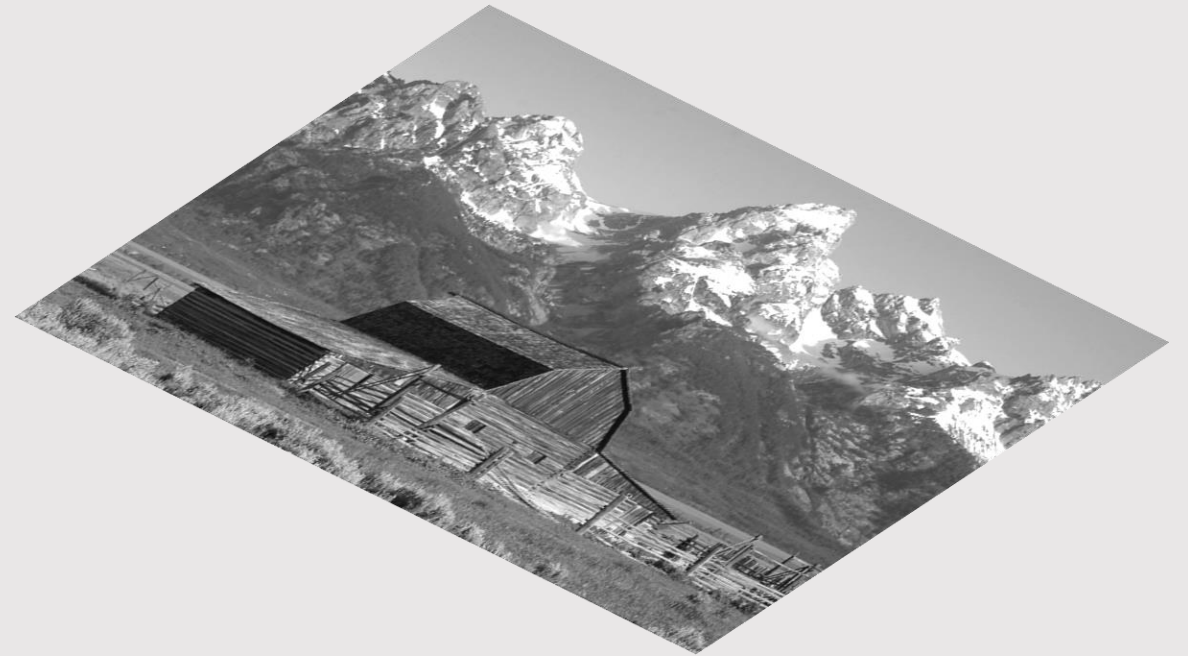
Image = Set of 2D array(s)

Image credit: MNIST dataset

GRAYSCALE IMAGES

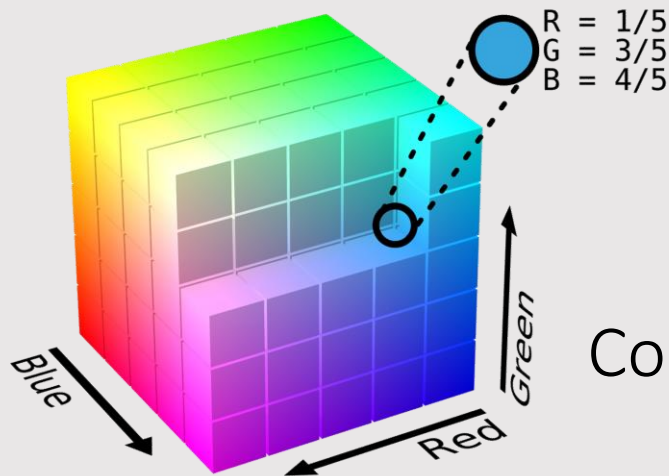
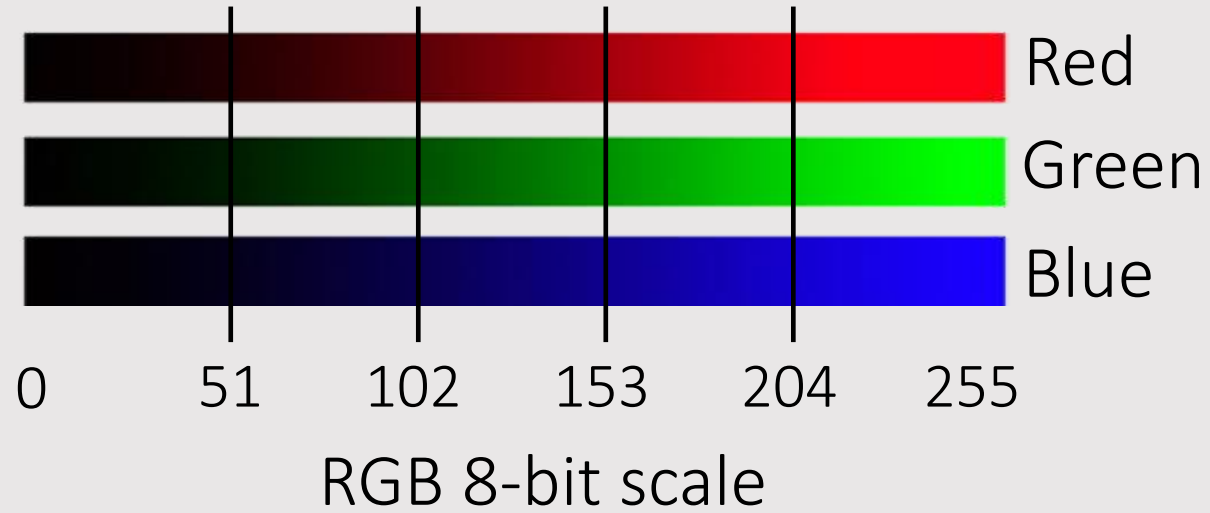


8-bit scale

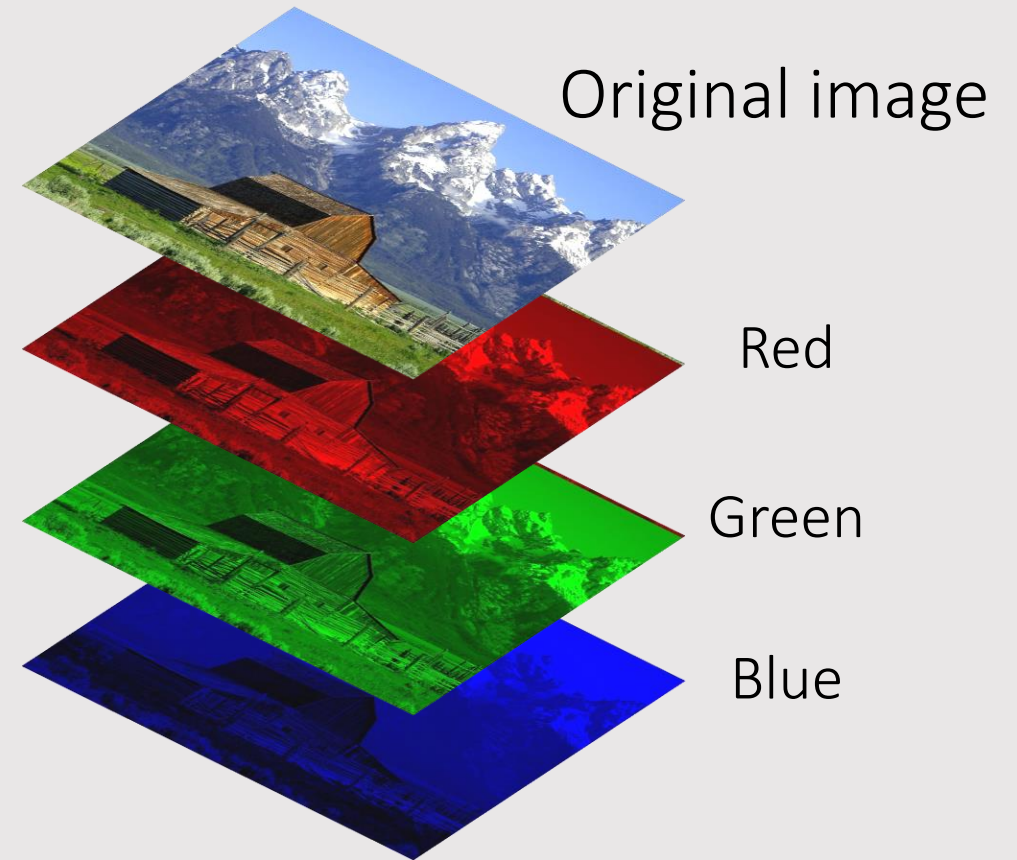


Grayscale image example

COLOR IMAGES



Color spectrum of RGB



Color image example

COMMONLY USED UNITS OF COLOR DEPTH



8-bit color

0 51 102 153 204 255

Normalized 8-bit color

0 0.2 0.4 0.6 0.8 1.0

HEX

#000000 #333333 #666666 #999999 #cccccc #ffffff

R G B

LOADING IMAGES IN PYTHON WITH matplotlib

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
```

Import matplotlib.pyplot and matplotlib.image

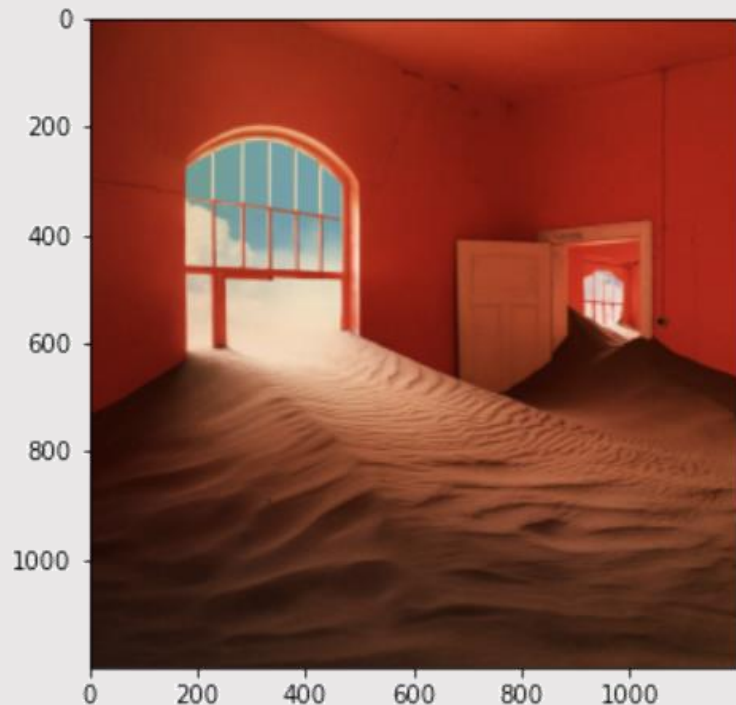
```
img = mpimg.imread('TSR.jpg')
```

Load the image with imread()

```
fig = plt.figure(figsize=(5, 5))
```

Plot the image with imshow()

```
imgplot = plt.imshow(img)
```



NOTE: When using imread, make sure you have your image in the **same directory** as ipynb OR Have a **correct directory path** to the image.

NOTE: plt.imshow() automatically normalizes the pixel values between 0 – 1 or 0 - 255. Explicitly set vmin and vmax parameters to 0, 1 or 0, 255 to force standard normalization.

Image credit: Tame Impala Official Instagram

READING IMAGES AS NUMPY ARRAYS

```
img.shape
```

```
(1200, 1200, 3)
```

Typical images have (Height, Width, 3)

```
img_red = img[:, :, 0]  
img_green = img[:, :, 1]  
img_blue = img[:, :, 2]
```

Extract Red (Top) Channel
Green (Middle)
Blue (Bottom)

```
print(img_red, img_red.shape)
```

```
[[107 108 107 ... 163 165 167]  
 [110 111 110 ... 165 164 163]  
 [109 110 109 ... 165 162 160]  
 ...  
 [ 42  41  41 ...  91  88  86]  
 [ 43  43  42 ...  89  87  85]  
 [ 44  43  43 ... 100  98  96]] (1200, 1200)
```

Extracted channel is a Numpy 2D array of 8-bit color values with shape (W, H).

NOTE: some images may use normalized 8-bit color instead of standard values.

WRITING IMAGE WITH matplotlib EXAMPLE:

Write Red channel of an image into a .png file

```
import numpy as np
```

Import Numpy

```
img_redonly = np.zeros((1200, 1200, 3), dtype = 'int')
```

Create an empty 3D array that has same dimension as original image

```
img_redonly[:, :, 0] = img[:, :, 0]
```

Populate the first layer of channel axis with Red channel of the image.

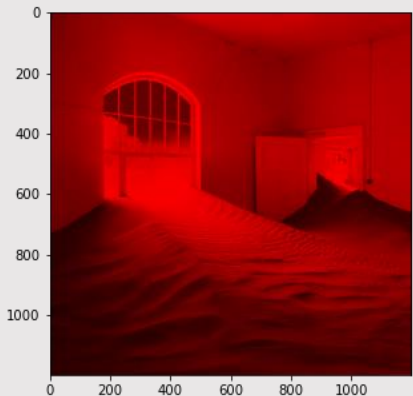
```
fig = plt.figure(figsize=(5, 5))
```

```
imgplot_redonly = plt.imshow(img_redonly)
```

Plot the new image with imshow()

