Workshop Notebook 3: Process a Single Image with Pillow

Mandatory Disclosures

1. This is a whirlwind introduction, not exhaustive instruction
2. All images are by courtesy of the University Archives at Texas State University: http://www.univarchives.txstate.edu
(http://www.univarchives.txstate.edu)
3. img_qc_workshop is licensed under the GNU General Public License v3.0,
   https://github.com/photosbyjeremy/img_qc_workshop/blob/master/LICENSE
   (https://github.com/photosbyjeremy/img_qc_workshop/blob/master/LICENSE)
4. Any and all code provided is done so without any warranty or expectation of support by Jeremy Moore, Todd Peters, or Texas State University

In [ ]: # import necessary modules
from pathlib import Path
from PIL import Image
import matplotlib.pyplot as plt
import img_qc.img_qc as img_qc

In [ ]: # matplotlib options
# magic that lets us plot directly in the notebook
%matplotlib inline

# parameters for matplotlib to increase our default figure size -- NOTE: figure sizes are in INCHES
plt.rcParams["figure.figsize"] = (20, 12)  # set as needed for your screen and eyes (width, height)

# on a high-dpi monitor this will increase the quality of plots on-screen
%config InlineBackend.figure_format = 'retina'

In [ ]: # set current_directory with Path function cwd() [Current Working Directory]
current_directory = Path.cwd()

# path to access our image
image_path = current_directory.joinpath('data/workshop-3/AS-36-T4-E9-1943-c2_0002.tif')

print(f'current_directory: {current_directory}')
print(f'image_path: {image_path}')

In [ ]: # open image with Pillow
image = Image.open(image_path)

# display image with MatPlotLib
plt.imshow(image)

Rotate Image

In [ ]: # rotate image 23 degrees clockwise
rotated_image = image.rotate(-23)  # negative angle is Clockwise

# show rotated image
plt.imshow(rotated_image)

Interpolation

Rotating in anything other than 90 degree increments will result in the interpolation of pixel data -- the computer has to make up new tones. There are different algorithms available for interpolation while rotating and there's a trade-off between performance (how intensive the calculations are) and quality.

shift+tab keyboard shortcut will show options

Performance < ----- > Quality

• Nearest Neighbor <> Bilinear <> Bicubic

Always TEST and VERIFY which algorithm is right for your use case.
In [ ]: # rotate with higher quality interpolation and expand size to not crop
rotated_image = image.rotate(-23, resample=Image.BICUBIC, expand=True) # negative angle is Clockwise

# show rotated image
plt.imshow(rotated_image)

**Resize with Image.resize()**

In [ ]: # resize image
image_resized = image.resize((500, 500)) # (width, height)

# show resized image
plt.imshow(image_resized)

**Resize with image.thumbnail() WARNING!!**

In [ ]: # create a copy of our image as Image.thumbnail() MODIFIES THE IMAGE IN-PLACE
thumbnail = image.copy()

# resize the image with thumbnail
thumbnail.thumbnail((500, 500))

# show the resized image
plt.imshow(thumbnail)

**Resize with img_qc.get_image_resized_pillow()**

In [ ]: # resize image
image_resized = img_qc.get_image_resized_pillow(image, width=500) # (width, height)

# show resized image
plt.imshow(image_resized)

**Jupyter Magic: %timeit**

Compare the speed of 3 different interpolation settings for rotate on image_resized

In [ ]: # create resample dictionary with names and Pillow resize methods
resample_dictionary = {'Nearest Neighbor': Image.NEAREST, 'Bilinear 2x2': Image.BILINEAR, 'Bicubic 4x4': Image.BICUBIC}

for name in resample_dictionary:
    print(name)

    # get resample_method by accessing the resample_dictionary with the key `name`
    resample_method = resample_dictionary[name]

    # call magic %timeit to time this line in our loop
    %timeit rotated_image = image_resized.rotate(-23, resample=resample_method, expand=True) # negative angle is Clockwise

**Crop**

As previously mentioned, digital images are graphs of pixels channels and intensities in a 2D plane.

