Images as ”2-D” structures

Index		Columns			
		0	1	2	…
Rows	0	(108, 85, 71)	(149, 131, 121)	(210, 201, 196)	…
	1	(106, 87, 73)	(190, 177 168)	(220, 215, 211)	…
	2	(103, 87, 74)	(208, 199, 190)	(223, 219, 216)	…
	…	…	…	…	…

Pixel at (2,1) with RGB color components of (208, 199, 190)

Complexity doesn’t arise from using more complex features of Python (I haven’t been holding out on you), but on combining the features we have already learned about. Even the libraries we have used that offer “complex” capabilities, are encapsulating operations implemented with the same core features we have learned about. Our focus this week is how we can apply what we have learned to more complex applications.

We can think of an image as a single entity, or a 2-D (or even 3-D) structure of pixels, i.e., the image has a width and a height, and each pixel has a position described by specific row and column. For our purposes, let's assume that (0, 0) is the top-left corner of the image.

In this case the pixel at row index 2, column index 1 as the color (208, 199, 190), as described by its RGB color components. That is, we model/store each pixel as a 3-tuple. Why might we think of an image as a 3-D structure? Rows, columns and color components within a pixel!

Retrieving and modifying pixels

```
img = Image("https://.../linderman.jpg")
pix = img.get_pixel(100, 120) # row 100, column 120
pix.red += 100
img.set_pixel(100, 120, pix)
img.show()
```

```
class Pixel:
  def __init__(self, r, g, b):
    self.red = r
    self.green = g
    self.blue = b
```

Today we will use starter code containing an `Image` class for storing and manipulating images as 2-D structures. This class provides a simplified wrapper around the Pillow Python Imaging Library. `Image` loads an image from a URL and provides methods for getting and setting pixels at specific rows and columns. For example, the this loads an image from a URL, "red shifts" one pixel by adding 100 to the red-component, then shows it on the screen.

The `get_pixel` method returns a Pixel object, shown here, with instance variables for the tree color components (stored as integers).

This only shifts one pixel, if we were going to shift the entire image, like shown, we would need some type of loops. What kind? For loops…

Interlude: Nested loops and ”linear indices”

```
ROWS=3
COLS=4

for row in range(ROWS):
  for col in range(COLS):
    print("Row:",row,"Col:",col,"Linear:",row*COLS+col)



for i in range(ROWS*COLS):
  row = i // COLS
  col = i % COLS
  print("Row:",row,"Col:",col,"Linear:",i)
```

```
Row: 0 Col: 0 Linear: 0
Row: 0 Col: 1 Linear: 1
Row: 0 Col: 2 Linear: 2
Row: 0 Col: 3 Linear: 3
Row: 1 Col: 0 Linear: 4
Row: 1 Col: 1 Linear: 5
Row: 1 Col: 2 Linear: 6
Row: 1 Col: 3 Linear: 7
Row: 2 Col: 0 Linear: 8
Row: 2 Col: 1 Linear: 9
Row: 2 Col: 2 Linear: 10
Row: 2 Col: 3 Linear: 11
```

A common task is to perform an operation with or to each pixel. We will need a loop, and since the number of pixels is known at the start of the loop, a for loop is a natural choice. Performing an operation to each pixel can be implemented by performing an operation for all possible combinations of row and column positions, i.e., (0,0), (0, 1) … (2, 2) in the 3×3 example above. All combinations of multiple sequences, in this case, the row and column indices, is readily implemented with nested loops.

Here the outer loop iterates over all rows, while the inner loop iterates over all columns. Since the loops are nested, we only advance to the next row, the next iteration of the outer loop, after we have iterated through all columns (for that row). In the output we should observe all combinations of row and column indices. We also see a common pattern for computing a “linear” index, that is an index if we traversed all the elements in row-order. The last is common way of iterating through a “2-D” structure stored in a list or other “1-D” structure. The loop belows the reverse mapping, i.e., translating from a linear index to the associated rows and columns.

When might we use one vs. the other? It depends on how our data is stored, and whether the row/column structure matters to us. For example, do we want to do something after iterating through each row? If so, the first structure will likely be a better choice.

Which of the following loop structures could we use to “red shift” the entire image?