```
plt.savefig('imgplot_redonly.png')
```

Save the image with savefig('filename.xyz')



imgplot_redonly.png

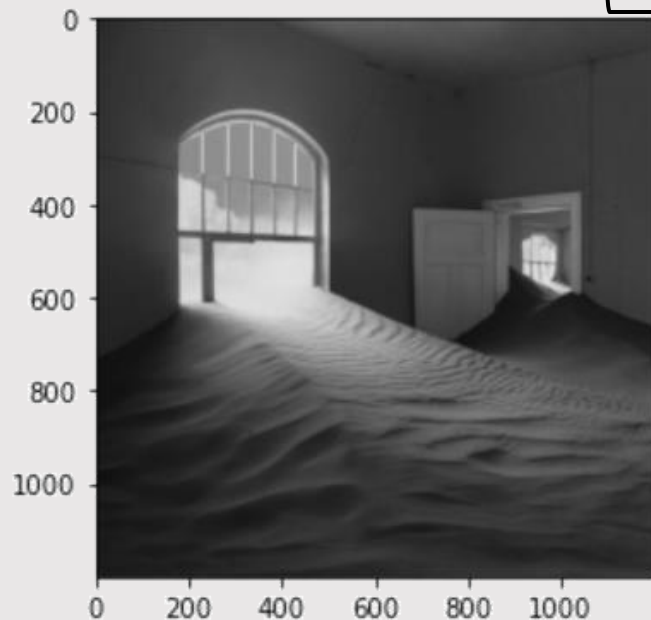
NOTE: The empty array must be of type 'int' to correctly represent 8-bit color depth

CONVERTING COLOR IMAGE TO GRAYSCALE

```
from skimage import color
from skimage import io

img_gray = color.rgb2gray(io.imread('TSR.jpg'))
```

```
imgplot = plt.imshow(img_gray, cmap = 'gray')
```

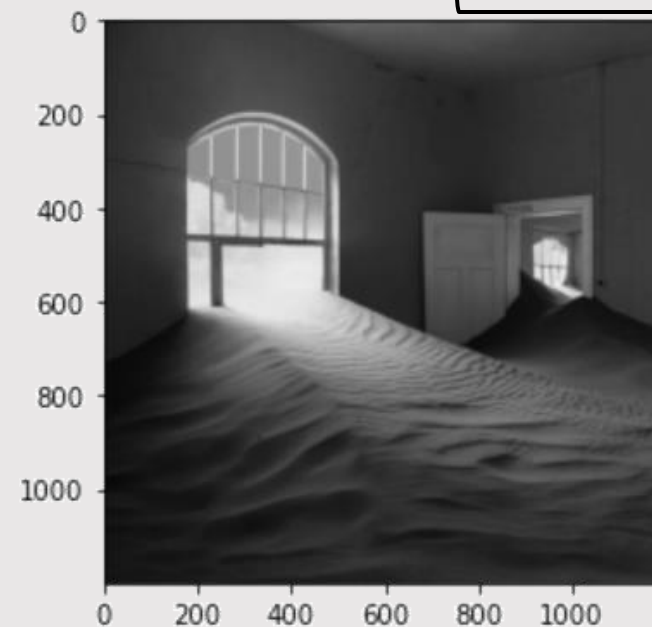


Need this line to tell imshow you are plotting grayscale

Using skimage

```
from PIL import Image
img_gray = Image.open('TSR.jpg').convert('L')
```

```
imgplot = plt.imshow(np.array(img_gray), cmap = 'gray')
```



Need this line to tell imshow you are plotting grayscale

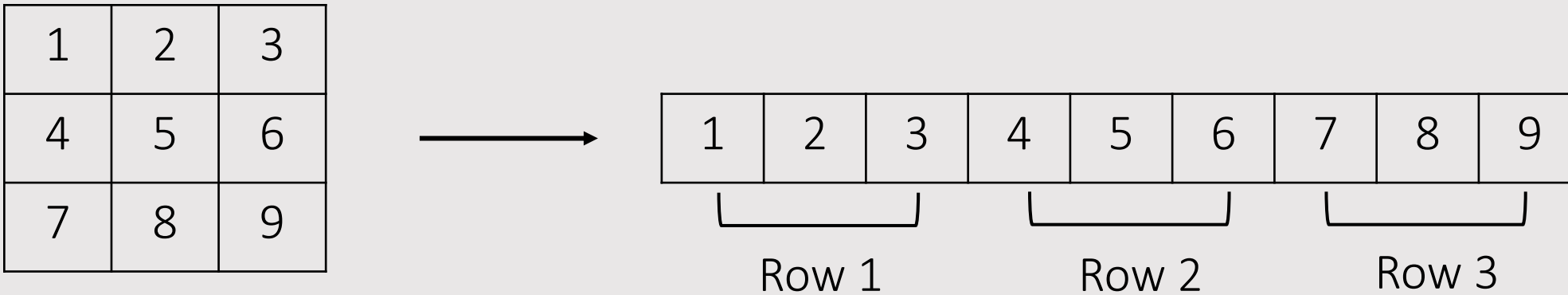
Need to cast into np.array() to turn image into numpy array

Using Pillow

NOTE: skimage uses normalized 8-bit scale and pillow uses standard 8-bit (0 - 255)

OPERATIONS ON IMAGE (GRAYSCALE)

CONVERTING 2D ARRAY INTO 1D ARRAY WITH `np.ndarray.flatten()`



```
sample_2d = np.vstack([np.arange(3), np.arange(3,6), np.arange(6,9)])
```

Construct `sample_2d`

```
print(sample_2d)
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

```
print(np.ndarray.flatten(sample_2d))
```

```
[0 1 2 3 4 5 6 7 8]
```

Use `np.ndarray.flatten()` to convert 2D array to 1D

ANALYZING AN IMAGE WITH PIXEL HISTOGRAM

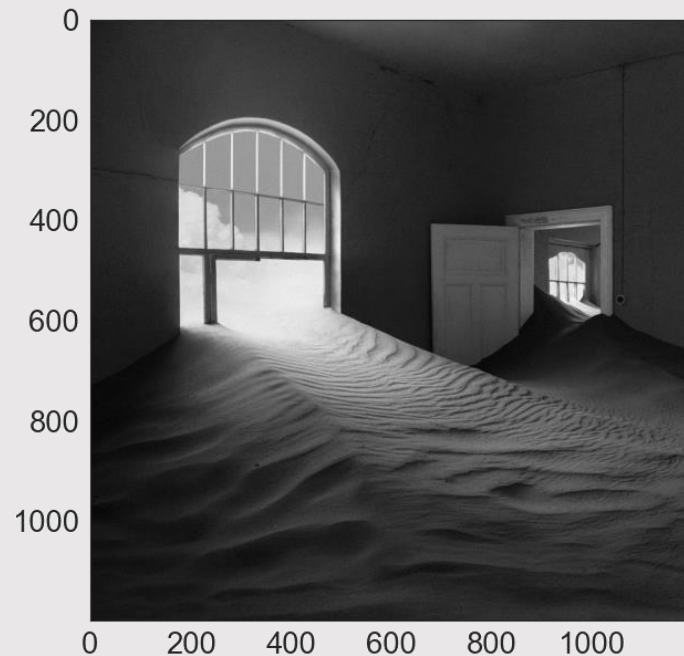
```
flattened_2D_arr = np.ndarray.flatten(img_gray)

fig = plt.figure(figsize=(10, 5))

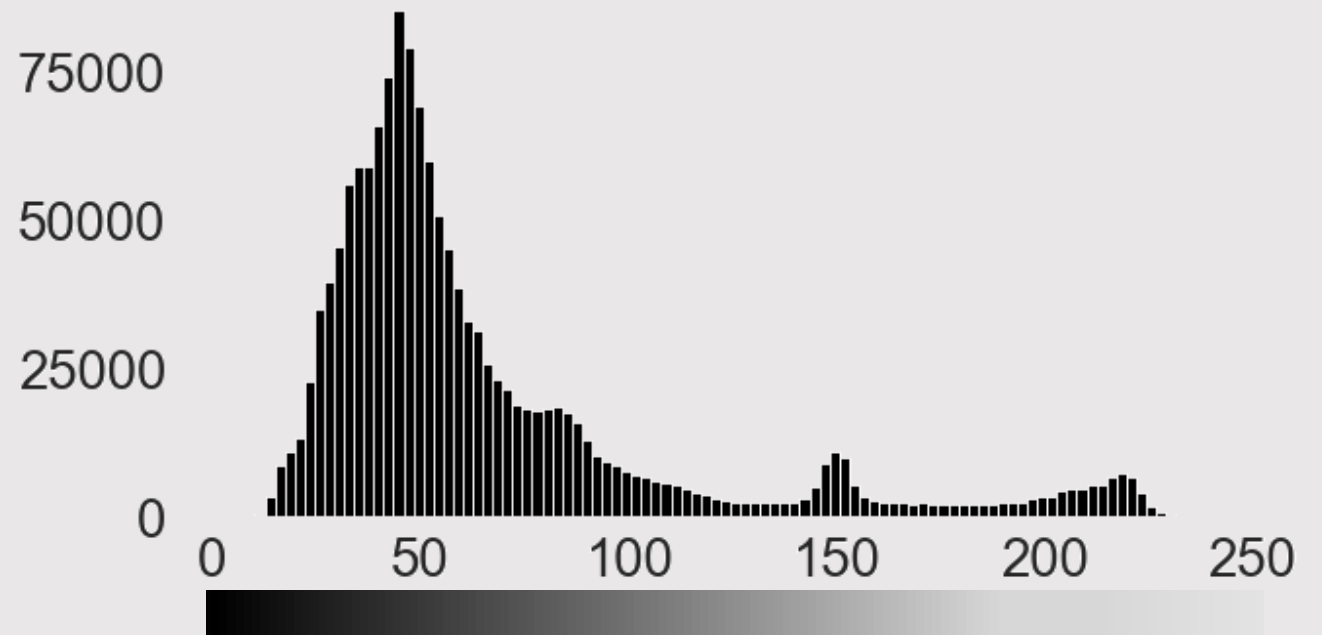
plt.hist(flattened_2D_arr, bins = 100, color = 'black')
```

Flatten img_gray 2D array (slide 11) -> 1D array

Plot the histogram of the pixel values

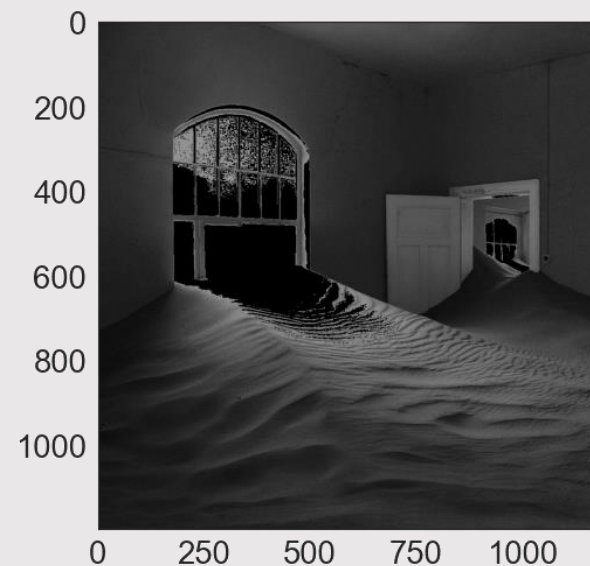
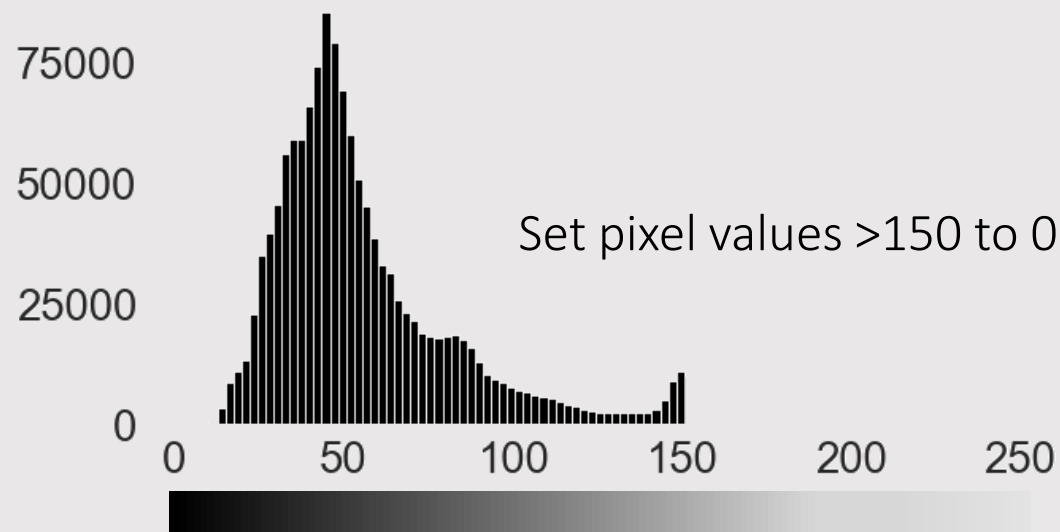
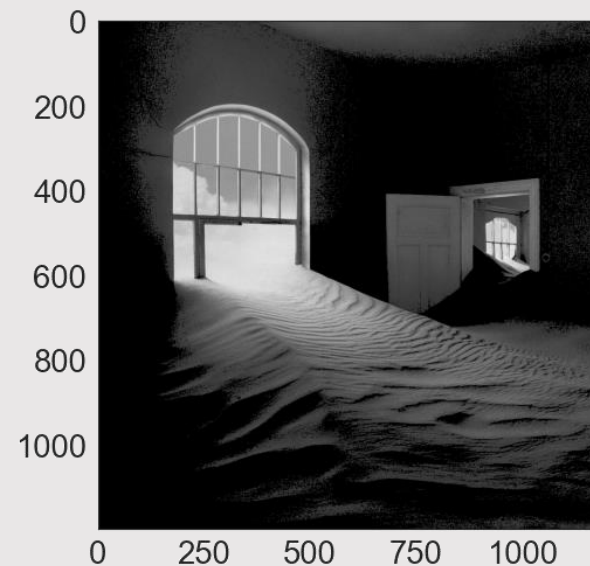
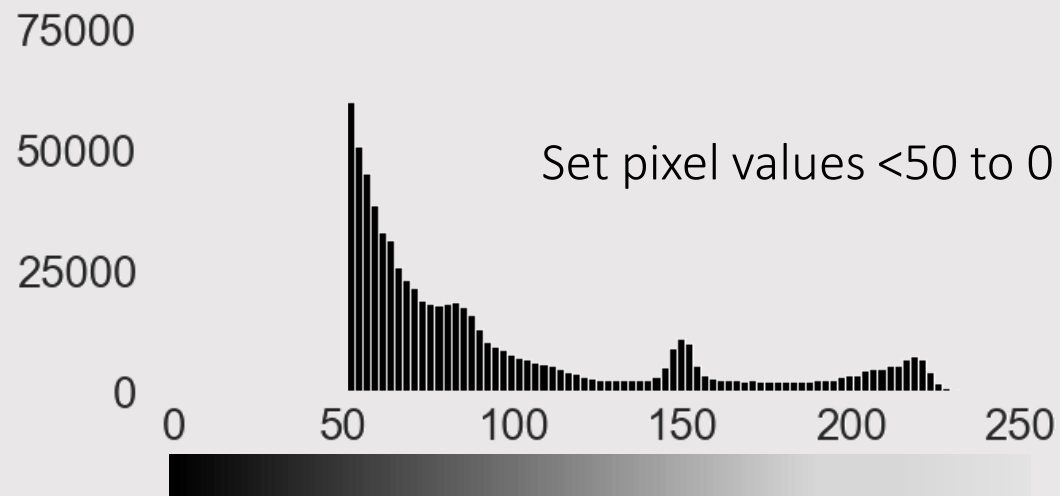


