CS180/280A Discussion 1

[GSI, Date]

Slides adapted from: Justin, Chung Min, Brent

Discussions return this year!

- Both practical + conceptual; in-scope for exams 🫣

Discussions return this year!

- Both practical + conceptual; in-scope for exams 🫣
- Please give us feedback!

Discussions this year!

- Both practical + conceptual; in-scope for exams 🫣
- All new material this year, give us feedback!
- **Minimal laptop**

Discussions this year!

- Both practical + conceptual; in-scope for exams 🫣
- All new material this year, give us feedback!
- Minimal laptop
- Collaborative! Talk to people!

1 minute icebreaker!

Turn to 2-3 people around you

- Name
- Favorite fruit

Here’s mine

Reminders

Project 1 due in one week! Fri **9/15** 11:59pm

Worksheets online:

Topics

Discussion	Topic	Materials
Week 1	Python & NumPy Fundamentals for Computer Vision	Main sheet | Challenge | Solutions

Today: we’re talking about

NumPy

Today’s goal

Refresh + review NumPy fundamentals through the lens of pixels

This week is a bit weird, usually I won’t talk so much

Today’s goal

Refresh + review NumPy fundamentals through the lens of pixels

This week is a bit weird, usually I won’t talk so much

Probably will only get through Q1 and Q2

Today’s goal

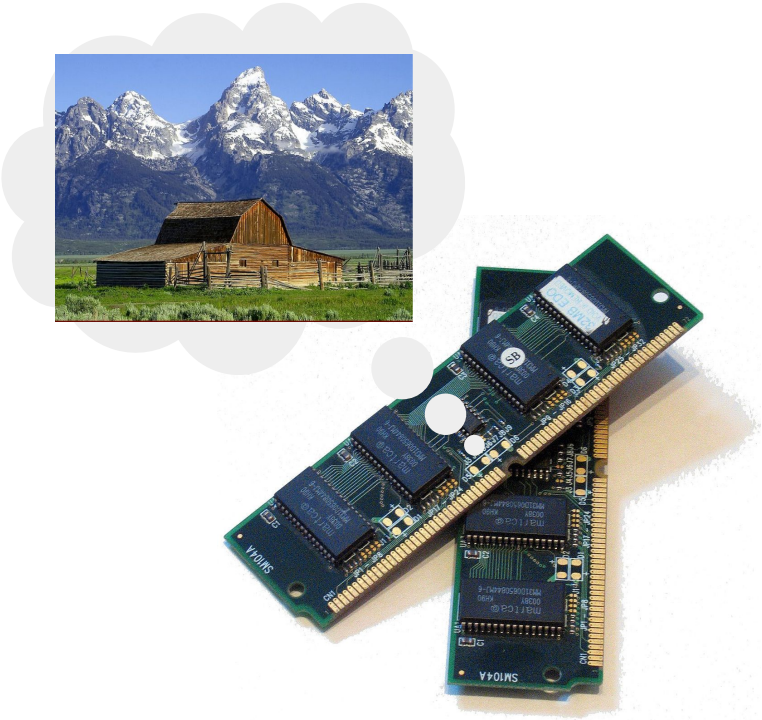
Refresh + review NumPy fundamentals through the lens of pixels

This week is a bit weird, usually I won’t talk so much

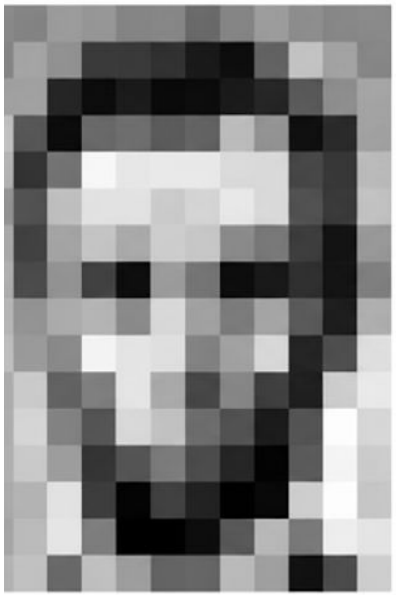
Probably will only get through Q1 and Q2

Quick poll: have you used NumPy before?

