Image Processing in Python

Image Processing in Python

Black Raven (James Ng)
13 Mar 2021 · 51 min read

This is a memo to share what I have learnt in Image Processing (in Python), capturing the learning objectives as well as my personal notes. The course is taught by Rebeca Gonzalez from DataCamp, and it includes 4 chapters:

Chapter 1. Introducing Image Processing and scikit-image
Chapter 2. Filters, Contrast, Transformation and Morphology
Chapter 3. Image restoration, Noise, Segmentation and Contours
Chapter 4. Advanced Operations, Detecting Faces and Features

Images are everywhere! We live in a time where images contain lots of information, which is sometimes difficult to obtain. This is why image pre-processing has become a highly valuable skill, applicable in many use cases. In this course, you will learn to process, transform, and manipulate images at your will, even when they come in thousands. You will also learn to restore damaged images, perform noise reduction, smart-resize images, count the number of dots on a dice, apply facial detection, and much more, using **scikit-image**.

After completing this course, you will be able to apply your knowledge to different domains such as machine learning and artificial intelligence, machine and robotic vision, space and medical image analysis, retailing, and many more. Take the step and dive into the wonderful world that is computer vision!

Chapter 1. Introducing Image Processing and scikit-image

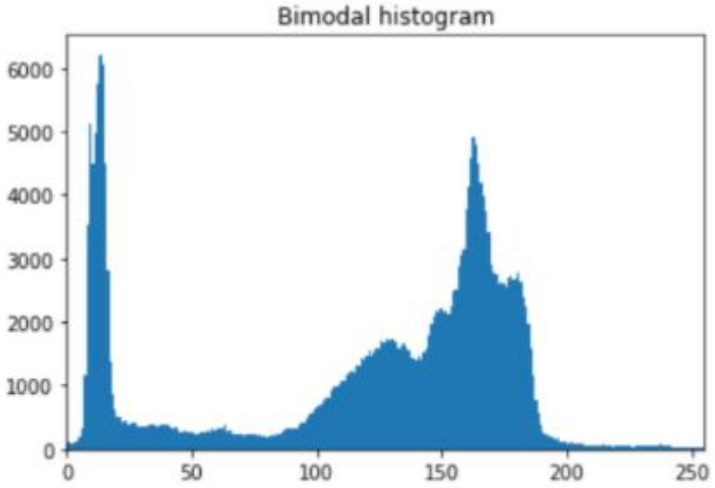
Jump into digital image structures and learn to process them! Extract data, transform and analyze images using NumPy and Scikit-image. With just a few lines of code, you will convert RGB images to grayscale, get data from them, obtain histograms containing very useful information, and separate objects from the background!

Make images come alive with scikit-image

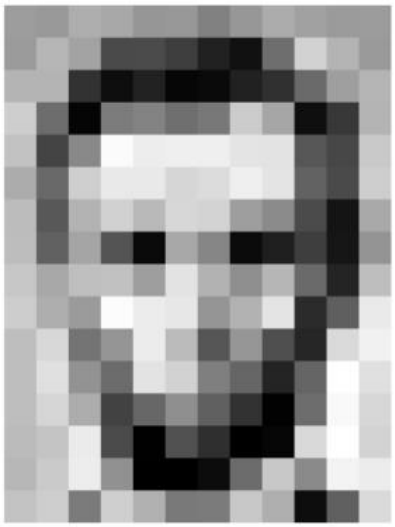
Purpose of image processing

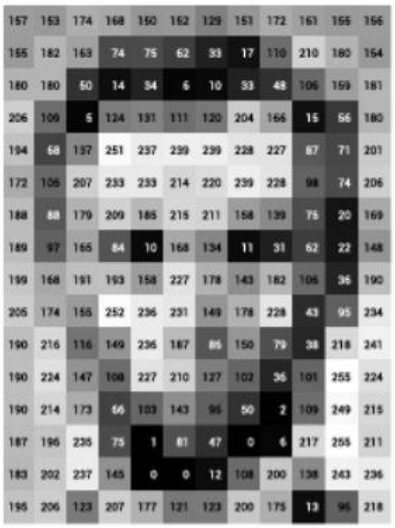
1. Visualisation – to observe objects that are not visible
2. Image sharpening and restoration – for a better image
3. Image retrieval – seek for the image of interest
4. Measurement of pattern – measure various objects
5. Image recognition – distinguish objects in an image

What is an image?
A digital image is an array, or a matrix, of square pixels (picture elements) arranged in columns and rows: in other words, a 2-dimensional matrix. These pixels contain information about color and intensity.