Original Greyscale image



Pixel value histogram

MANIPULATING IMAGE PIXELS WITH HISTOGRAM



MANIPULATING IMAGE PIXELS WITH HISTOGRAM: BOOLEAN MASK METHOD

Original 2D array

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

array2d

Boolean mask

T	T	T	T	T
T	T	T	T	T
T	T	T	T	F
F	F	F	F	F
F	F	F	F	F

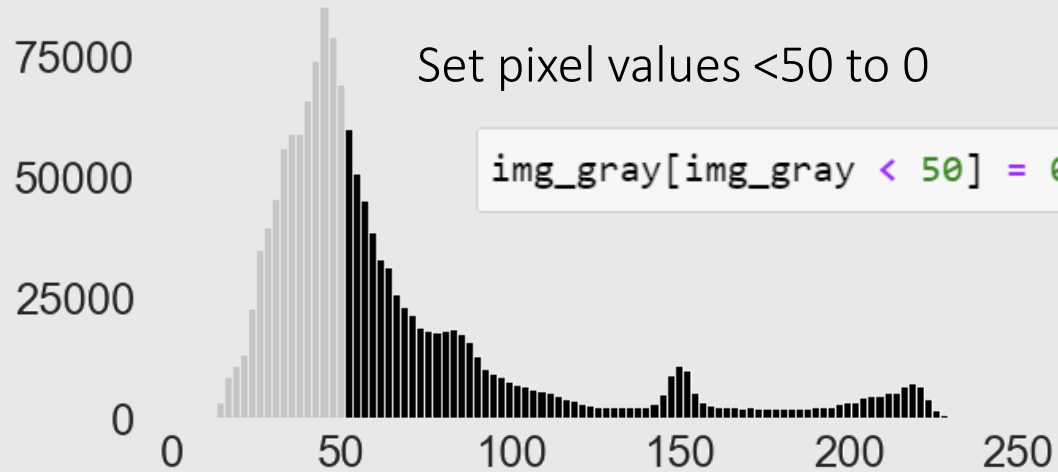
array2d < 15
T = True, F = False

Applying the mask

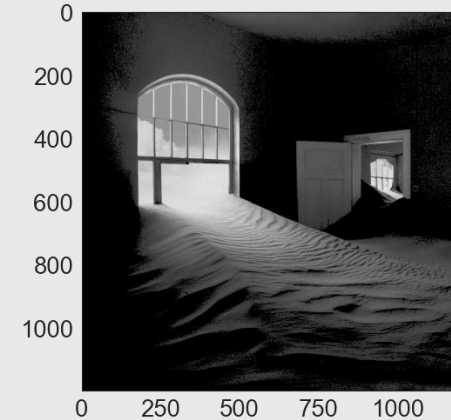
n	n	n	n	n
n	n	n	n	n
n	n	n	n	15
16	17	18	19	20
21	22	23	24	25

array2d[array2d < 15] = n

MANIPULATING IMAGE PIXELS WITH HISTOGRAM: BOOLEAN MASK METHOD



```
plt.imshow(img_gray, cmap = 'gray',  
vmin = 0, vmax = 255)
```



NOTE:

Set `vmin = 0, vmax = 255` in `imshow()` to properly normalize the image

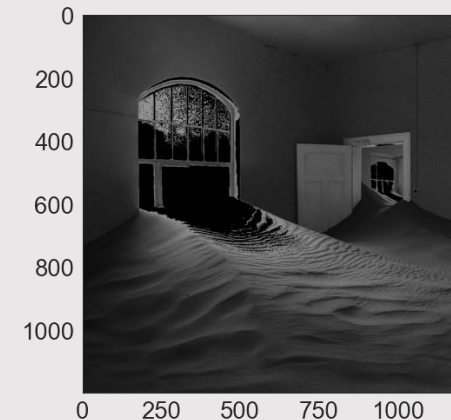
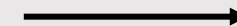
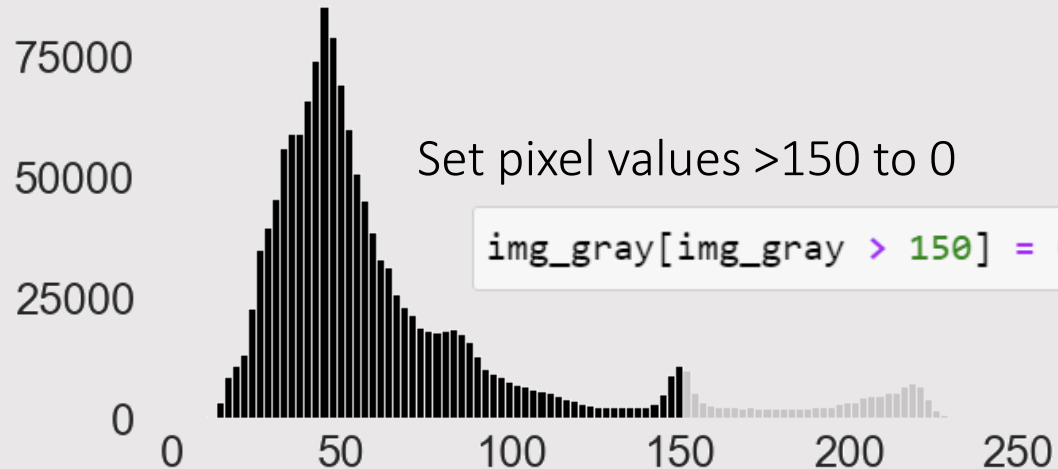


IMAGE FLIPPING WITH NUMPY FLIP FUNCTIONS

Original Image



Horizontal Flip



Vertical Flip



IMAGE FLIPPING WITH NUMPY FLIP FUNCTIONS

Horizontal Flip

1	2	3	→	3	2	1
4	5	6		6	5	4
7	8	9		9	8	7

`arr`

`np.fliplr(arr)`

Vertical Flip

1	2	3	→	7	8	9
4	5	6		4	5	6
7	8	9		1	2	3

`arr`

`np.flipud(arr)`

IMAGE FLIPPING WITH NUMPY FLIP FUNCTIONS

Original Image



Horizontal Flip



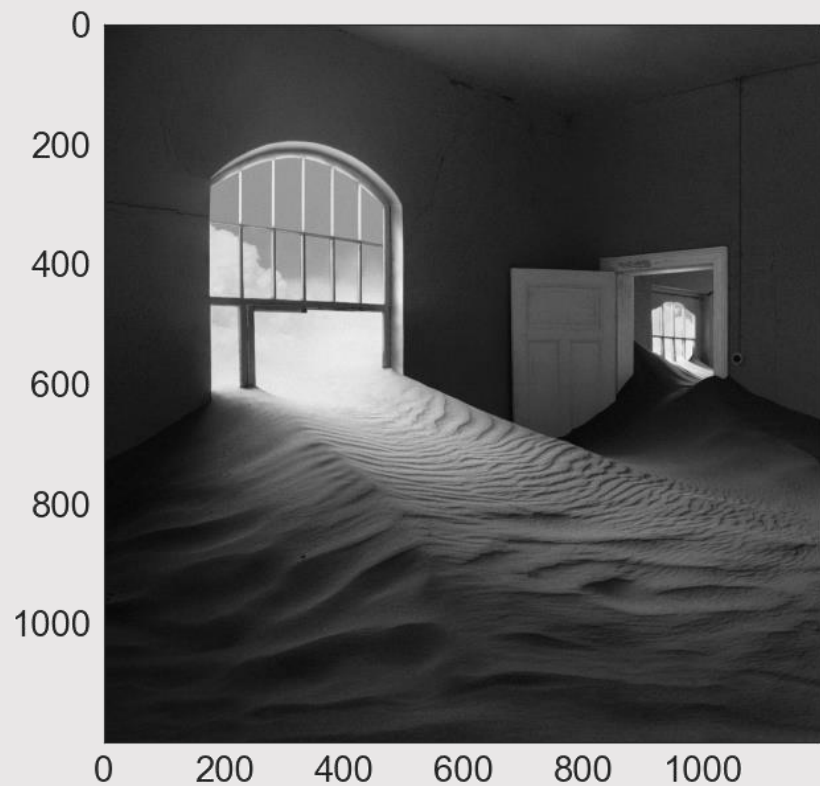
`np.fliplr(img)`

Vertical Flip

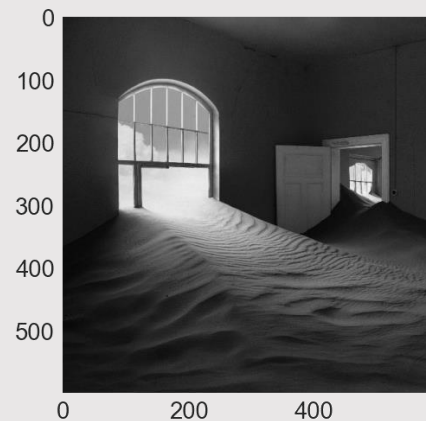


`np.flipud(img)`

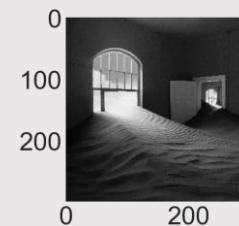
IMAGE DOWNSAMPLING



Original image
(1200 x 1200)



Downsampled by x2
(600 x 600)



Downsampled by x4
(300 x 300)

IMAGE DOWNSAMPLING WITH SIMPLE INDEXING (Pick one Method)


Original 2D array

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

array2d

Downsampling via indexing

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25



1	3	5
11	13	15
21	23	25

array2d[::2, ::2]

IMAGE DOWNSAMPLING BY SEGMENT AVERAGING

Original 2D array

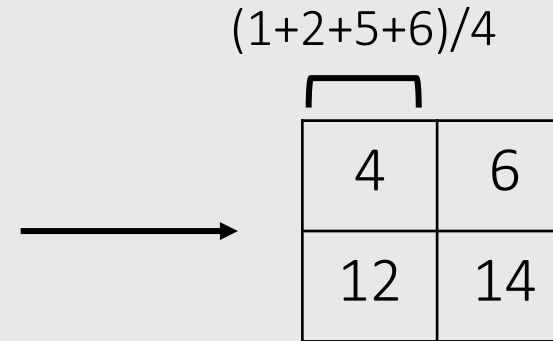
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

array2d

Downsampling via averaging

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Split the array into
segments