Why NumPy? Visual world 🤝 Computers

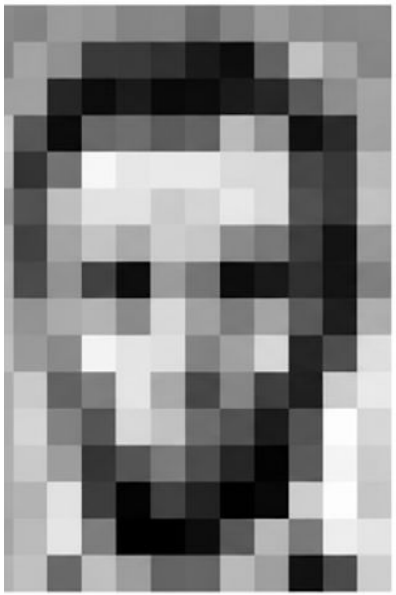


An image is an **array** of pixels!



157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	158	181
206	109	5	124	141	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
206	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	236	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	200	123	207	177	121	122	200	175	13	96	218

What is an array?

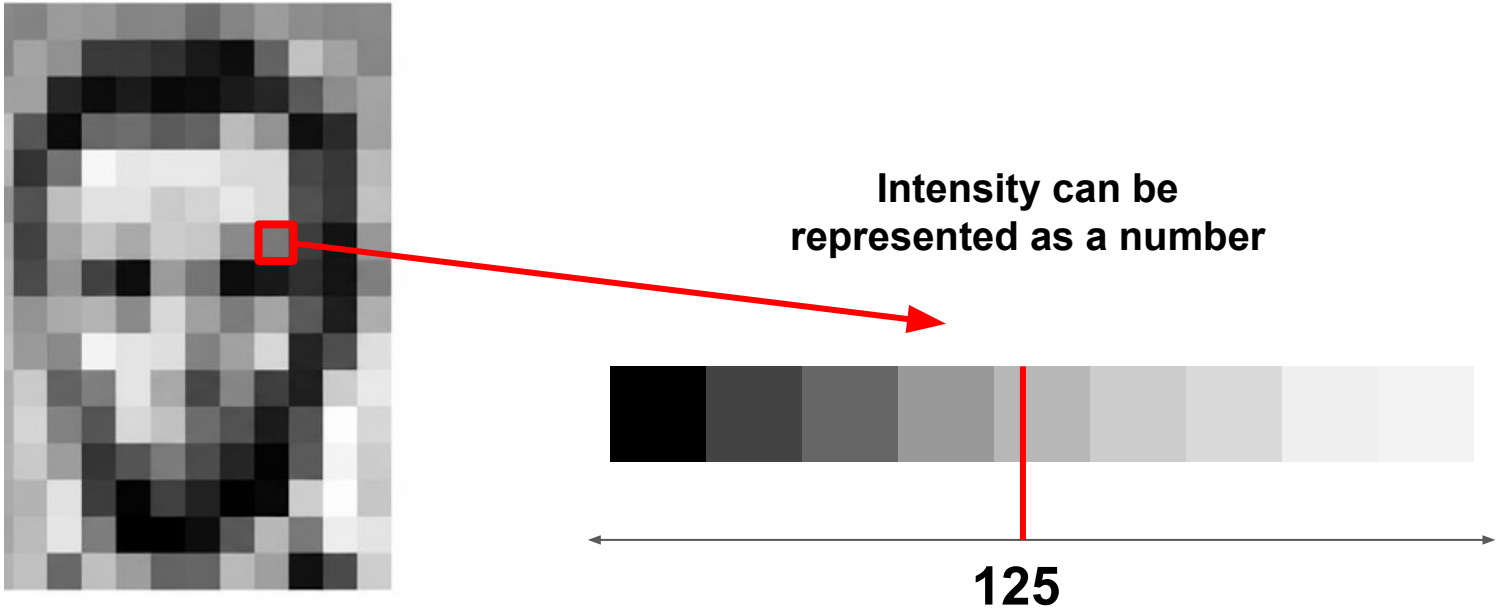


157	153	174	166	160	162	129	161	172	161	165	165
165	182	163	74	75	62	33	17	110	210	180	164
180	180	50	14	34	5	10	33	48	106	155	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	106	207	253	253	214	230	239	228	98	74	206
188	88	179	209	165	215	211	158	139	75	20	188
189	87	165	84	10	168	154	11	31	62	22	188
199	168	191	192	158	227	178	143	182	105	36	190
205	174	155	252	236	231	148	178	228	43	95	234
190	216	116	149	236	187	86	160	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	234
190	214	173	66	158	143	56	50	2	109	249	218
187	156	236	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	122	200	175	13	96	218