Here's an example of the matrix for a 2D grayscale image. Here we can see that the first image is a pixelated image. The numbers that we see on top of the second image correspond to the intensity of each pixel in the image. So at the end, an image can be treated as an intensities matrix.





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	159	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	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
205	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	235	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	123	200	175	13	96	218

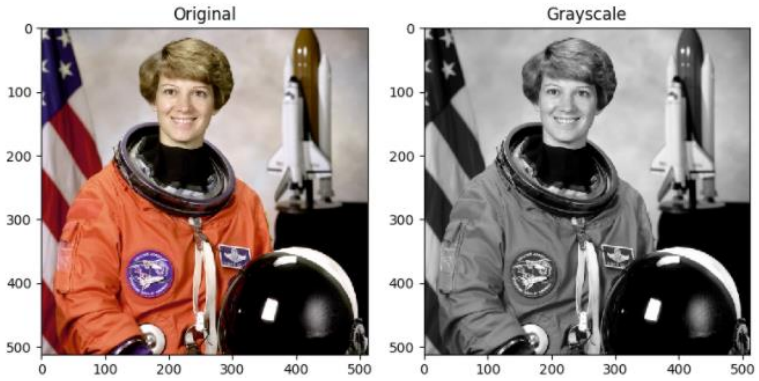

```
from skimage import data
rocket_image = data.rocket()
```

2-dimensional color images are often represented in RGB—3 layers of 2-dimensional arrays, where the three layers represent Red, Green and Blue channels of the image.

Grayscale images only have shades of black and white. Often, the grayscale intensity is stored as an 8-bit integer giving 256 possible different shades of gray. Grayscale images don't have any color information.

230	229	232	234	235	232	148
237	236	236	234	233	234	152
255	255	255	251	230	236	161
99	90	67	37	94	247	130
222	152	255	129	129	246	132
154	199	255	150	189	241	147
216	132	162	163	170	239	122

```
from skimage import color
grayscale = color.rgb2gray(original)
rgb = color.gray2rgb(grayscale)
```



RGB images have three color channels, while grayscaled ones have a single channel. We can convert an image with RGB channels into grayscale using the function rgb2gray() provided in the color module. We can also turn grayscale to RGB using gray2rgb().

```
def show_image(image, title='Image', cmap_type='gray'):
    plt.imshow(image, cmap=cmap_type)
    plt.title(title)
    plt.axis('off')
    plt.show()
```

```
from skimage import color
grayscale = color.rgb2gray(original)

show_image(grayscale, "Grayscale")
```

Is this gray or full of color?

Whats the main difference between the images shown below?

These images have been preloaded as `coffee_image` and `coins_image` from the scikit-image `data` module using:

```
coffee_image = data.coffee()
coins_image = data.coins()
```

Choose the right answer that best describes the main difference related to color and dimensional structure.

In the console, use the function `shape()` from NumPy, to obtain the image shape (Height, Width, Dimensions) and find out. NumPy is already imported as `np`.

- ○ Both have 3 channels for RGB-3 color representation.
- ○ `coffee_image` has a shape of (303, 384), grayscale. And `coins_image` (400, 600, 3), RGB-3.
- ◉ `coins_image` has a shape of (303, 384), grayscale. And `coffee_image` (400, 600, 3), RGB-3.
- ○ Both are grayscale, with single color dimension.

```
In [1]: coffee_image.shape
Out[1]: (400, 600, 3)

In [2]: coins_image.shape
Out[2]: (303, 384)
```

The coffee image is RGB-3 colored, that's why it has a 3 at the end, when displaying the shape (H, W, D) of it. While the coins image is grayscale and has a single color channel.

RGB to grayscale

In this exercise you will load an image from scikit-image module `data` and make it grayscale, then compare both of them in the output.