A

```
for row in range(img.get_height()):
  for col in range(img.get_width()):
    pix = img.get_pixel(row, col)
    …
```

B

```
for row in range(img.get_width()):
  for col in range(img.get_height()):
    pix = img.get_pixel(row, col)
    …
```

C

```
row=0
col=0
for row in range(img.get_height()):
  pix = img.get_pixel(row, col)
  …
for col in range(img.get_width()):
  pix = img.get_pixel(row, col)
  …
```

D

```
for i in range(img.get_height()*img.get_width()):
  row = i // img.get_height()
  col = i % img.get_height()
  pix = img.get_pixel(row, col)
  …
```

Answer: A

We want to proceed all combinations of row and column indices. All combinations of multiple sequences, in this case, the row and column indices, is readily implemented with nested loops.

What is the problem with?
B: row and columns are reversed. If the image is not square, we might access a pixel that doesn’t exist.
C: Only processing a single row and column (since loops are not nested)
D: This is close. We will execute one loop iteration for each pixel, but the translation from ”linear” index to row and column is incorrect. The divisor, should be the width, i.e., get_width(). That will produce the correct row and column

Match the data structure for storing the image with a possible implementation for `get_pixel(row, col)`

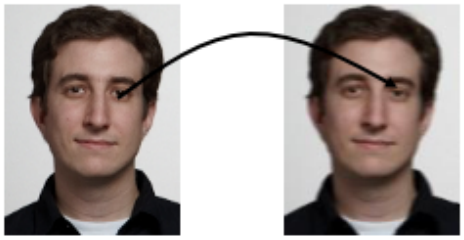
1. List of lists 2. Single list 3. Dictionary	I. Not a possible data structure for an image II. `return image[row*WIDTH+col]` III. `return image[(row, col)]` IV. `return image[row][col]`

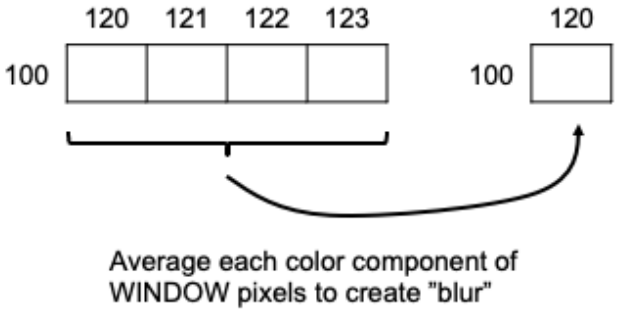
A. 1-IV, 2-II, 3-I
B. 1-I, 2-II, 3-III
C. 1-II, 2-III, 3-IV
D. 1-IV, 2-III, 3-II
E. 1-IV, 3-III, 2-II

Answer: E

The list of lists would look like `[[ … row 0 … ], […  row 1 ], …]` and thus be accessed by first retrieving the list for the row, e.g., `image[row]`. That expression evaluates to a list, then we use the indexing operator again to select the pixel in that list associated with the specific column in that row. For a single list, we use the "linearized" indexing we saw earlier to map the row and column to a specific index in the row-ordered list of pixels. For the dictionary, we are using tuples of row-column indices as the keys.

Nesting more loops! Implementing horizontal blur





In the "red-shift" example above, we perform a "fixed" transformation for each pixel. But we could imagine transformations that might be variable, i.e., require a loop in some way. For example, considering horizontal "blur", where each pixel is the average of itself and `WINDOW-1` pixels to its right. Specifically, if `WINDOW` was 4, each pixel would be the average of itself and the the 3 pixels immediately to its right. Since `WINDOW` could change, we would want to implement with a loop.

Which of the following loop bodies could we use to implement horizontal blur?

```
WINDOW=4
for row in range(img.get_height()):
  for col in range(img.get_width()-WINDOW+1):
```

Common outer loops for all answers

A

```
px = img.get_pixel(row, col)
for i in range(WINDOW):
  blur_px = img.get_pixel(row,col+i)
  px.red += blur_px.red
  …
img.set_pixel(row,col,Pixel(px.red//WINDOW,…))
```

B

```
px = img.get_pixel(row, col)
for i in range(WINDOW):
  blur_px = img.get_pixel(row,col+i)
  px.red += blur_px.red
  …
img.set_pixel(row,col,Pixel(px.red,…))
```