I think of these as *nested lists*

```
image = [
  [157, 153, 174, 166, 160, 162, 123, 161, 172, 161, 166, 165],
  [165, 182, 163, 74, 75, 62, 33, 17, 110, 210, 180, 164],
  [180, 180, 50, 14, 34, 5, 10, 33, 48, 106, 155, 181],
  [206, 109, 5, 124, 131, 111, 120, 204, 166, 15, 56, 180],
  [194, 68, 137, 251, 227, 239, 239, 228, 227, 87, 71, 201],
  [172, 106, 207, 233, 253, 214, 230, 239, 228, 98, 74, 206],
  [188, 88, 179, 209, 165, 215, 211, 158, 139, 75, 20, 188],
  [189, 87, 165, 64, 10, 168, 154, 11, 31, 62, 22, 188],
  [199, 168, 191, 183, 158, 227, 178, 143, 182, 105, 36, 190],
  [205, 174, 155, 252, 236, 231, 148, 178, 228, 43, 95, 234],
  [190, 216, 116, 149, 236, 187, 86, 160, 79, 38, 218, 241],
  [190, 224, 147, 108, 227, 210, 127, 102, 36, 101, 255, 234],
  [190, 214, 173, 66, 158, 143, 56, 50, 2, 109, 249, 218],
  [187, 156, 225, 75, 1, 81, 47, 0, 6, 217, 255, 211],
  [183, 202, 237, 145, 0, 0, 12, 108, 200, 138, 243, 238],
  [195, 206, 123, 207, 177, 121, 123, 200, 175, 13, 95, 218]
]
```

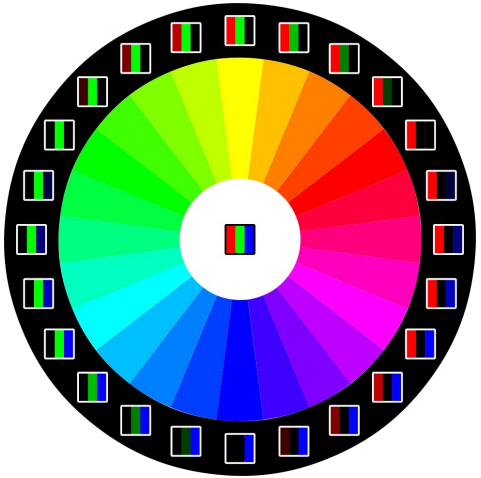
What is an array?