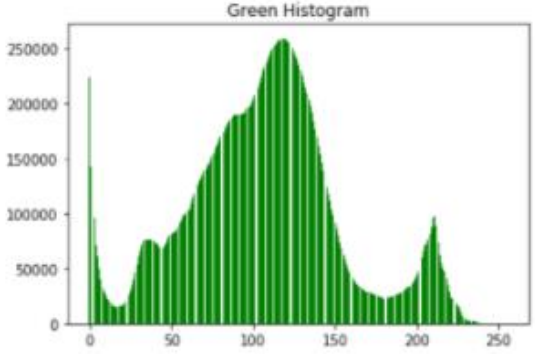
We have preloaded a function `show_image(image, title='Image')` that displays the image using Matplotlib. You can check more about its parameters using `?show_image()` or `help(show_image)` in the console.


```
# Import the modules from skimage
from skimage import data, color

# Load the rocket image
```

```
rocket = data.rocket()

# Convert the image to grayscale
gray_scaled_rocket = color.rgb2gray(rocket)

# Show the original image
show_image(rocket, 'Original RGB image')

# Show the grayscale image
show_image(gray_scaled_rocket, 'Grayscale image')
```

You converted an image to grayscale. For many applications of image processing, color information doesn't help us identify important edges or other features. Something that we will cover later in the course.

NumPy for images



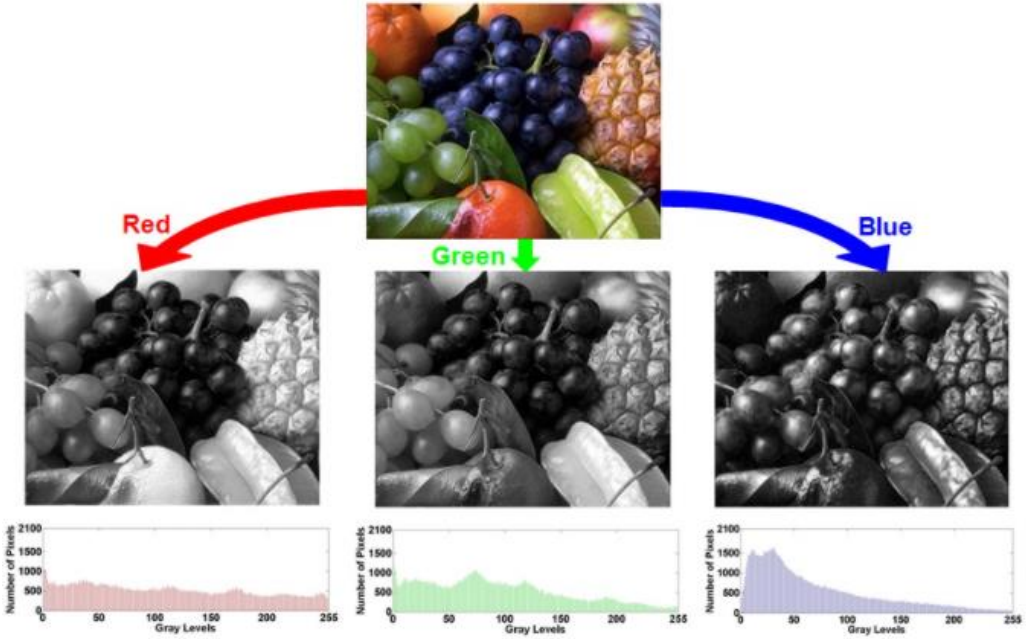
Flipping image, Extract/Analyse features

```
# Loading the image using Matplotlib
madrid_image = plt.imread('/madrid.jpeg')

type(madrid_image)
```

```
<class 'numpy.ndarray'>
```

ndarray = Numpy multi-dimensional arrays

```
# Obtaining the red values of the image
red = image[:, :, 0]

# Obtaining the green values of the image
green = image[:, :, 1]

# Obtaining the blue values of the image
blue = image[:, :, 2]
```



Here we can see the individual color intensities along the image. For example, we obtain the red color of an image by keeping the height and width pixels and selecting only the values of the first color layer.


```
plt.imshow(red, cmap="gray")
plt.title('Red')
plt.axis('off')
plt.show()
```

```
# Accessing the shape of the image
madrid_image.shape
```

```
(426, 640, 3)
```

This Madrid picture is 426 pixels high and 640 pixels wide. It has three layers for color representation: it's an RGB-3 image.