Downsampling rate = x2

Average each
segment + rounding

SIMPLE INDEXING VS SEGMENT AVERAGING

Downsample factor: x5

Downsample factor: x10

Original Image



Pick one method



Segment averaging method

Images are magnified to scale

Smooother images

IMAGE BLENDING

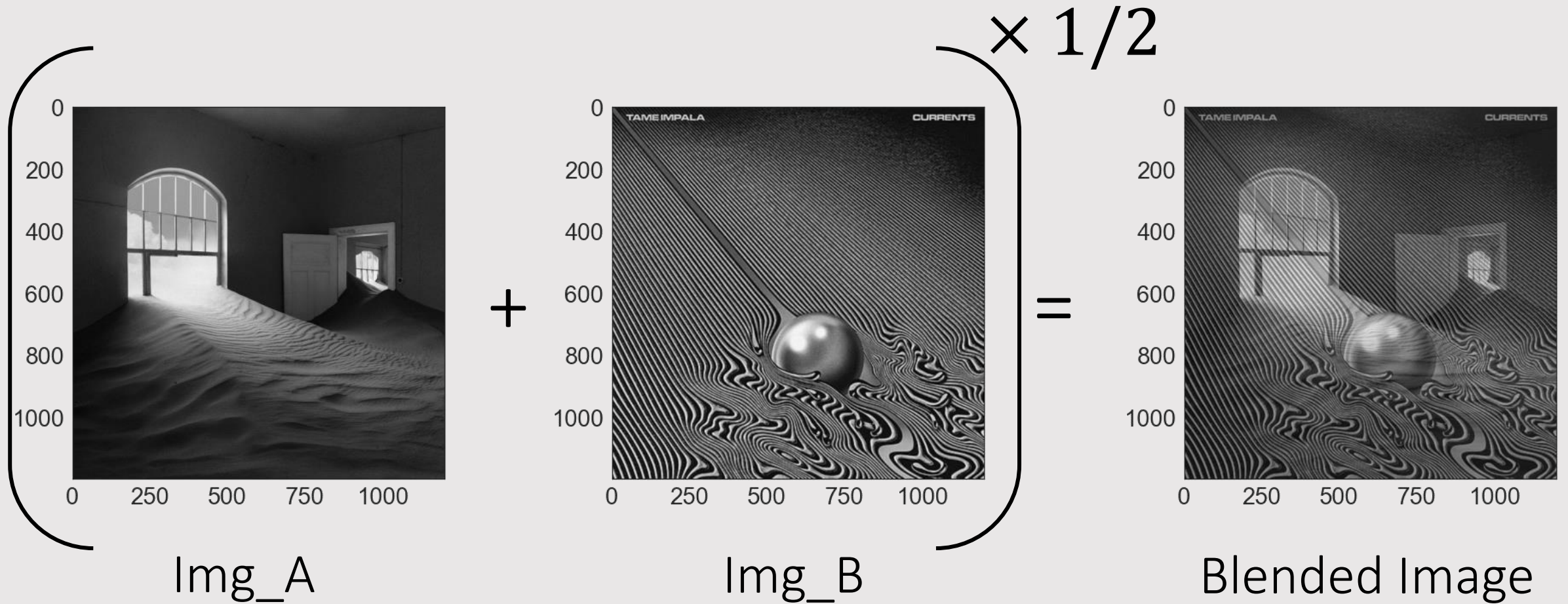
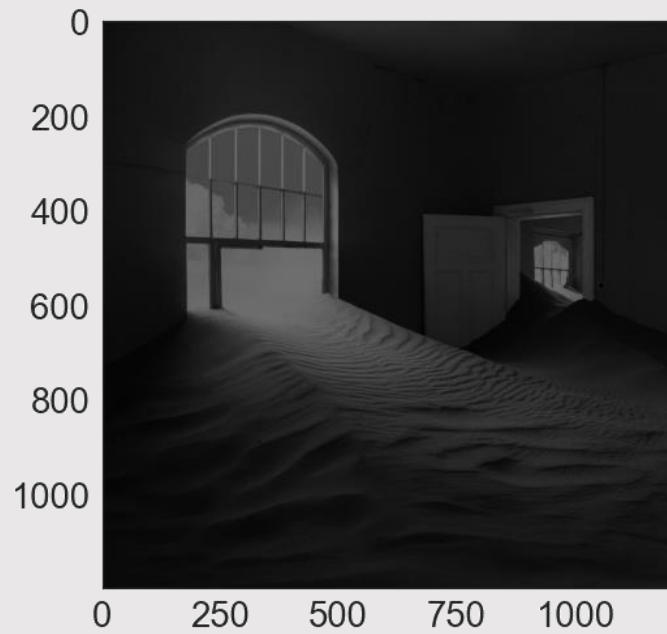


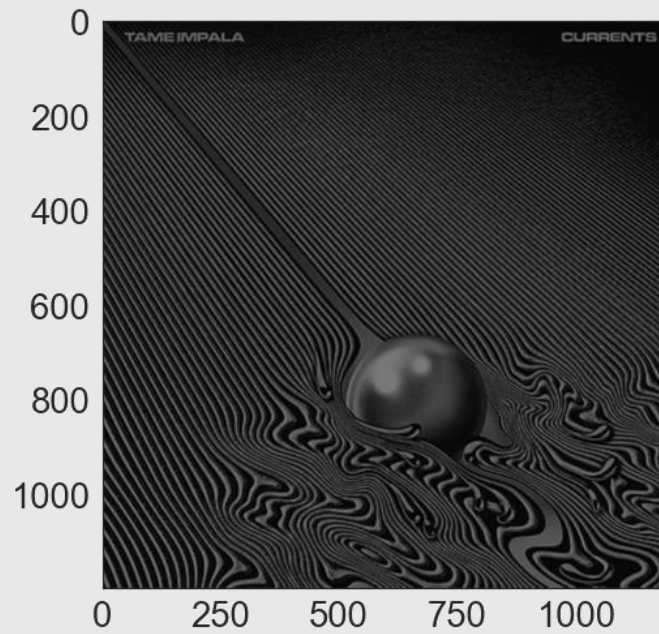
Image credit: Tame Impala Official Instagram

IMAGE BLENDING



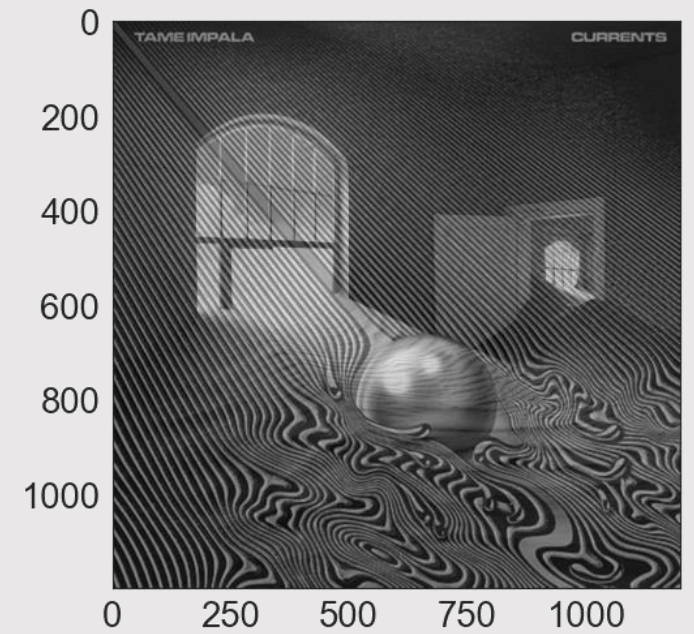
$0.5 * \text{Img_A}$

+



$0.5 * \text{Img_B}$

=



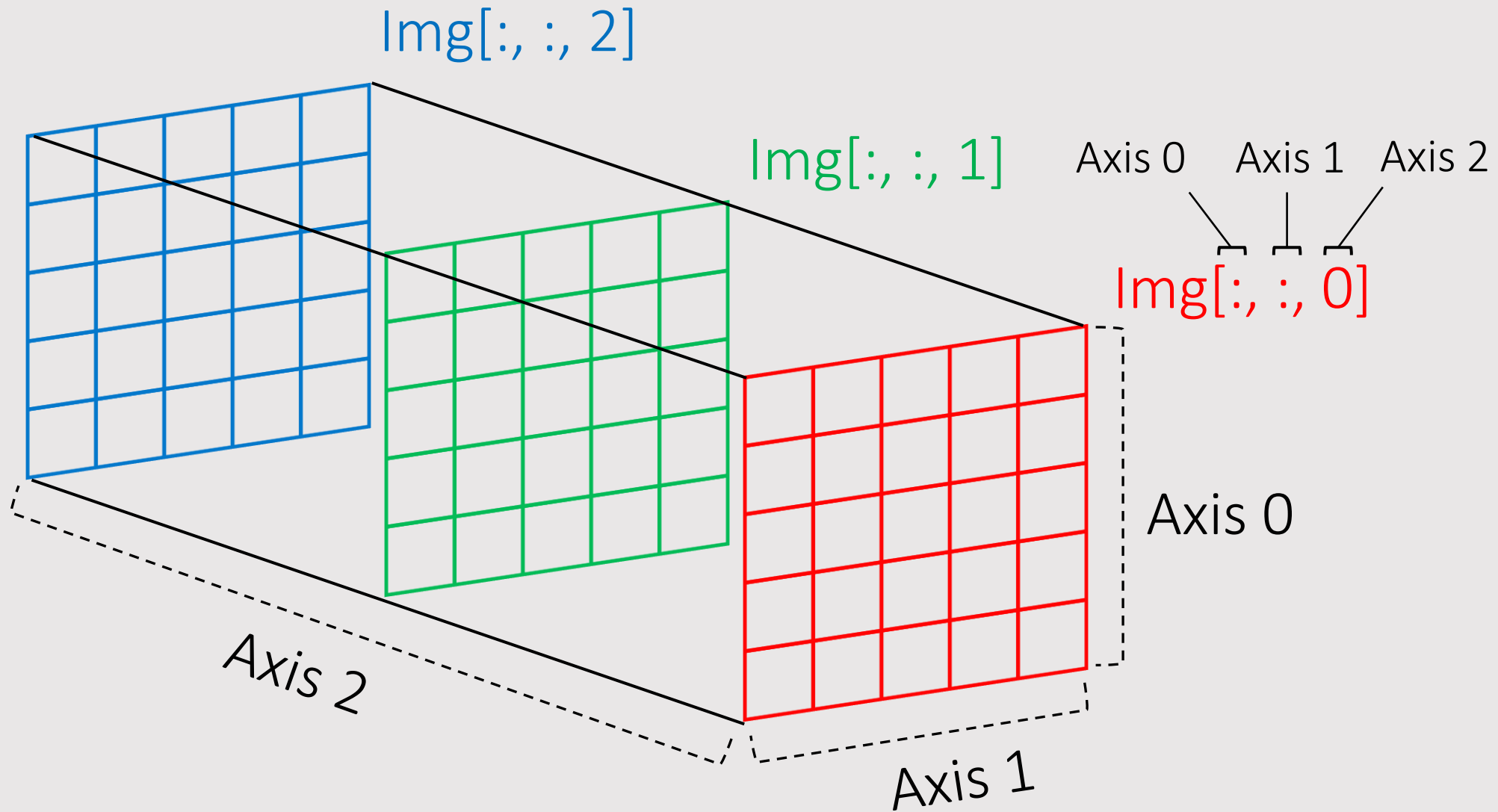
$0.5 * \text{Img_A} + 0.5 * \text{img_B}$

Question:

What would be the effects of using different weights - e.g. $0.3 * \text{img_A} + 0.7 * \text{img_B}$?