RGB images: shape (H, W, C), where C=3

"List of list of lists" 😔

3 channels: red, green, blue

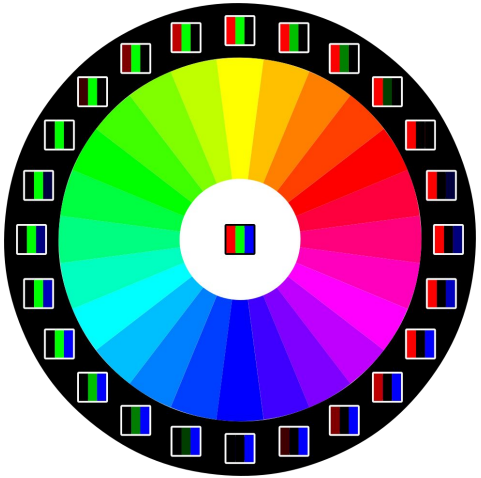


RGB images: shape (H, W, C), where C=3

A *“pixel”* is one (i,j) coordinate: ie arr[0,0], which has all 3 RGB *channels*

"List of list of lists" 😔

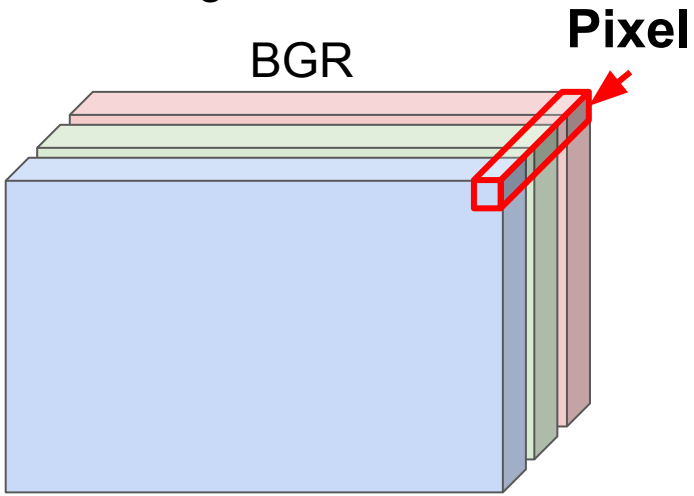
3 channels: red, green, blue



Pixel layout in an array

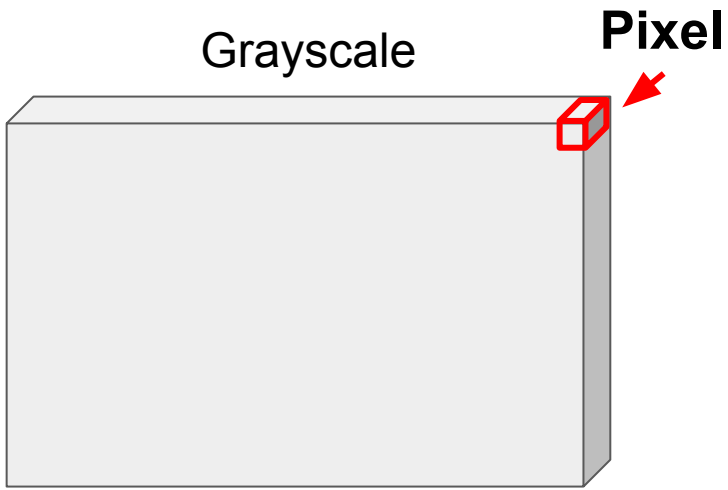
```
>>> img = cv2.imread("img.jpg") # shape (1080, 1920, 3)
```

Pixels stored along ***channels***.



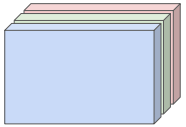
`(H,W,C)` `C=3` "channel-last"

… `or` `(C,H,W)` "channel-first"



`(H,W)` or `(H,W,1)`

Singleton dim.



Let's start: Color channel manipulation (1.3 Slicing)

`>>> img`

Original Image

150
101
105

`>>> img[...,0]`

Blue Channel

105

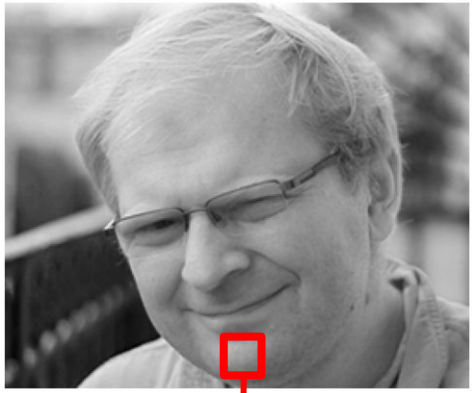
`>>> img[...,1]`

Green Channel

101

`>>> img[...,2]`

Red Channel

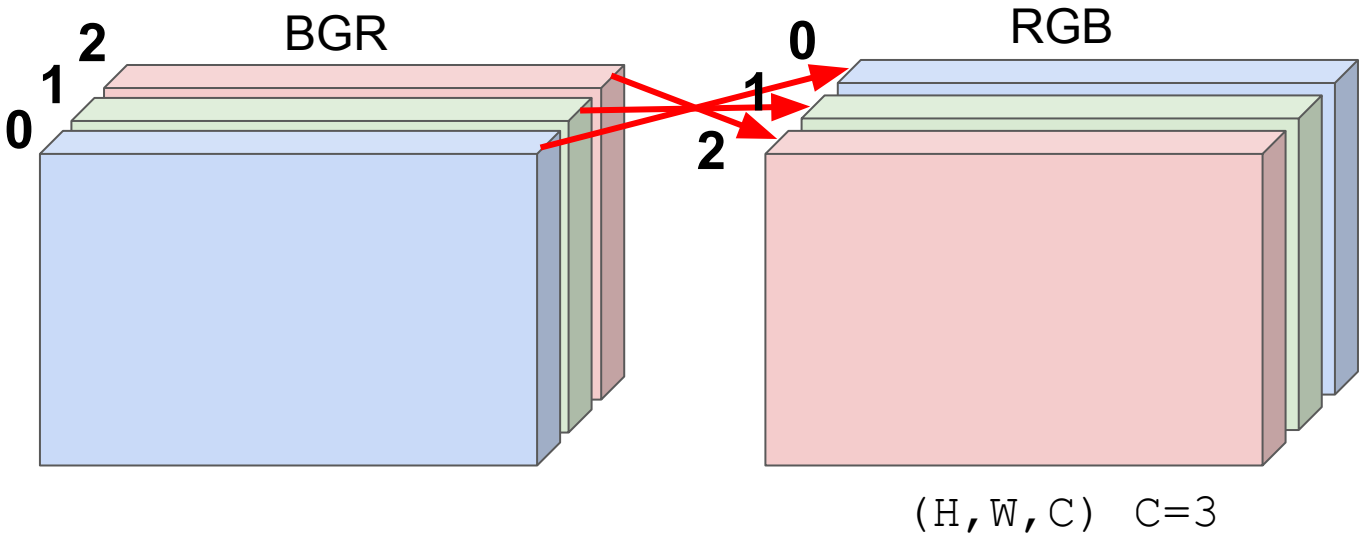


150

*Reminds of Proj 1!

