Isai B. Cicourel FOLLOW
Software Developer Engineer at Amazon Go

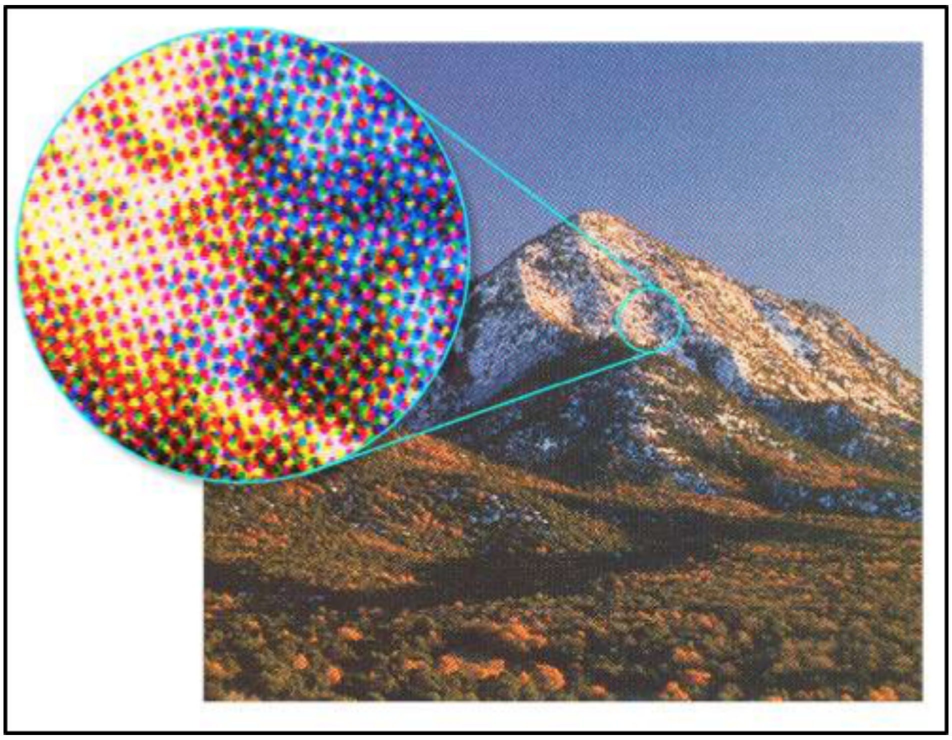
Image Manipulation in Python

Published Sep 21, 2016 Last updated Jan 18, 2017

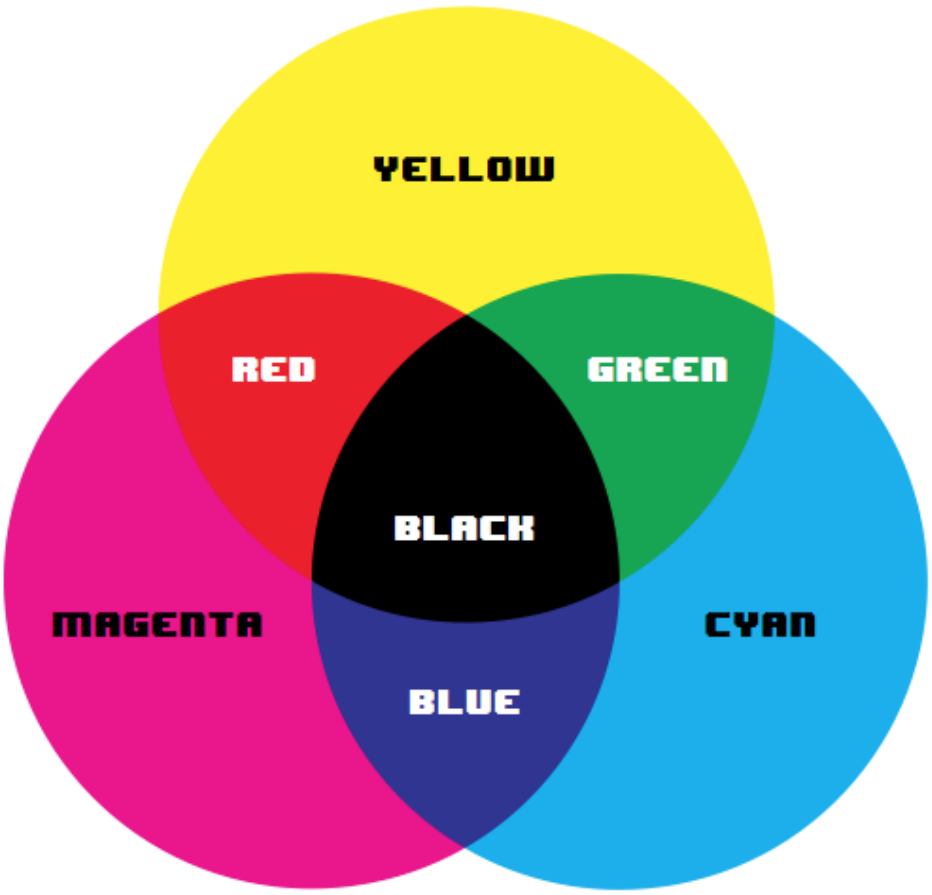
Color Space Background

CMYK Color Representation:

When we see a printed advertisement or poster, the colors are printed with color spaces based on the CMYK color model, using the subtractive primary colors of pigment **C**yan, **M**agenta, **Y**ellow, and blac**K**. This is also known as the four-colors print process.