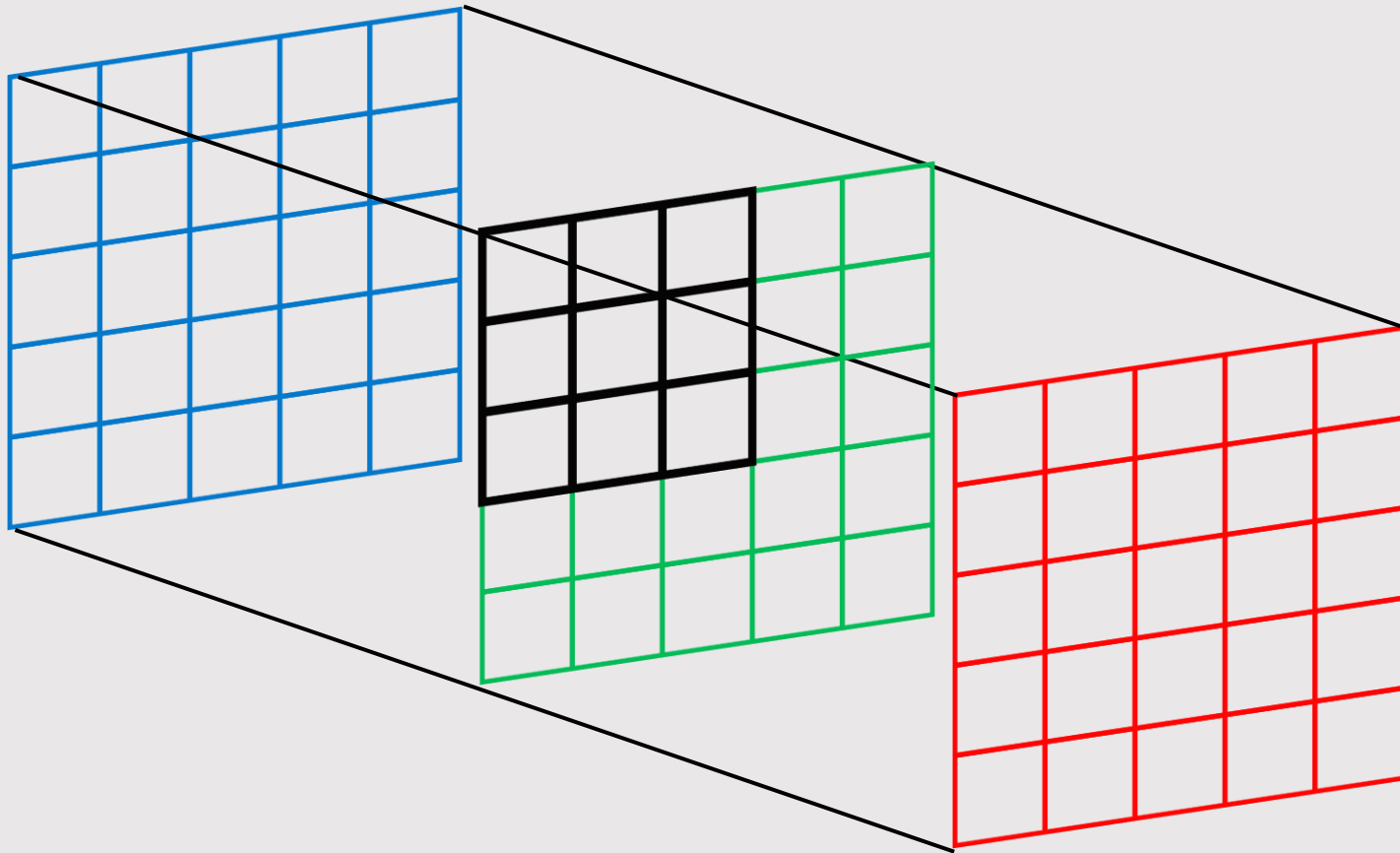
OPERATIONS ON IMAGE (COLOR)

WORKING WITH 3D ARRAYS: COLOR IMAGE ARRAY STRUCTURE



WORKING WITH COLOR IMAGE ARRAY: INDEXING

Desired array subset



Code

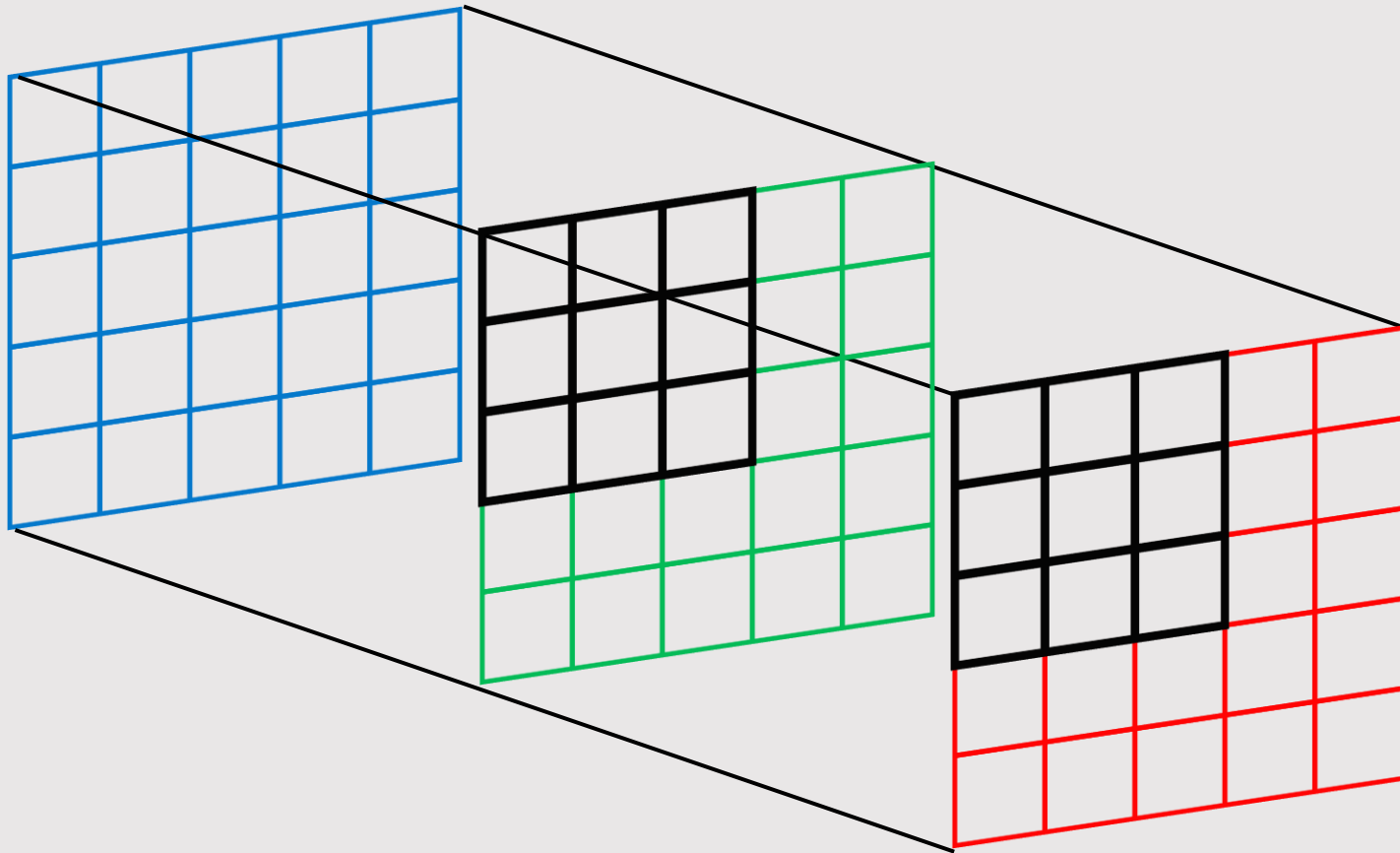
```
Img[:3, :3, 1]
```

Output array shape

(3, 3)

WORKING WITH COLOR IMAGE ARRAY: INDEXING

Desired array subset



Code

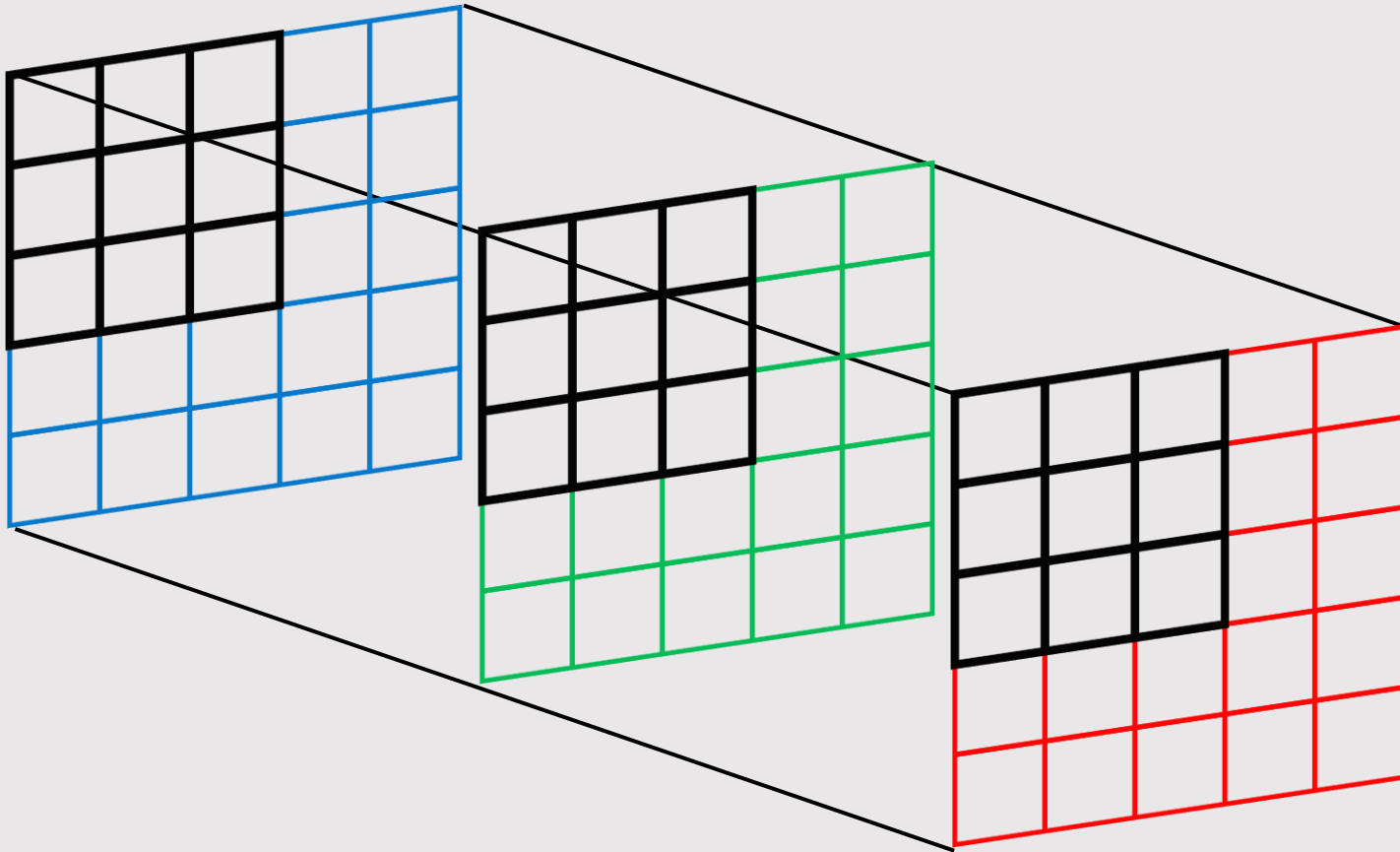
```
Img[:3, :3, :2]
```

Output array shape

```
(3, 3, 2)
```

WORKING WITH COLOR IMAGE ARRAY: INDEXING

Desired array subset



Code

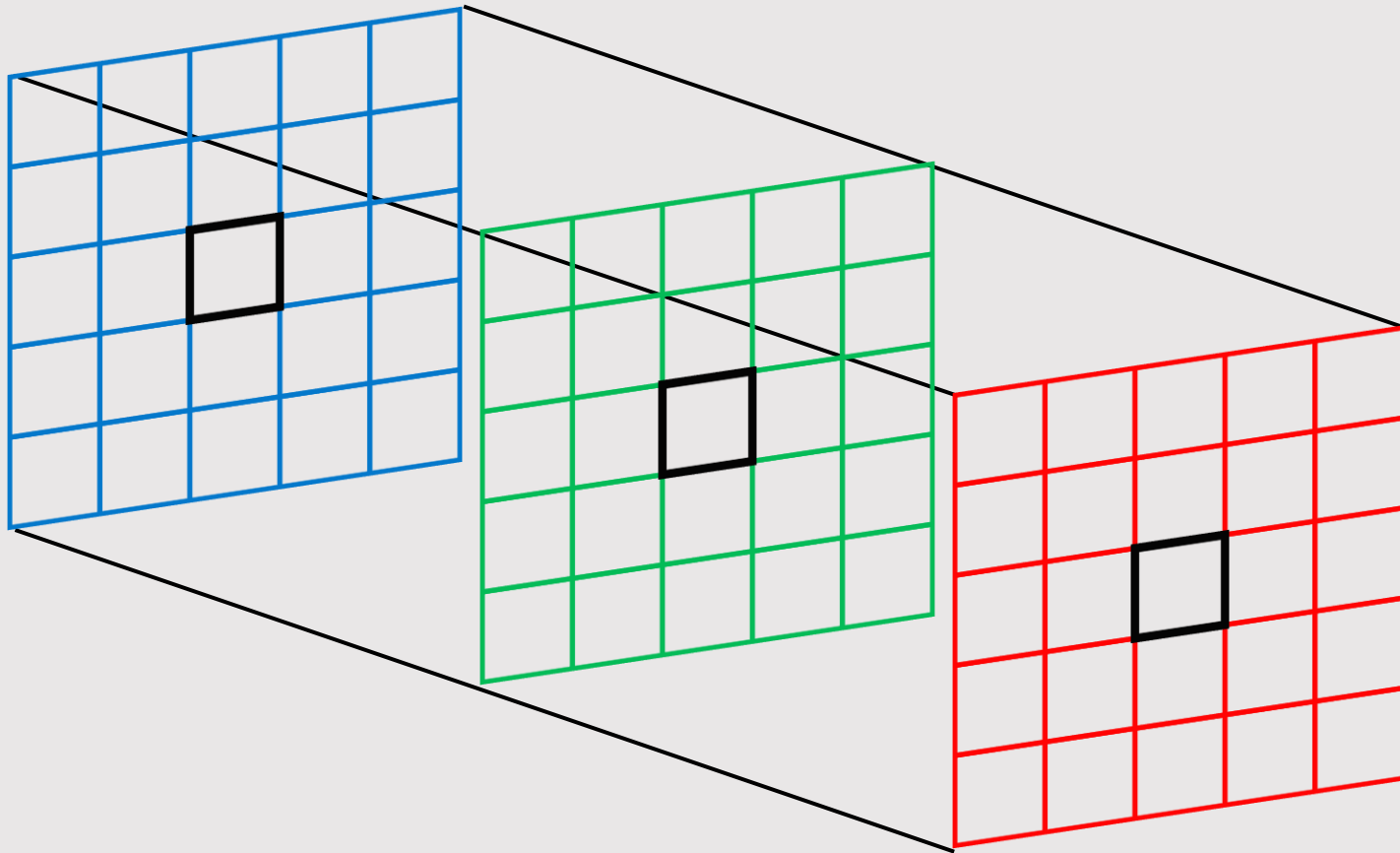
```
Img[:3, :3, :]
```

Output array shape

```
(3, 3, 3)
```

WORKING WITH COLOR IMAGE ARRAY: INDEXING

Desired array subset



Code

```
Img[2, 2, :]
```

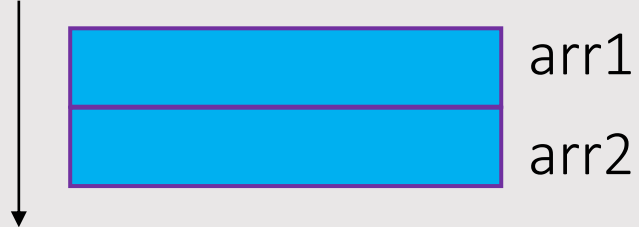
Output array shape

(3,)