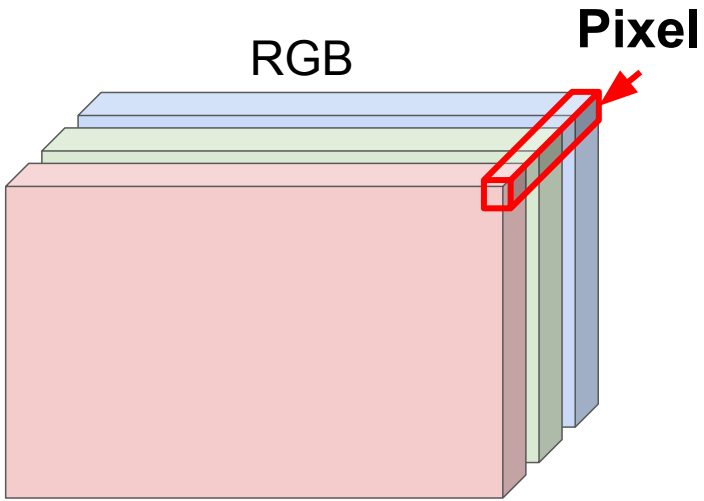
BGR => RGB (1.3 Slicing)

```
>>> img = img[:, :, [2,1,0]]
```



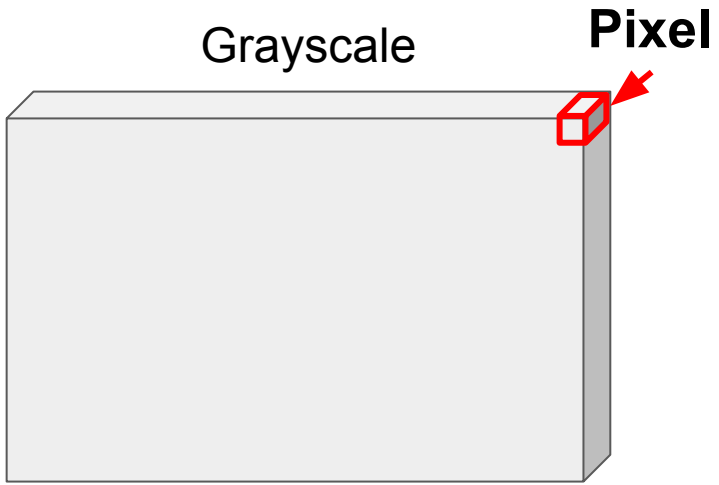
My mental model: a “volume” of data

Shape is how big each side of this volume is



“Channel-last” `(H,W,C)` `C=3`

… `or (C,H,W)` “channel-first”



`(H,W)` **or** `(H,W,1)`

Singleton dim.

Can be many dimensional (“N-D Arrays”)

Videos are *batches* of images! “*List of list of list of lists*”

`video`
`(T,H,W,C)`

`video[0]` **`video[5]`** **`video[11]`** **`video[15]`**

`(H,W,C)`

…or a batch of videos (**5**-D array!)

Point: getting comfortable with N-D arrays is ***very important*** for CV!

batch

`(B,T,H,W,C)`

video[0] **video[5]** **video[11]** **video[15]**

`(T,H,W,C)`

How do we get pixels into python?

Python demo on course site for basic image loading + saving + ops!

```
image = iio.imread('your_image.jpg')

print(f"Image shape: {image.shape}")
print(f"Image dtype: {image.dtype}")
print(f"Min value: {image.min()}, Max value: {image.max()}")
```

✓ 0.0s

```
Image shape: (300, 451, 3)
Image dtype: uint8
Min value: 0, Max value: 231
```

Try Q1!