C

```
px = Pixel(0, 0, 0)
for i in range(WINDOW):
  blur_px = img.get_pixel(row,col+i)
  px.red += blur_px.red
  …
img.set_pixel(row,col,Pixel(px.red//WINDOW,…))
```

D

```
px = img.get_pixel(row, col)
for i in range(1, WINDOW):
  blur_px = img.get_pixel(row,col+i)
  px.red += blur_px.red
  …
img.set_pixel(row,col,Pixel(px.red,…))
```

Answer: C

Here we are correct averaging all pixels. The problem with A is we are double counting the original pixel, while in B we are both double counting and missing the division to compute the average. Answer D doesn't double count but is missing the division to compute the average.

Why is the limit for the inner loop img.get_width()-WINDOW+1 (and not img.get_width())? With the latter we would attempt to access pixels beyond the boundary of the image. For simplicity we ignore the right most pixels in the image.

Putting it all together, face “averaging”

1. Download starter code from class notes page
   Make sure to install pillow module. Use Thonny “Tools”→ "Manage Packages" menu command, search for and install pillow.
2. Complete `average` function to generate average of multiple images (similar to blur, compute average of each color component). Assume all images are the same size.
3. Experiment with provided faces
4. Answer the following questions:
   - Do certain sets look significantly different from others
   - Are there faces that seems particularly important to the final output?

Adapted from: Peck, E, et al., https://ethicalcs.github.io

The bulk of today is going to adapting the ideas we have seen to implement image averaging, specifically with faces. That is, I want you to download the starter code and implement the average function to compute an average image from a list of input images by average each color component. That is the red component of an output pixel is the average of the red components for the red components of the same pixel in all in the input images, and the same for the green and blue components. Once you have done so. Experiment with averaging different subsets of the provided faces and discuss the following questions.

What did you observe?

```
def average(in_files, out_file):
  imgs = []  # Load images
  for url in in_files:
    imgs.append(Image(url))

  first_img = imgs[0] # Update 1st image with avg. pixels
  for row in range(first_img.get_height()):
    for col in range(first_img.get_width()):
      avg_pix = Pixel()
      for i in range(len(imgs)):
        add_pix = imgs[i].get_pixel(row, col)
        avg_pix.red += add_pix.red
        avg_pix.green += add_pix.green
        avg_pix.blue += add_pix.blue
      first_img.set_pixel(row, col,
        Pixel(avg_pix.red//len(imgs),avg_pix.green//len(imgs),avg_pix.blue//len(imgs))
      )

    first_img.save_image(out_file) # Save modified image to file
```

The resulting average face depends on the inputs we use. Adding or removing faces can change the result!

The same is true for machine learning based methods we use for facial detection (is a face present), facial verification (does this face match a particular known face) and facial recognition (does this face match any of a large set of previously observed faces). In these “supervised” approaches developers assemble training data sets of images we specific labels – has a face, has a face, doesn’t have a face – and uses that data to learn features associated with those labels. Those features will depend on the training data used. If that training data does not contain many (any) faces from underrepresented groups, the resulting algorithm is unlikely to be accurate when used with new examples from those groups. And that is exactly what researchers have found when they test those tools. The gender shades project tested commercial gender classification tools (predict gender from a facial photograph) and observed that accuracy was lower for darker female faces than lighter male faces (by 20-34 percentage points).

When we talk about “Algorithmic Bias”, this is problem we are describing. Disparate results for automated tools for different racial groups, gender identities, etc. It might seem that the underlying algorithms are “neutral”. For example, nothing in our averaging code was specific to a particular image, gender, etc. But as we observed the choice of inputs to that function can have an impact on the results. And any downstream uses of those average faces would reflect those choices. That is, as the

Gender Shades authors write “Automated systems are not inherently neutral. They reflect the priorities, preferences, and prejudices [of the developers].” This isn’t to suggest that the developers are actively trying generate discriminatory outcomes, but more that is the unintentional result of the choices the developers make (for training data, for evaluation benchmarks, etc.). The first of a very famous set of “laws of technology” (by historian Melvin Kranzberg) captures this idea: “Technology is neither good nor bad; nor is it neutral.”.