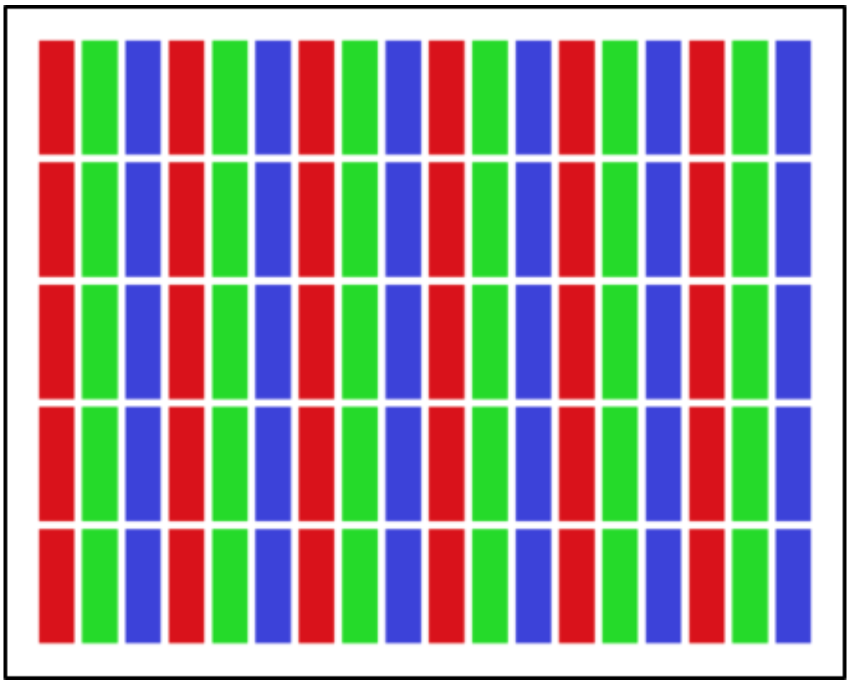


The "primary" and "secondary" colors in a four-color print process are exhibited below.



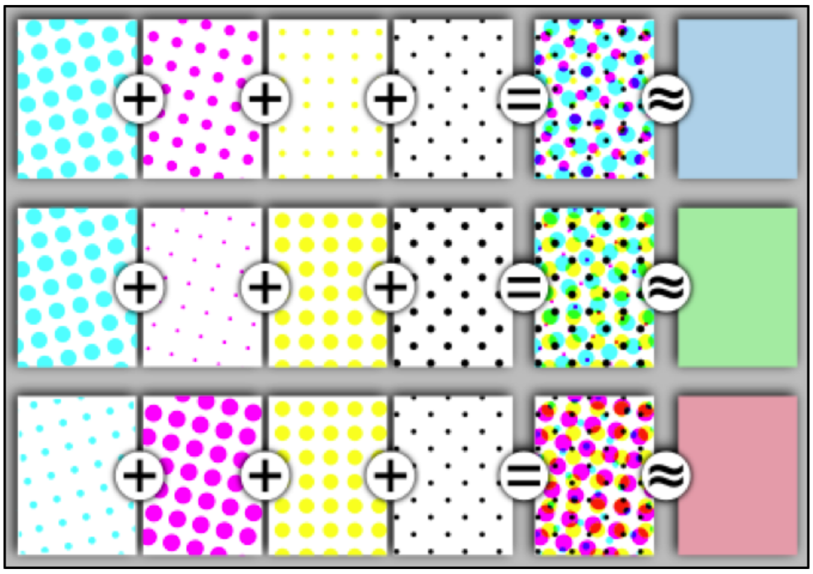
RGB Color Representation:

While printed colors are represented with the use of the four-color process, monitors represent color using the RGB color model which is an additive color model in which **R**ed, **G**reen and **B**lue light are added together in various ways to reproduce a broad array of colors.