- Shape + Dtype
- Stack vs. Concatenate
- Array Slicing
- Singleton Dims

Refresher on NumPy Basics

shape

```
np.array([1, 2, 3]) # 1D array! shape = (3,)
np.array([[1, 2], [3, 4]]) # 2D array! shape = (2,2)
np.full((3, 2), 7) # 3x2 array of 7s!
```

dtype

Default is np.float64! (double precision). You should also know: np.float32, np.uint8

Be careful of over/underflow! Remember uint8(256) = 0!

```
arr2 = arr.astype(np.float32) # convert array dtype
```

Joining arrays

```
np.stack([arr1, arr2],axis=0) # creates new 0 axis -> shape = (2,...)
np.concatenate([arr1,arr2],axis=0) # uses existing 0 axes
```

Slicing! For 2D 10x10 array.

```
arr[0]  # first row, shape (10,)
arr[:3] # first 3 rows, shape (3,10)
arr[:, :3] # first 3 columns, shape (10,3)
arr[arr==3] = 0 # sets all elements equal to 3 to 0
```

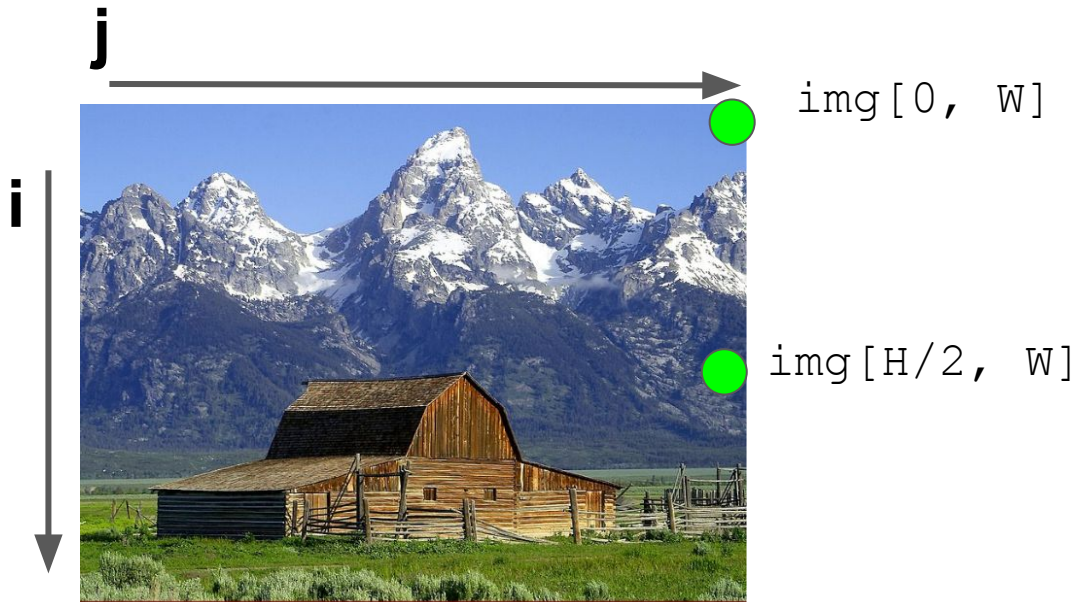
Singleton dims

Arrays can have shapes with size 1, called a “singleton dimension”

```
np.array([1]) # shape (1,)
np.array([[1]]) <---- extra list! # shape (1,1)!
np.expand_dims(arr, 0) # adds a singleton dim at index 0
```

Image stuff! Indexing conventions

Index into image axes like ij in a matrix, **not like** xy in a coordinate plane!



Many image ops are just array ops!

ren **alyosha**

`np.concatenate([ren,alyosha],axis=0)`

`np.concatenate([ren,alyosha],axis=1)`

🤔

🤔

Many image ops are just array ops!

ren **alyosha**

`np.concatenate([ren,alyosha],axis=0)`

`np.concatenate([ren,alyosha],axis=1)`

Many image ops are just array ops!

```
>>> left_half = img[:, :100, :]

>>> bottom_half = img[100:]
```

Many image ops are just array ops!

`img`

`img[...,0]`

`img[...,1]`

`img[...,2]`

Many image ops are just array ops!

```
>>> bgr = np.flip(rgb, axis=2)
```

Red | Green | Blue

Blue | Green | Red

Q2: Pixels!