The technology creator, that is you, wields tremendous power, whether they realize it or not. The choices they make, intentional or otherwise, can have substantial impact on the results of using the tools they build. Because of how fast and easily technology can spread (incorporating facial recognition into an application could be as simple as downloading a piece of software), those impacts can be felt far and wide.

What does this mean for us? That we raise our expectations for the tools we use, that is technology doesn’t work unless it works for everyone. Think about a time that you needed to use an application that felt like it wasn’t designed for you. And I don’t just mean bad design (e.g., Banner), but an application that seemed to be designed for someone else. A common example is names… There is a famous blog post “Falsehoods Programmers Believe About Names” that has 40 false assumptions, such as “People have exactly N names, for any value of N.” There is a wide range of naming approaches in the world, of which most people will only encounter a small subset.

The author writes about an example of individual with a hyphenated last name “…complaining about how a computer system he was working with described his last name as having invalid characters. It of course does not, because anything someone tells you is their name is — by definition — an appropriate identifier for them. John was understandably vexed about this situation, and he has every right to be, because **names are central to our identities**, *virtually by definition*.”

Having your name is rejected is more than just frustrating. ”Authentic name” policies can be discriminatory (e.g., Native American names are rejected, as has happened) or if using your “real” name would expose you to harm. ”In September 2014, hundreds of transgender people and drag queens had their Facebook accounts shut down after they were reported as fake. [As individual users started to realize that this was happening to people across their communities, they began to organize, lodge complaints and protest. Though Facebook apologized to the lesbian, gay, bisexual and transgender (LGBT) and drag queen communities after the issue garnered media attention, as of today there has not been meaningful change in Facebook’s policy. Rather,] the company continues to require users to provide onerous documentation to verify names that are flagged by users or Facebook as potentially unauthentic. This is problematic for a range of communities, including trans and gender non-conforming people, drag queens, survivors of violence, Native Americans and more.”

So again, think about a time that you needed to use an application that felt like it wasn't designed for you. We don't want to anyone to feel that way. And we should do our work with that goal in mind. What I hope you will take away from today is that the work we do in here is not somehow separate from the world, but very much a part of it. Thus, it can reflect both reflect society's ills, however defined, *and* be a vehicle, a powerful one, for effecting change. I am not trying to encourage you to make any particular choice or choices. The choices you make will and should be personal. Instead, I want you to remember that technology is not neutral and will, can, reflect your values. So don't put your values down when you pick up your keyboard.

Further reading/watching:

- https://www.youtube.com/watch?v=rWMLcNaWfe0
- https://www.acm.org/binaries/content/assets/public-policy/ustpc-facial-recognition-tech-statement.pdf
- https://www.ted.com/talks/joy_buolamwini_how_i_m_fighting_bias_in_algorithms
- http://gendershades.org
- https://www.kalzumeus.com/2010/06/17/falsehoods-programmers-believe-about-names/
- https://media.business-humanrights.org/media/documents/files/documents/Sarah_Gunther_Facebook_Real_Name_Policy.pdf
- https://www.userinterviews.com/blog/design-failure-examples-caused-by-bias-noninclusive-ux-research

Example of technology as a societal enabler (not disabler)

U.S. DIGITAL SERVICE

"Recruit top designers, engineers, product managers, and digital policy experts. Pair these digital experts with our nation's leading civil servants. Deploy these teams to address some of the most critical government services, together."

"Too often, outdated tools, systems, and practices make interacting with the government cumbersome and frustrating. The challenges behind the rollout of HealthCare.gov made clear that accessing government services should be as easy as ordering a book online. Founded by President Obama in August of 2014, the U.S. Digital Service brought together the best engineering, design, and government talent to change our government's approach to technology. We planned to hire ten people for three critical national priorities: modernizing immigration, Veterans' benefits, and HealthCare.gov. During the 2015 State of the Union address, we launched an online application to join the team. We worried if ten people would even apply. 1000 did."

I think of it is a little like the Peace Corps, but for programming. The USDS grew out of the "tech surge", a group of engineers from Google and elsewhere that were recruited to help make HealthCare.gov more robust and usable during its initial and very troubled deployment. Mikey Dickerson, one of the main people involved the tech surge became the initial director of the USDS. It may seem like this is the kind of thing that only graduates of large Universities do, but that is not the case. Dickerson is a Pomona alum.