CONSTRUCTING 3D ARRAYS FROM 2D ARRAYS

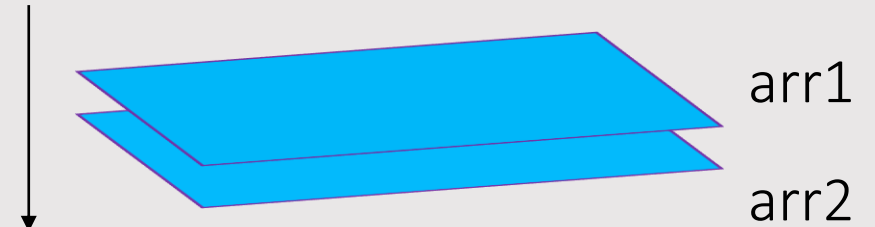
Recall `np.stack()` can be used to stack arrays alongside new dimension

New dimension



1D -> 2D

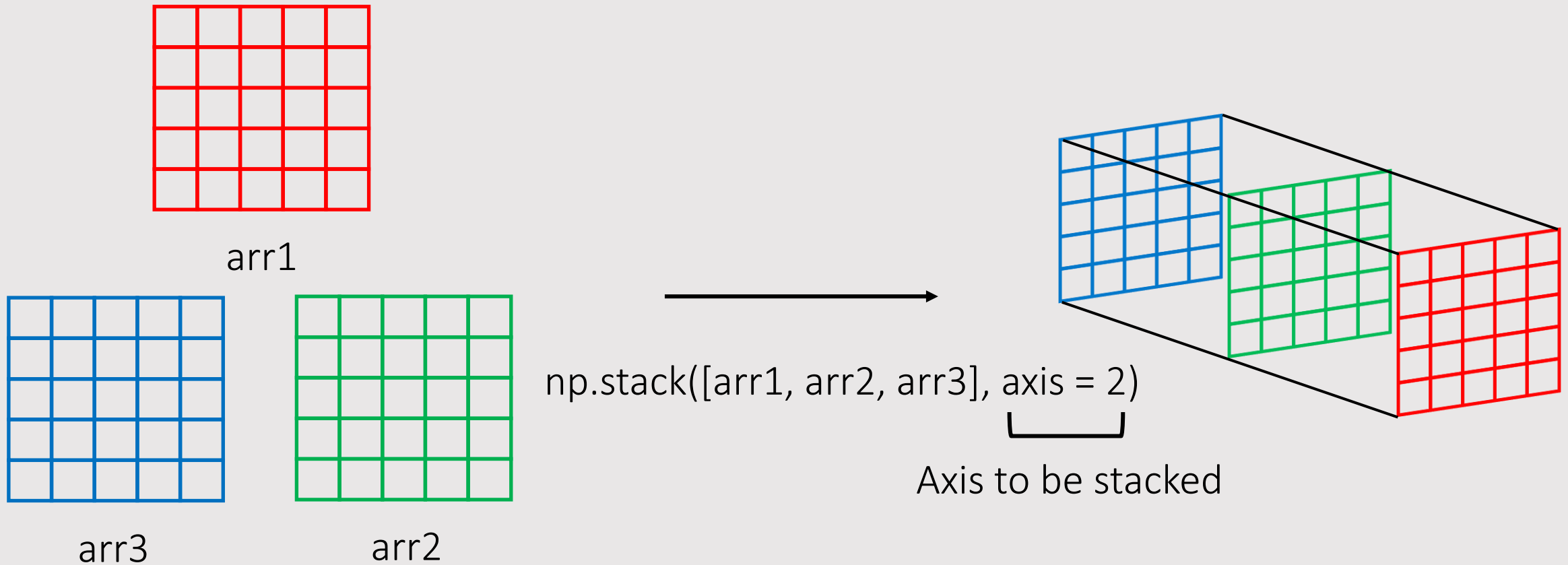
New dimension



2D -> 3D

We can use `np.stack()` to construct RGB image from 2D arrays

CONSTRUCTING 3D ARRAYS FROM 2D ARRAYS



EXPANDING IMAGE OPERATIONS TO COLOR:

Image Flipping

```
img = mpimg.imread('TSR.jpg')  
img = img.copy()
```

Load a color image

```
img_red = img[:, :, 0]  
img_green = img[:, :, 1]  
img_blue = img[:, :, 2]
```

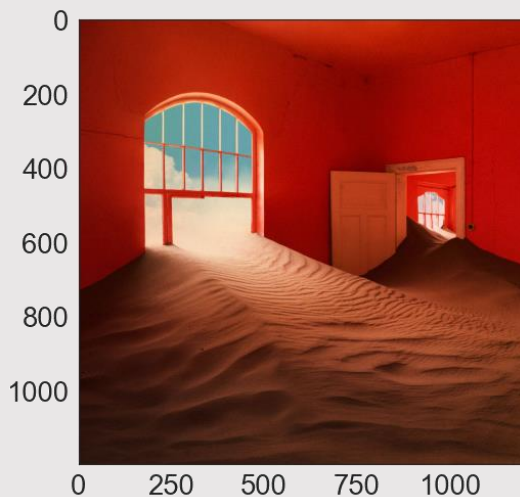
Extract Red, Green, Blue channels

```
img_red_flipped = np.fliplr(img_red)  
img_green_flipped = np.fliplr(img_green)  
img_blue_flipped = np.fliplr(img_blue)
```

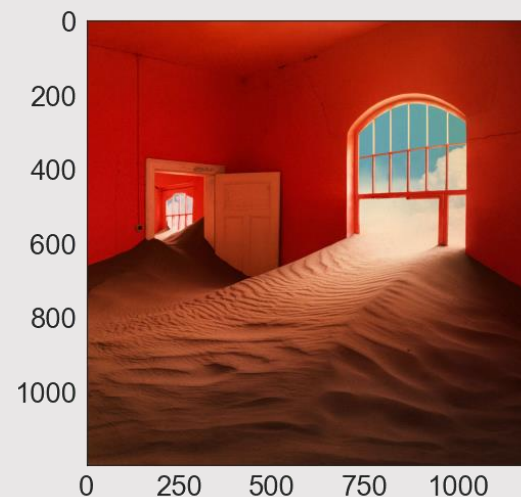
Apply flip on each channel

```
flipped_img = np.stack([img_red_flipped, img_green_flipped, img_blue_flipped], axis = 2)
```

Reconstruct the RGB image



Original
image



Flipped image
(Horizontal)

EXPANDING IMAGE OPERATIONS TO COLOR:

Image Flipping (Partial)

```
img = mpimg.imread('TSR.jpg')  
img = img.copy()
```

```
img_red = img[:, :, 0]  
img_green = img[:, :, 1]  
img_blue = img[:, :, 2]
```

```
img_green_flipped = np.fliplr(img_green)
```

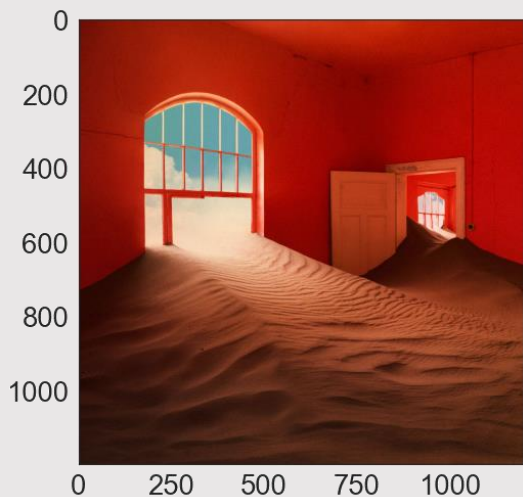
```
flipped_img = np.stack([img_red, img_green_flipped, img_blue], axis = 2)
```

Load a color image

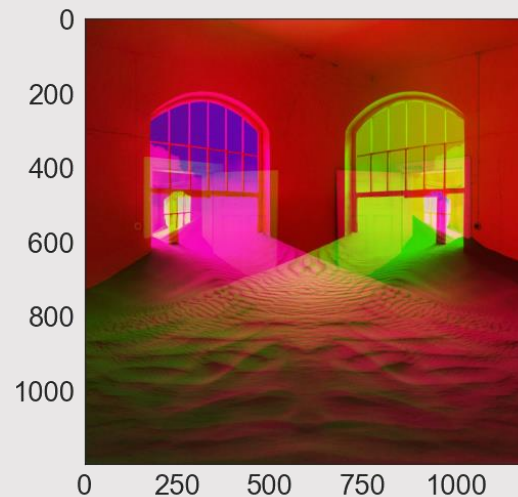
Extract Red, Green, Blue channels

Apply flip on subset of channels

Reconstruct the RGB image



Original
image



Partially Flipped image
(Horizontal)

EXPANDING IMAGE OPERATIONS TO COLOR:

Image Downsampling

```
img = mpimg.imread('TSR.jpg')  
img = img.copy()
```

Load a color image

```
img_red = img[:, :, 0]  
img_green = img[:, :, 1]  
img_blue = img[:, :, 2]
```

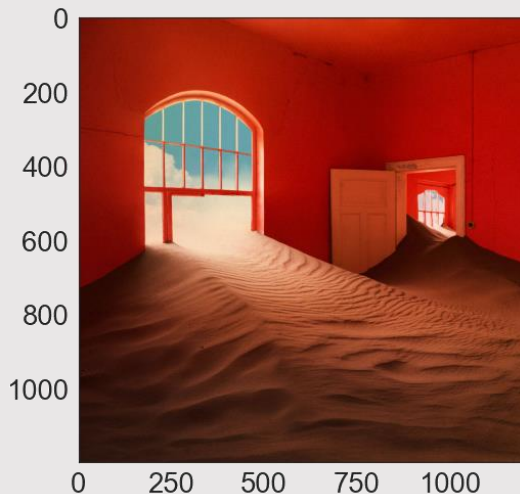
Extract Red, Green, Blue channels

```
img_red_DS = img_red[::10, ::10]  
img_green_DS = img_green[::10, ::10]  
img_blue_DS = img_blue[::10, ::10]
```

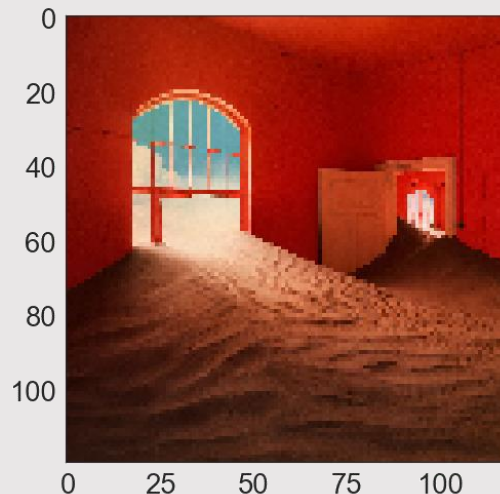
Down sample each channel

```
downsampled_img = np.stack([img_red_DS, img_green_DS, img_blue_DS], axis = 2)
```

Reconstruct the RGB image



Original
image



Downsampled image
(x10)



EXPANDING IMAGE OPERATIONS TO COLOR:

Image Blending

```
img1 = mpimg.imread('TSR.jpg')
img1 = img1.copy()

img2 = mpimg.imread('currents.jpg')
img2 = img2.copy()
```

Load 2 images to be blended

```
img1_red, img2_red = img1[:, :, 0], img2[:, :, 0]
img1_green, img2_green = img1[:, :, 1], img2[:, :, 1]
img1_blue, img2_blue = img1[:, :, 2], img2[:, :, 2]
```