Try the problems, discuss with your neighbors

Swap and Flip

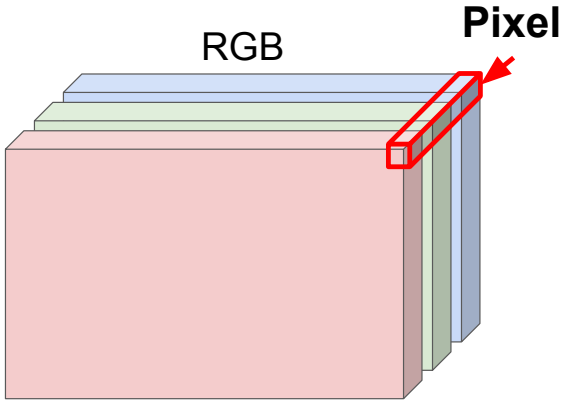
Swapping axes

```
arr = np.ones((3,10,5))
arr2 = arr.transpose(1,2,0) # re-order axes, shape (10,5,3)
```

np.flip

Reverse the order of elements in an axis

```
A = np.array([[1,2,3]]) # shape (1,3)
np.flip(A, axis=1) # shape (1,3), values [3,2,1]
np.flip(A, axis=0) # same array as A
```



"Channel-last" `(H,W,C)` `C=3`

… `or (C,H,W)` "channel-first"

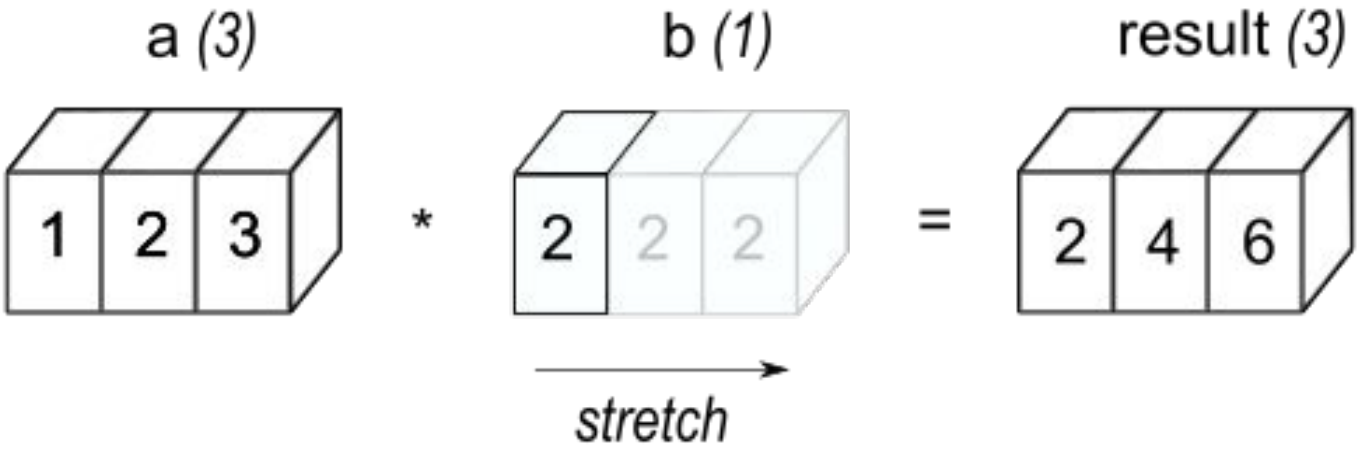
What happens when shapes don’t match?


```
>>> img # shape (1080, 1920, 3)

>>> brighter_img = img + np.array([100,100,100])  # shape (1080,1920,3) + (3,) ??
```

Broadcasting: automatically *repeat* elements to match!

```
>>> a = np.array([1,2,3])  # shape (3,)
>>> b = np.array(2)        # shape (1,)
>>> print(a*b) # [2.0,4.0,6.0] shape (3,)
```



Broadcasting

Rule for figuring out behavior:

1. **Line up** array shapes starting from the **right**
2. **For each axis:**
    a. If shapes match, continue to the left
    b. If shapes don’t match and one is 1, stretch its values to fit the larger
    c. If shapes don’t match and *neither* are 1, throw an error

Broadcasting: try problem 3!

Rule for figuring out behavior:

1. **Line up** array shapes starting from the **right**
2. **For each axis:**
   a. If shapes match, continue to the left
   b. If shapes don't match and one is 1, stretch its values to fit the larger
   c. If shapes don't match and *neither* are 1, throw an error

np.arange(3) => [0, 1, 2]
.reshape(3, 1) => [[0], [1], [2]]

Vectorization

"Vectorization" means writing things with native NumPy operations rather than for loops

Much faster when possible!