The Difference Between Print and Monitor Color

Offset lithography is one of the most common ways of creating printed materials. A few of its applications include newspapers, magazines, brochures, stationery, and books. This model requires the image to be converted or made in the CMYK color model. All printed material relies on creating pigments of colors that when combined, forms the color as shown below.



The ink's semi-opacity property is in conjunction with the halftone technology and this is responsible for allowing pigments to mix and create new colors with just four primary ones.

On the other hand, media that transmit light (such as the monitor on your PC, tablet, or phone) use additive color mixing, which means that every pixel is made from three colors (RGB color model) by displaying different intensity the colors get produced.

Why Should I Care?

The first thing you might be wondering is why am I telling you how traditional press works, the existence of CMYK and RGB color models, or the pigment process (ink opacity plus halftoning) to print a brochure.

The rest of the tutorial will show you how to transform an image with different filters and techniques to deliver different outputs. These methods are still in use and part of a process known as Computer-To-Plate (CTP), used to create a direct output from an image file to a photographic film or plate (depending on the process), which are employed in industrial machines like the ones from Heidelberg.

This tutorial will give you insight into the filters and techniques used to transform images, some international image standards, and hopefully, some interest in the history of printing.

Quick Notes

1. Pillow is a fork of PIL (Python Imaging Library)
2. Pillow and PIL cannot co-exist in the same environment. Before installing Pillow, uninstall PIL.
3. `libjpeg-dev` is required to be able to process jpeg's with Pillow.
4. Pillow >= 2.0.0 supports Python versions 2.6, 2.7, 3.2, 3.3, and 3.4
5. Pillow >= 1.0 no longer supports `import Image`. Use `from PIL import Image` instead.

Installing Required Library

Before we start, we will need Python 3 and Pillow. If you are using Linux, Pillow will probably be there already, since major distributions including Fedora, Debian/Ubuntu, and ArchLinux include Pillow in packages that previously contained PIL.

The easiest way to install it is to use pip:

```
pip install Pillow
```

The installation process should look similar to the following.

If any trouble happens, I recommend reading the documentation from the Pillow Manual.

Basic Methods

Before manipulating an image, we need to be able to open the file, save the changes, create an empty picture, and to obtain individual pixels color. For the convenience of this tutorial, I have already made the methods to do so, which will be used in all subsequent sections.

These methods rely on the imported image library from the Pillow package.

```
# Imported PIL Library from PIL import Image

# Open an Image
def open_image(path):
  newImage = Image.open(path)
  return newImage

# Save Image
def save_image(image, path):
  image.save(path, 'png')

# Create a new image with the given size
def create_image(i, j):
  image = Image.new("RGB", (i, j), "white")
  return image

# Get the pixel from the given image
def get_pixel(image, i, j):
    # Inside image bounds?
    width, height = image.size
    if i > width or j > height:
      return None

    # Get Pixel
    pixel = image.getpixel((i, j))
    return pixel
```

Grayscale Filter

The traditional grayscale algorithm transforms an image to grayscale by obtaining the average channels color and making each channel equals to the average.

A better choice for grayscale is the ITU-R Recommendation BT.601-7, which specifies methods for digitally coding video signals by normalizing the values. For the grayscale transmissions, it defines the following formula.

```
# Create a Grayscale version of the image
def convert_grayscale(image):
  # Get size
  width, height = image.size

  # Create new Image and a Pixel Map
  new = create_image(width, height)
  pixels = new.load()

  # Transform to grayscale
  for i in range(width):
    for j in range(height):
      # Get Pixel
      pixel = get_pixel(image, i, j)

      # Get R, G, B values (This are int from 0 to 255)
      red =   pixel[0]
      green = pixel[1]
      blue =  pixel[2]

      # Transform to grayscale
      gray = (red * 0.299) + (green * 0.587) + (blue * 0.114)

      # Set Pixel in new image
      pixels[i, j] = (int(gray), int(gray), int(gray))

    # Return new image
    return new
```

By applying the filter with the above code, and using the BT.601-7 recommendation, we get the following result.