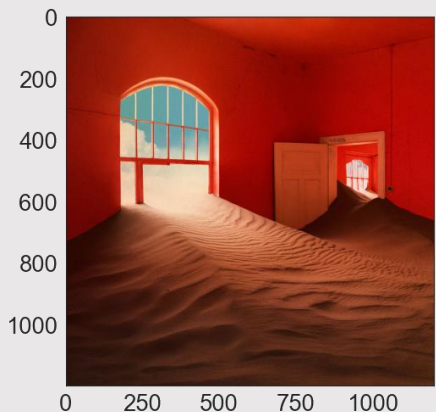
Extract channels from each image

```
blended_img_red = 0.5 * img1_red + 0.5 * img2_red
blended_img_green = 0.5 * img1_green + 0.5 * img2_green
blended_img_blue = 0.5 * img1_blue + 0.5 * img2_blue
```

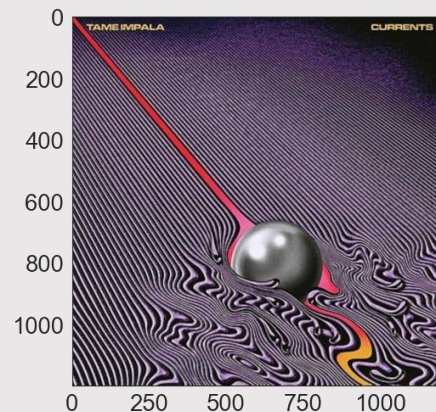
Add each channel with weights

```
blended_img = np.stack([blended_img_red, blended_img_green, blended_img_blue], axis = 2)
blended_img = blended_img.astype('int')
```

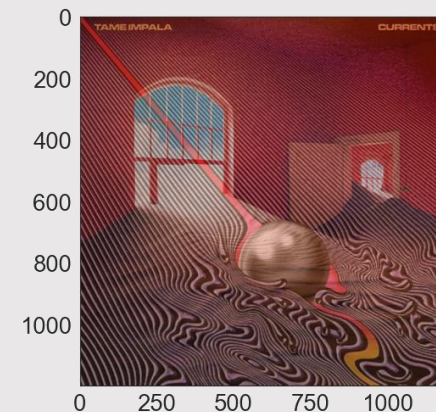
Reconstruct the image - **Make sure to convert the pixel values into integers**



Original
image 1



Original
image 2



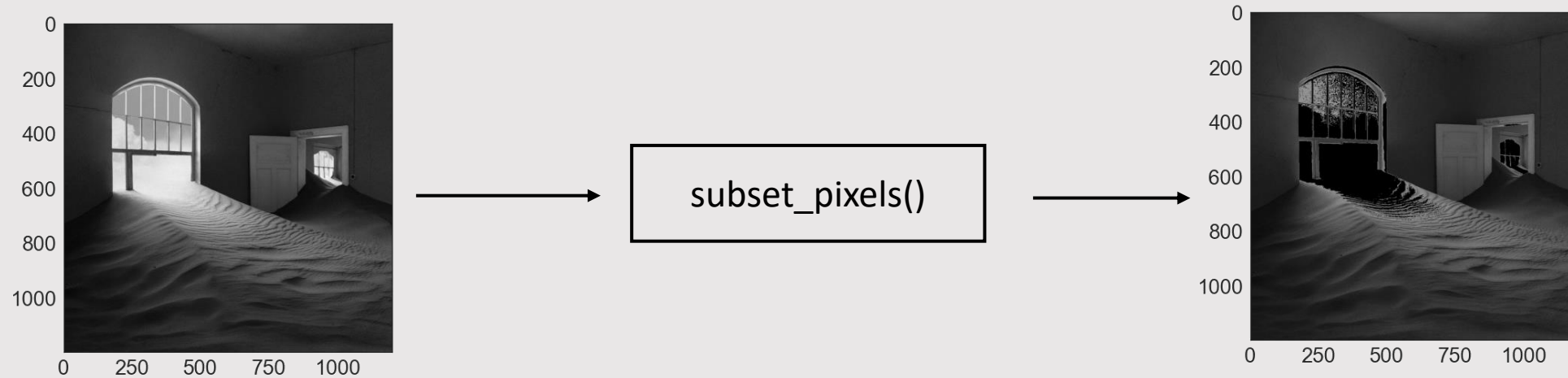
Blended
image

LAB ASSIGNMENTS

Download ipynb template in Canvas page:

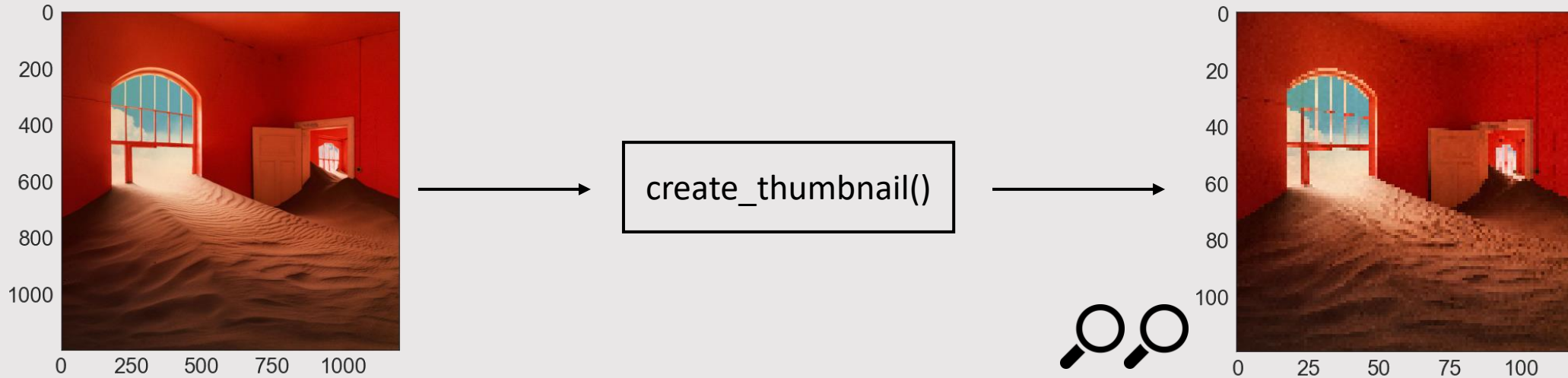
Assignments/Lab 3 report -> click “Lab 3 Report Templates”

EXERCISE 1: Generalized function for subsetting pixels



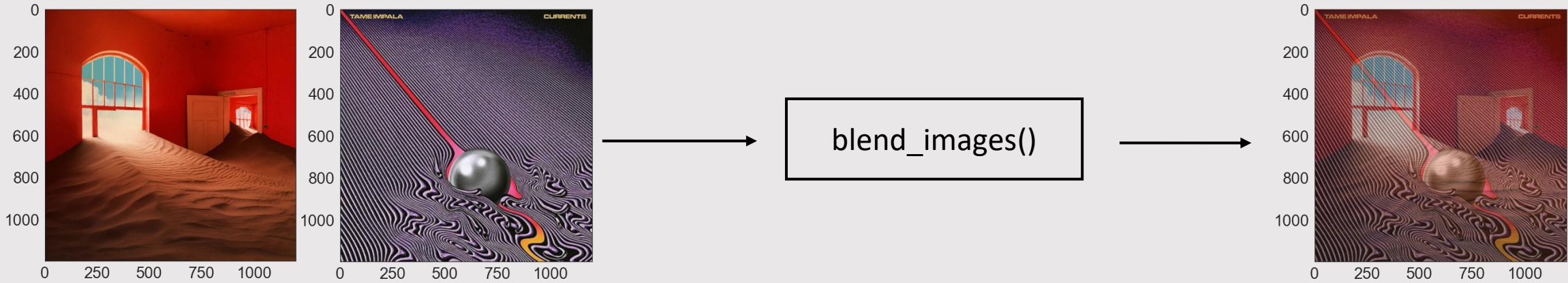
- Create a function **`subset_pixels()`** which takes a grayscale image as an input, and outputs the image with subsetting pixels according to minimum and maximum pixel values.
- The function should accept following parameters
 - Image – 2D array corresponding to input image
 - `min_pixel_depth` – minimum pixel depth value
 - `max_pixel_depth` – maximum pixel depth value
 - `replacement_val` – a pixel depth value to mask pixels that fall outside of min and `max_pixel_depth`
- Your function should mask the pixels s.t. values that fall outside of min and max values are set to `replacement_val`.
- Test your function against 3 provided parameter sets on a single grayscale image, and save them as `e1_output1.png`, `e1_output2.png`, `e1_output3.png`.

EXERCISE 2: Thumbnail generator function



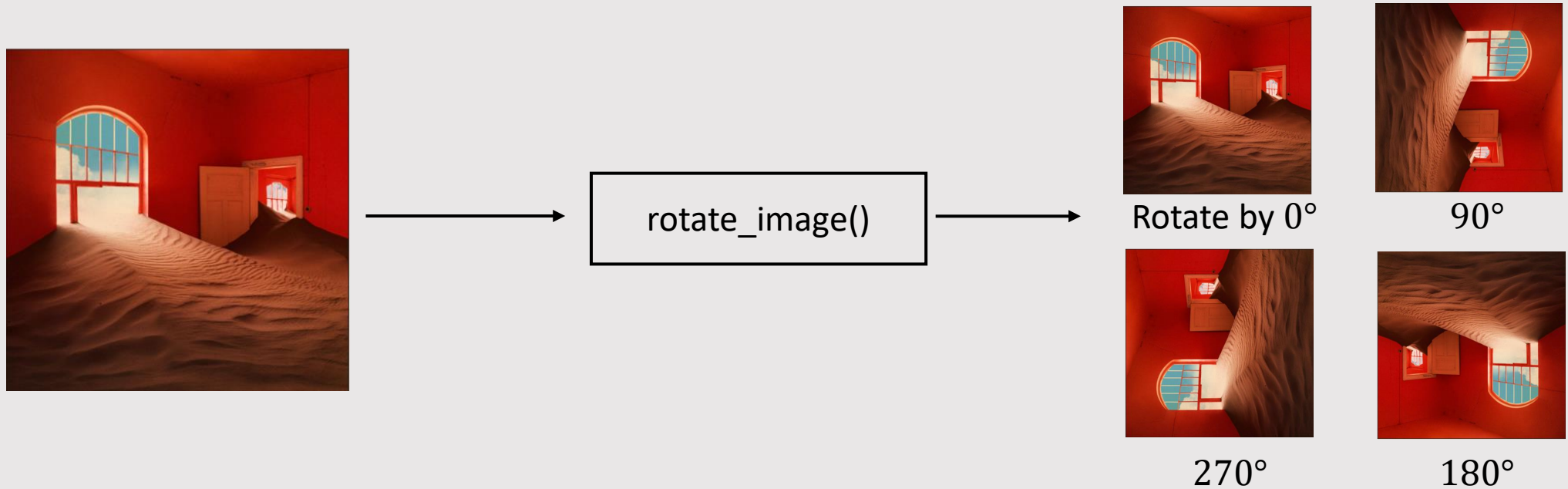
- Create a function **create_thumbnail()** which takes an RGB image as an input, and outputs the downsampled RGB image according to given compression rate using the **segment averaging technique** (slide 23).
- The function should accept following parameters
 - image – 3D array corresponding to input image
 - downsampling_rate - an integer corresponding to the ratio between dimensions of original vs downsampled image. e.g. 10, 5, 3.
- Test your function against a provided color image with following downsampling rate: 5x, 10x, 20x. Save your outputs as e2_output1.png, e2_output2.png, e2_output3.png.
- You can assume the function only handles square images and the downsampling rate divides both the height and width of the image
- NOTE: **DO NOT USE** pre-existing downsampling function from a Python package

EXERCISE 3: Generalized image blender function



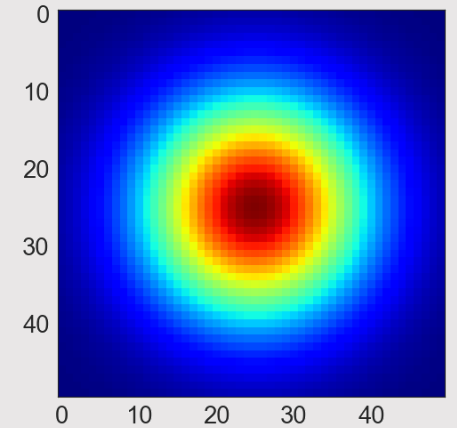
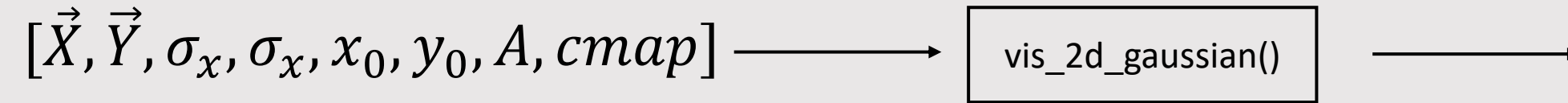
- Create a function **`blend_images()`** which takes multiple RGB images as an input, and outputs a blended image.
- The function should accept following parameters
 - `image_list` - A Python list of 3D arrays where each 3D array corresponds to an RGB image
 - `weight_list` - A Python list of float values between (0, 1) corresponding to the pixel weight to be given to each image – e.g. `[0.2, 0.3, 0.1, 0.4]` for 4 images. The sum of the weights should be equal to 1.
- Test your function against a provided list of 5 images with following `weight_lists` (i.e. 2 blended images)
 - `[0.2, 0.2, 0.2, 0.2, 0.2]` – blend all 5 images
 - `[0.2, 0.3, 0.5]` – blend first 3 images
- NOTE: **DO NOT USE** pre-existing image blending functions.
- Save your outputs as `e3_output1.png`, `e3_output2.png`.