Native C (low overhead) vs Python (high overhead)

SLOW

```
for i in range(H):

    for j in range(W):

        out[i,j,:] = (a[i,j] + b[i,j])/2.0
```

```
(ren + alyosha)/2.0
```

Fast

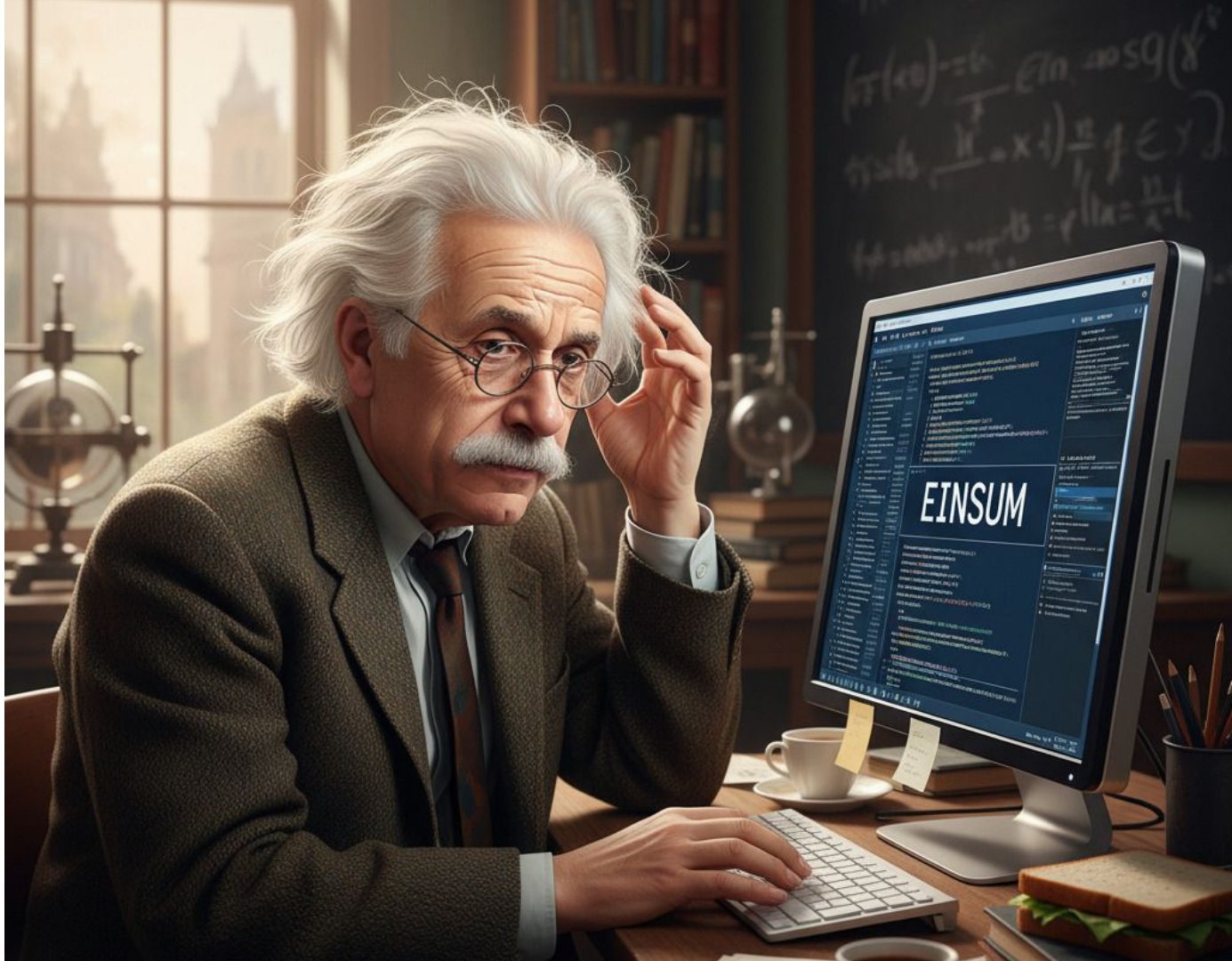
Vectorization: do problem 4!

“Vectorization” means writing things with native NumPy operations rather than for loops

Thanks for coming! Please provide feedback!

https://bit.ly/4yazC9Y

Einsum



`einsum` examples!

```
>>> a = np.arange(4)  # (4,)
>>> b = np.arange(4)  # (4,)
```

```
>>> np.einsum('i,i->i', a, b)
>>> np.einsum('i,j->ij', a, b)
>>> np.einsum('...i,...i->...', a, b)
```

```
array([0, 1, 4, 9])     # (4,)
```

```
array([[0, 0, 0, 0],
       [0, 1, 2, 3],
       [0, 2, 4, 6],
       [0, 3, 6, 9]])  # (4, 4)
```

```
np.int64(14)            # (,)
```

Manipulating shapes

Many times we want to shuffle the order of axes or combine them.

Remember arrays are *row-major!*