Half-tone Filter

The halftoning filter is the traditional method of printing images. It is a reprographic technique that simulates continuous tone imagery through the use of dots.

To generate the effect of shades of gray using only dots of black, we will define a size, in this case, a two by two matrix, and depending on the saturation we will draw black dots in this matrix.

```
# Create a Half-tone version of the image
def convert_halftoning(image):
  # Get size
  width, height = image.size

  # Create new Image and a Pixel Map
  new = create_image(width, height)
  pixels = new.load()

  # Transform to half tones
  for i in range(0, width, 2):
    for j in range(0, height, 2):
      # Get Pixels
      p1 = get_pixel(image, i, j)
      p2 = get_pixel(image, i, j + 1)
      p3 = get_pixel(image, i + 1, j)
      p4 = get_pixel(image, i + 1, j + 1)

      # Transform to grayscale
      gray1 = (p1[0] * 0.299) + (p1[1] * 0.587) + (p1[2] * 0.114)
      gray2 = (p2[0] * 0.299) + (p2[1] * 0.587) + (p2[2] * 0.114)
      gray3 = (p3[0] * 0.299) + (p3[1] * 0.587) + (p3[2] * 0.114)
      gray4 = (p4[0] * 0.299) + (p4[1] * 0.587) + (p4[2] * 0.114)

      # Saturation Percentage
      sat = (gray1 + gray2 + gray3 + gray4) / 4

      # Draw white/black depending on saturation
      if sat > 223:
        pixels[i, j]         = (255, 255, 255) # White
        pixels[i, j + 1]     = (255, 255, 255) # White
        pixels[i + 1, j]     = (255, 255, 255) # White
        pixels[i + 1, j + 1] = (255, 255, 255) # White
      elif sat > 159:
        pixels[i, j]         = (255, 255, 255) # White
        pixels[i, j + 1]     = (0, 0, 0)       # Black
        pixels[i + 1, j]     = (255, 255, 255) # White
        pixels[i + 1, j + 1] = (255, 255, 255) # White
      elif sat > 95:
        pixels[i, j]         = (255, 255, 255) # White
        pixels[i, j + 1]     = (0, 0, 0)       # Black
        pixels[i + 1, j]     = (0, 0, 0)       # Black
        pixels[i + 1, j + 1] = (255, 255, 255) # White
      elif sat > 32:
        pixels[i, j]         = (0, 0, 0)       # Black
        pixels[i, j + 1]     = (255, 255, 255) # White
        pixels[i + 1, j]     = (0, 0, 0)       # Black
        pixels[i + 1, j + 1] = (0, 0, 0)       # Black
      else:
        pixels[i, j]         = (0, 0, 0)       # Black
        pixels[i, j + 1]     = (0, 0, 0)       # Black
        pixels[i + 1, j]     = (0, 0, 0)       # Black
        pixels[i + 1, j + 1] = (0, 0, 0)       # Black

    # Return new image
    return new
```

By applying the filter using the code above, we are separating in five ranges and coloring the pixels on the array white or black depending on the saturation, which will produce the following result.

Dithering Filter

Dithering is an intentionally applied form of noise; it is used for processing an image to generate the illusion of colors by using the halftone filter on each color channel.