EXERCISE 4: Image rotation function



- Create a function **`rotate_images()`** which takes a color image as an input, and outputs a rotated image by multiples of 90° in a **clockwise** direction.
- The function should accept following parameters
 - Image - A 3D array corresponding to a color image
 - rotate angle – Angle to rotate the image. Takes one of 4 values $[0^\circ, 90^\circ, 180^\circ, 270^\circ]$.
- Test your function against a provided image with 0° , 90° , 180° and 270° rotations (i.e. generate 4 images). Save your outputs as `e4_output1.png`, `e4_output2.png`, `e4_output3.png`, `e4_output4.png`.
- NOTE: **DO NOT USE** pre-existing rotation function from a Python package

EXERCISE 5: 2D Gaussian image generator



- 2D Gaussian function is provided by an equation: $f(x, y) = A \exp\left(-\left(\frac{(x - x_0)^2}{2\sigma_x^2} + \frac{(y - y_0)^2}{2\sigma_y^2}\right)\right)$
- You will implement a function `vis_2d_gaussian()` which creates an 2D array of $f(x, y)$ for given x, y -domains and outputs visualization of the function using `plt.imshow()`.
- The function should accept following parameters:
 - \vec{X}, \vec{Y} - 2D numpy arrays of floats corresponding to grids of x and y-coordinates respectively (more detail in template).
 - σ_x, σ_y - standard deviations in x, y directions
 - x_0, y_0 - x center and y center coordinates
 - A – Scaling factor of the function. Set this to **255** so that $f(x, y) \in (0, 255)$ thus consistent with 8-bit color code
 - `cmap` – color spectrum to be used for the plotting function
- Test your function against 2 sets of parameters provided by the lab template. For each parameter set, produce 2 plots one using `cmap = 'gray'` (grayscale) and other one using `cmap = 'jet'` (blue to red color spectrum). Save your outputs as `e5_output1.png`, `e5_output2.png`, `e5_output3.png`, `e5_output4.png`.

SUPPLEMENTARY: DEBUGGING YOUR CODE VIA LOGGING & COMMON ARRAY MISTAKES

GENERAL TIPS ON MINIMIZING ERRORS

Do not panic when you get errors

Outline your code structure ahead of time

Keep your code organized

Test your code often

More tips on avoiding errors by Berkeley online textbook

<https://pythonnumericalmethods.berkeley.edu/notebooks/chapter10.00-Errors-Practices-Debugging.html>

BASIC DEBUGGING WITH PRINT()

Code block 1

`print("done running block 1")` → Runs when code block 1 doesn't produce an error

Code block 2

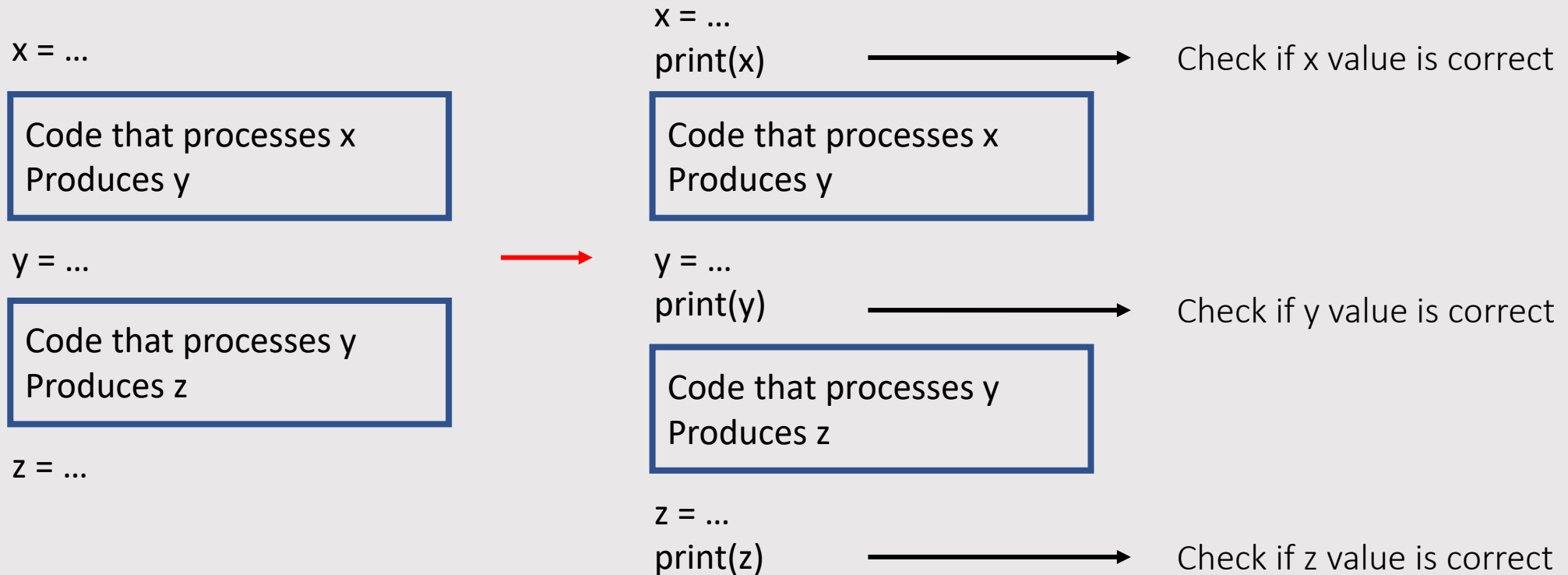
`print("done running block 2")` → Runs when code block 2 doesn't produce an error

Code block 3

`print("done running block 3")`

•
•
•

BASIC DEBUGGING WITH PRINT()

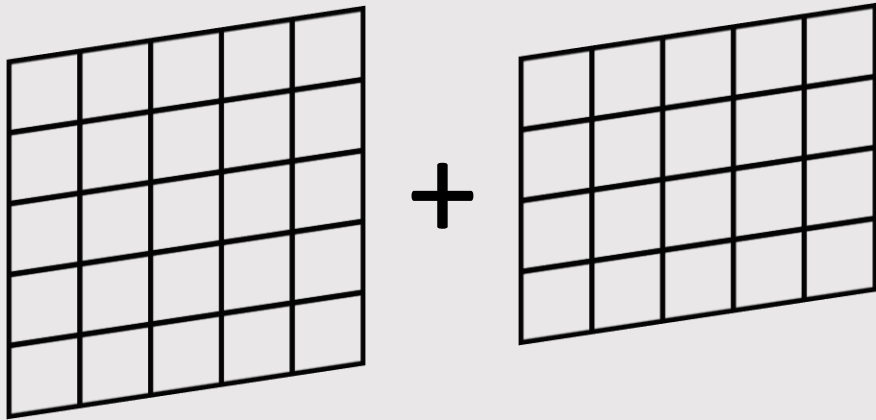


Original code

Debugging each step with print()

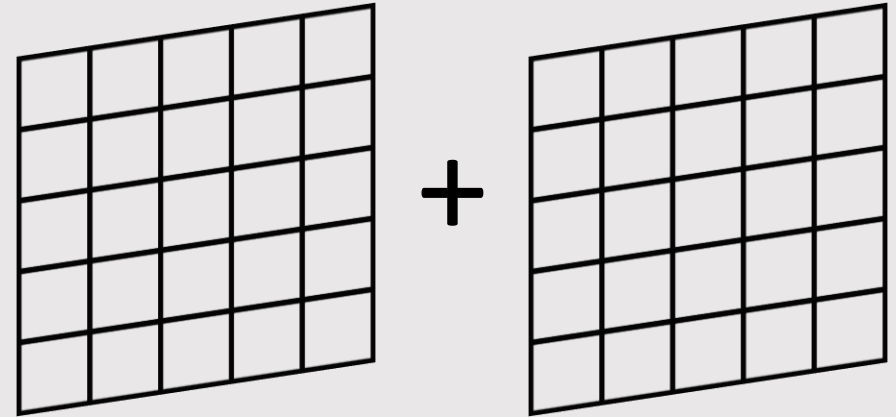
COMMON ARRAY MISTAKES – ARRAY ARITHMETIC

Dimension mismatch example
during array addition



arr_1.shape = (5, 5) arr_2.shape = (4, 5)

Correct operation



arr_1.shape = (5, 5) arr_2.shape = (5, 5)

```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-14-926f132f2a32> in <module>  
      1 arr_1 = np.ones((5,5))  
      2 arr_2 = np.ones((4,5))  
----> 3 np.add(arr_1, arr_2)
```

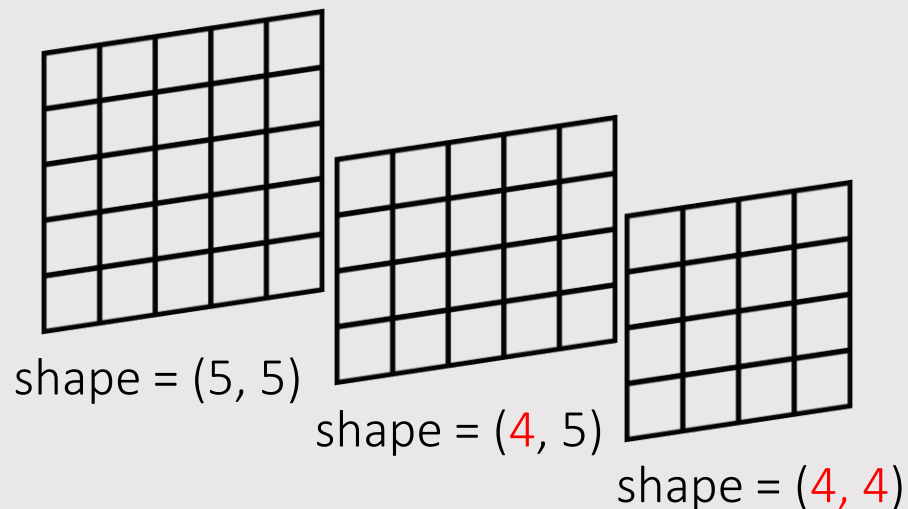
ValueError: operands could not be broadcast together with shapes (5,5) (4,5)

```
1 arr_1 = np.ones((5,5))  
2 arr_2 = np.ones((5,5))  
3  
4 np.add(arr_1, arr_2)
```

```
array([[2., 2., 2., 2., 2.],  
       [2., 2., 2., 2., 2.],  
       [2., 2., 2., 2., 2.],  
       [2., 2., 2., 2., 2.],  
       [2., 2., 2., 2., 2.]])
```

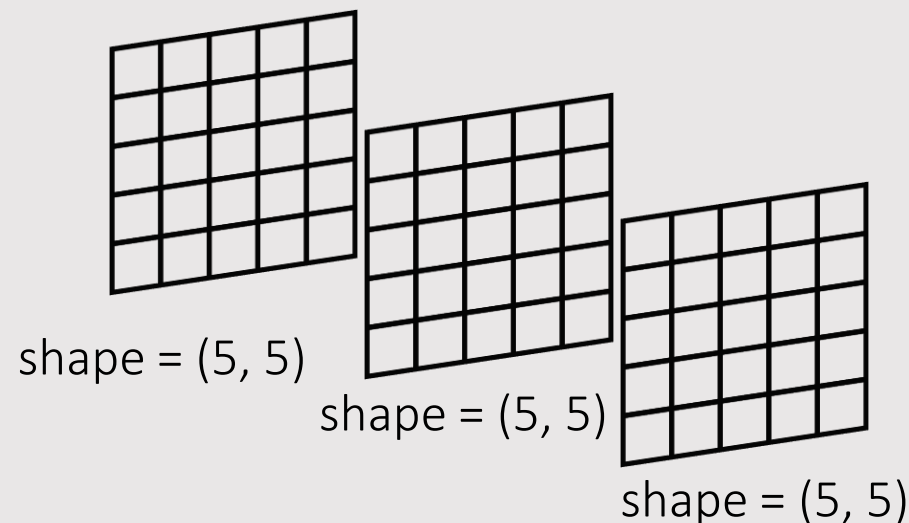
COMMON ARRAY MISTAKES – ARRAY STACKING

Dimension mismatch example
during array stacking



```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-16-328819f5fe4c> in <module>  
      3 arr_3 = np.ones((4,4))  
      4  
----> 5 np.stack([arr_1, arr_2, arr_3], axis = 2)  
  
<__array_function__ internals> in stack(*args, **kwargs)  
  
~\anaconda3\lib\site-packages\numpy\core\shape_base.py in stack(arrays, axis, out)  
    425     shapes = {arr.shape for arr in arrays}  
    426     if len(shapes) != 1:  
--> 427         raise ValueError('all input arrays must have the same shape')  
    428  
    429     result_ndim = arrays[0].ndim + 1  
  
ValueError: all input arrays must have the same shape
```

Correct operation



```
1 arr_1 = np.ones((5,5))  
2 arr_2 = np.ones((5,5))  
3 arr_3 = np.ones((5,5))  
4  
5 arr_combined = np.stack([arr_1, arr_2, arr_3], axis = 2)  
6 print(arr_combined.shape)
```

(5, 5, 3)