The graph’s origin (0, 0) is the starting point of the image and the x-value increases with each pixel of width.

Even though we’re technically graphing into negative y-values, the y-value increases with each pixel of height and extends BELOW the image.

We will use this coordinate system to crop our image.

http://pillow.readthedocs.io/en/5.1.x/handbook/concepts.html#coordinate-system
In []: # Pillow needs a box with upper-left (x, y) values and lower-right (x, y) values to crop an image
image_cropped = image.crop(box=(0, 0, 500, 500)) # start in upper-left and go right 500 pixels, down 500 pixels
plt.imshow(image_cropped)

In []: # show image
plt.imshow(image)

In []: # crop image around page and color bar
image_cropped = image.crop(box=(2450, 800, 6450, 6000))

# print width & height
print(f'width: {image_cropped.size[0]}') # (width, height)
print(f'height: {image_cropped.size[1]}')

# show image
plt.imshow(image_cropped)

Channels
Our RGB image has 3 color channels that we can access using Pillow

In []: # split into separate channels
red_channel, green_channel, blue_channel = image_cropped.split()

figure, (red, green, blue) = plt.subplots(nrows=3, ncols=1, figsize=(18, 8)) # figsize is (width, height) in inches
red.imshow(red_channel)
red.set_title("Red Channel")
green.imshow(green_channel)
green.set_title("Green Channel")
blue.imshow(blue_channel)
blue.set_title("Blue Channel")

# some Matplotlib code that draws subplots close together while padding axes so they don't overlap
plt.tight_layout()

In []: # let's crop our cropped image down to the color bar to better see our different color channels
color_bar = image_cropped.crop(box=(0, 4600, 4000, 5200))
plt.imshow(color_bar)

In []: # split into separate channels
red_channel, green_channel, blue_channel = color_bar.split()

figure, (color, red, green, blue) = plt.subplots(nrows=4, ncols=1, figsize=(20, 15)) # figsize is (width, height) in inches
color.imshow(color_bar)
color.set_xticks([])
color.set_yticks([])
color.set_title("RGB Image")
red.imshow(red_channel)
red.set_xticks([])
red.set_yticks([])
red.set_title("Red Channel")
green.imshow(green_channel)
green.set_xticks([])
green.set_yticks([])
green.set_title("Green Channel")
blue.imshow(blue_channel)
blue.set_xticks([])
blue.set_yticks([])
blue.set_title("Blue Channel")

# plt.tight_layout()

Convert to Grayscale
When converting from color to grayscale a choice is made on how much to weigh the intensity of each color band on each pixel. By default, Pillow uses the ITU-R 601-2 luma transform:

\[
L(\text{luminance}) = \text{Red} \times \frac{299}{1000} + \text{Green} \times \frac{587}{1000} + \text{Blue} \times \frac{114}{1000}
\]

Convert to Bitonal with `Image.convert()`

When converting from color (RGB) or grayscale (L) to Bitonal (1) using `Image.convert` the default is to use the Floyd-Steinberg dither, which we DON'T want if we're converting images for OCR.

We can alternatively set the dither to `Image.NONE`, but this just converts non-zero to white according to the documentation.

Convert to Bitonal with `Image.point()`

We can convert our image to bitonal and choose a threshold using the `Image.point()` function. `Image.point()` does something to every pixel in the image. We can set a threshold value and use `lambda` with point to process each pixel in the image.

The documentation is wrong

If you look at our third image with `dither=Image.NONE`, it actually looks like it used a 50% threshold. Values up to 50% of the intensity range round down to 0 (black) and those above 50% round up to 255 (white).
plt.figure()
# show image
plt.imshow(image_bitonal_120)

**Process Page into Bitonal**

In [ ]: # crop scan down to the page
    image_page_crop = image_cropped.crop(box=(300, 125, 3800, 4600))

    # convert to grayscale
    image_page_gray = image_page_crop.convert(mode='L')

    # set threshold value
    threshold = 185

    # convert to bitonal with Image.point() method
    image_page_bitonal = image_page_gray.point(lambda pixel: pixel > threshold and 255)

    # show image
    plt.imshow(image_page_bitonal)