This method is used in traditional print as explained earlier.
In our approach, we will set the saturation for each channel,
in a direct fashion by replicating the halftone algorithm in each channel.
For a more advanced dither filter, you can read about the Floyd–Steinberg dithering.

```
# Return color value depending on quadrant and saturation
```

```
def get_saturation(value, quadrant):
  if value > 223:
    return 255
  elif value > 159:
    if quadrant != 1:
      return 255

    return 0
  elif value > 95:
    if quadrant == 0 or quadrant == 3:
      return 255

    return 0
  elif value > 32:
    if quadrant == 1:
      return 255

    return 0
  else:
    return 0

# Create a dithered version of the image
def convert_dithering(image):
  # Get size
  width, height = image.size

  # Create new Image and a Pixel Map
  new = create_image(width, height)
  pixels = new.load()

  # Transform to half tones
  for i in range(0, width, 2):
    for j in range(0, height, 2):
      # Get Pixels
      p1 = get_pixel(image, i, j)
      p2 = get_pixel(image, i, j + 1)
      p3 = get_pixel(image, i + 1, j)
      p4 = get_pixel(image, i + 1, j + 1)

      # Color Saturation by RGB channel
      red   = (p1[0] + p2[0] + p3[0] + p4[0]) / 4
      green = (p1[1] + p2[1] + p3[1] + p4[1]) / 4
      blue  = (p1[2] + p2[2] + p3[2] + p4[2]) / 4

      # Results by channel
      r = [0, 0, 0, 0]
      g = [0, 0, 0, 0]
      b = [0, 0, 0, 0]

      # Get Quadrant Color
      for x in range(0, 4):
        r[x] = get_saturation(red, x)
        g[x] = get_saturation(green, x)
        b[x] = get_saturation(blue, x)

      # Set Dithered Colors
      pixels[i, j]         = (r[0], g[0], b[0])
      pixels[i, j + 1]     = (r[1], g[1], b[1])
      pixels[i + 1, j]     = (r[2], g[2], b[2])
      pixels[i + 1, j + 1] = (r[3], g[3], b[3])

    # Return new image
    return new
```

By processing each channel, we set the colors to the primary and secondary ones. This is a total of eight colors, but as you notice in the processed image, they give the illusion of a wider array of colors.

Complete Source

The complete code to process images takes a PNG file in RGB color mode (with no transparency), saving the output as different images.

Due to limitations with JPEG support on various operating systems, I choose the PNG format.

```
 '''This Example opens an Image and transform the image into grayscale, halftc
You need PILLOW (Python Imaging Library fork) and Python 3.5
    -Isai B. Cicourel'''

# Imported PIL Library from PIL import Image

# Open an Image
def open_image(path):
  newImage = Image.open(path)
  return newImage
```

```python
# Save Image
def save_image(image, path):
  image.save(path, 'png')


# Create a new image with the given size
def create_image(i, j):
  image = Image.new("RGB", (i, j), "white")
  return image


# Get the pixel from the given image
def get_pixel(image, i, j):
  # Inside image bounds?
  width, height = image.size
  if i > width or j > height:
    return None

  # Get Pixel
  pixel = image.getpixel((i, j))
  return pixel

# Create a Grayscale version of the image
def convert_grayscale(image):
  # Get size
  width, height = image.size

  # Create new Image and a Pixel Map
  new = create_image(width, height)
  pixels = new.load()

  # Transform to grayscale
  for i in range(width):
    for j in range(height):
      # Get Pixel
      pixel = get_pixel(image, i, j)

      # Get R, G, B values (This are int from 0 to 255)
      red =   pixel[0]
      green = pixel[1]
      blue =  pixel[2]

      # Transform to grayscale
      gray = (red * 0.299) + (green * 0.587) + (blue * 0.114)

      # Set Pixel in new image
      pixels[i, j] = (int(gray), int(gray), int(gray))

  # Return new image
  return new


# Create a Half-tone version of the image
def convert_halftoning(image):
  # Get size
  width, height = image.size

  # Create new Image and a Pixel Map
  new = create_image(width, height)
  pixels = new.load()

  # Transform to half tones
  for i in range(0, width, 2):
    for j in range(0, height, 2):
      # Get Pixels
      p1 = get_pixel(image, i, j)
      p2 = get_pixel(image, i, j + 1)
      p3 = get_pixel(image, i + 1, j)
      p4 = get_pixel(image, i + 1, j + 1)

       # Transform to grayscale
       gray1 = (p1[0] * 0.299) + (p1[1] * 0.587) + (p1[2] * 0.114)
       gray2 = (p2[0] * 0.299) + (p2[1] * 0.587) + (p2[2] * 0.114)
       gray3 = (p3[0] * 0.299) + (p3[1] * 0.587) + (p3[2] * 0.114)
       gray4 = (p4[0] * 0.299) + (p4[1] * 0.587) + (p4[2] * 0.114)

       # Saturation Percentage
       sat = (gray1 + gray2 + gray3 + gray4) / 4

       # Draw white/black depending on saturation
       if sat > 223:
         pixels[i, j]         = (255, 255, 255) # White
         pixels[i, j + 1]     = (255, 255, 255) # White
         pixels[i + 1, j]     = (255, 255, 255) # White
         pixels[i + 1, j + 1] = (255, 255, 255) # White
       elif sat > 159:
         pixels[i, j]         = (255, 255, 255) # White
         pixels[i, j + 1]     = (0, 0, 0)       # Black
         pixels[i + 1, j]     = (255, 255, 255) # White
         pixels[i + 1, j + 1] = (255, 255, 255) # White
       elif sat > 95:
         pixels[i, j]         = (255, 255, 255) # White
         pixels[i, j + 1]     = (0, 0, 0)       # Black
         pixels[i + 1, j]     = (0, 0, 0)       # Black
         pixels[i + 1, j + 1] = (255, 255, 255) # White
       elif sat > 32:
         pixels[i, j]         = (0, 0, 0)       # Black
         pixels[i, j + 1]     = (255, 255, 255) # White
         pixels[i + 1, j]     = (0, 0, 0)       # Black
         pixels[i + 1, j + 1] = (0, 0, 0)       # Black
       else:
         pixels[i, j]         = (0, 0, 0)       # Black
         pixels[i, j + 1]     = (0, 0, 0)       # Black
         pixels[i + 1, j]     = (0, 0, 0)       # Black
         pixels[i + 1, j + 1] = (0, 0, 0)       # Black

  # Return new image
  return new


# Return color value depending on quadrant and saturation
def get_saturation(value, quadrant):
  if value > 223:
    return 255
  elif value > 159:
    if quadrant != 1:
      return 255

    return 0
  elif value > 95:
    if quadrant == 0 or quadrant == 3:
      return 255

    return 0
  elif value > 32:
    if quadrant == 1:
      return 255

    return 0
  else:
    return 0


# Create a dithered version of the image
def convert_dithering(image):
  # Get size
  width, height = image.size

  # Create new Image and a Pixel Map
  new = create_image(width, height)
  pixels = new.load()
  pixels = new.load()

  # Transform to half tones
  for i in range(0, width, 2):
    for j in range(0, height, 2):
      # Get Pixels
      p1 = get_pixel(image, i, j)
      p2 = get_pixel(image, i, j + 1)
      p3 = get_pixel(image, i + 1, j)
      p4 = get_pixel(image, i + 1, j + 1)

      # Color Saturation by RGB channel
      red   = (p1[0] + p2[0] + p3[0] + p4[0]) / 4
      green = (p1[1] + p2[1] + p3[1] + p4[1]) / 4
      blue  = (p1[2] + p2[2] + p3[2] + p4[2]) / 4

      # Results by channel
      r = [0, 0, 0, 0]
      g = [0, 0, 0, 0]
      b = [0, 0, 0, 0]

      # Get Quadrant Color
      for x in range(0, 4):
        r[x] = get_saturation(red, x)
        g[x] = get_saturation(green, x)
        b[x] = get_saturation(blue, x)

      # Set Dithered Colors
      pixels[i, j]         = (r[0], g[0], b[0])
      pixels[i, j + 1]     = (r[1], g[1], b[1])
      pixels[i + 1, j]     = (r[2], g[2], b[2])
      pixels[i + 1, j + 1] = (r[3], g[3], b[3])

  # Return new image
  return new


# Create a Primary Colors version of the image
def convert_primary(image):
  # Get size
  width, height = image.size

  # Create new Image and a Pixel Map
  new = create_image(width, height)
  pixels = new.load()

  # Transform to primary
  for i in range(width):
    for j in range(height):
      # Get Pixel
      pixel = get_pixel(image, i, j)

      # Get R, G, B values (This are int from 0 to 255)
      red =   pixel[0]
      green = pixel[1]
      blue =  pixel[2]

      # Transform to primary
      if red > 127:
        red = 255
      else:
        red = 0
      if green > 127:
        green = 255
      else:
        green = 0
      if blue > 127:
        blue = 255
      else:
        blue = 0

      # Set Pixel in new image
      pixels[i, j] = (int(red), int(green), int(blue))

  # Return new image
  return new


# Main
if __name__ == "__main__":
  # Load Image (JPEG/JPG needs libjpeg to load)
  original = open_image('Hero_Prinny.png')

  # Example Pixel Color
  print('Color: ' + str(get_pixel(original, 0, 0)))

  # Convert to Grayscale and save
  new = convert_grayscale(original)
  save_image(new, 'Prinny_gray.png')

  # Convert to Halftoning and save
  new = convert_halftoning(original)
  save_image(new, 'Prinny_half.png')

  # Convert to Dithering and save
```

```
new = convert_dithering(original)
save_image(new, 'Prinny_dither.png')

# Convert to Primary and save
new = convert_primary(original)
save_image(new, 'Prinny_primary.png')
```

The source code takes an image, then applies each filter and saves the output as a new image, producing the following results.

Don't forget to specify the path to the image in `original = open_image('Hero_Prinny.png')` and on the outputs. Unless you have that image, which would mean you are a Disgaea fan.

Wrapping Up

Image filters are not only something we use to make our pictures on social networking sites look cool, they are useful and powerful techniques for processing videos and images not only for printing in an offset; but also to compress and improve playback and speed of on-demand services.

Have You Tried The Following?

Now that you know some image filters, how about applying several of them on the same picture?
On a large image, what happens with a filter when you "fit to screen"?
Did you notice the extra filter in the complete source code?
Have you checked the size of the different output images?
Play around and check what happens. Create your filter or implement a new one, the idea is to learn new things. If you like this tutorial, share it with your friends. 😉

Other tutorials you might be interested in:

- Interview with Ray Phan: MATLAB & Image Processing Expert
- An Introduction to Materials & Standard Shader in Unity

Image processing | Image filtering | Image manipulation | Python

Report

Enjoy this post? Give **Isai B. Cicourel** a like if it's helpful.

21 8 SHARE

Isai B. Cicourel

Software Developer Engineer at Amazon Go

My name is Isai B. Cicourel; I am an Engineer at Amazon. Formerly a senior Oracle developer for the Spatial and Graph team. My experience involves the Web Ontology Language, Protege Integration, Jena, and Graph Databases. Since I ...

FOLLOW

8 Replies

Leave a reply

SRI FRENZILINO MAHAYYU AKBARISENA a year ago

Thanks!!

Reply

Wael Salman 2 years ago

Thank you. Appreciated.

One of the critical issues that I wonder about is: Once we scan an image, and we load it using pillow, pill or opencv, and then getting all pixels values and colors, we notice that the colors values are too different from the original

Show more

Reply

Isai B. Cicourel 2 years ago

As a recommendation, it is essential first to understand the concept of Color Proof.

Why?

Well, the answer you seek depends on your need to close the gap

Show more

Reply

Wael Salman 2 years ago

Thank you so much. :-) So appreciated

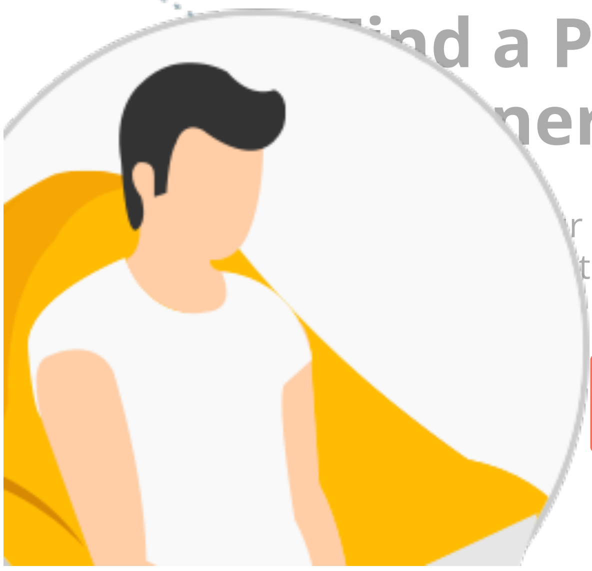
Reply

TresDeHope W 3 years ago

Very clean code example. Thank you.

Reply

Show more replies



ind a Pair Progra
ner on Coden

r programming skills? Ch
tors to pair program with

GET STARTED

Corentin

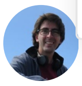
How to Listen for Webhooks with Python

Webhooks run a large portion of the "magic" that happens between applications. They are sometimes called reverse APIs, callbacks, and even notifications. Many services, such as SendGrid, Stripe, Slack, and GitHub use events to send webhooks as part of their API. This allows your application to listen for events and perform actions when they happen.

READ MORE

Join and start **discussions**
with fellow developers

START A DISCUSSION



Discover and read more posts from **Isai B. Cicourel** GET STARTED