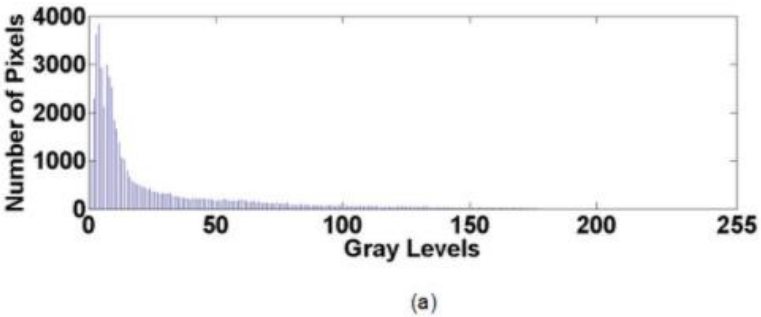
```
# Accessing the shape of the image
madrid_image.size
```

```
817920
```

So it has shape of (426, 640, 3). And a total number of pixels of 817920.

```
# Flip the image in up direction
vertically_flipped = np.flipud(madrid_image)

show_image(vertically_flipped, 'Vertically flipped image')
```

```
# Flip the image in left direction
horizontally_flipped = np.fliplr(madrid_image)

show_image(horizontally_flipped, 'Horizontally flipped image')
```

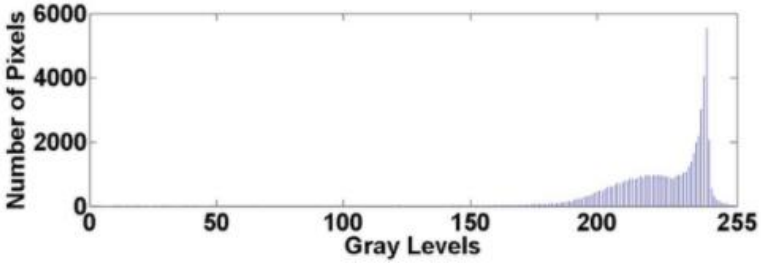


(a)

(b)

The histogram of an image is a graphical representation of the amount of pixels of each intensity value.

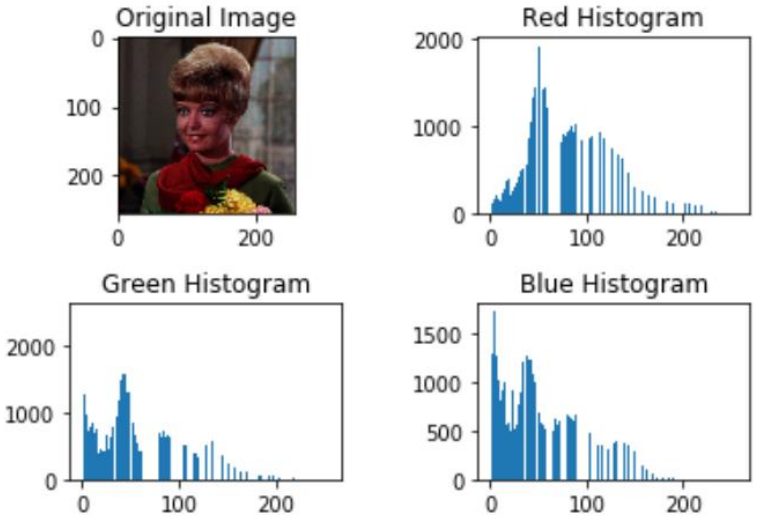
From 0 (pure black) to 255(pure white).



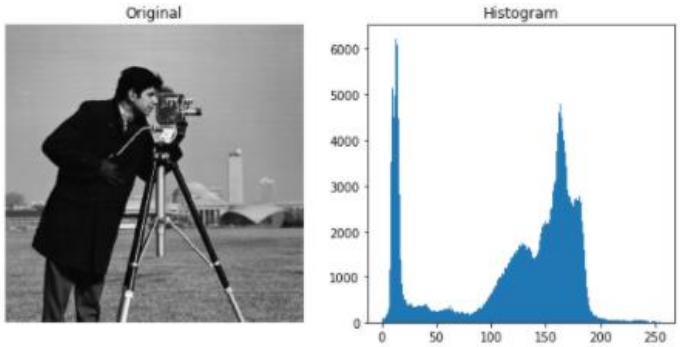
(a)

(b)

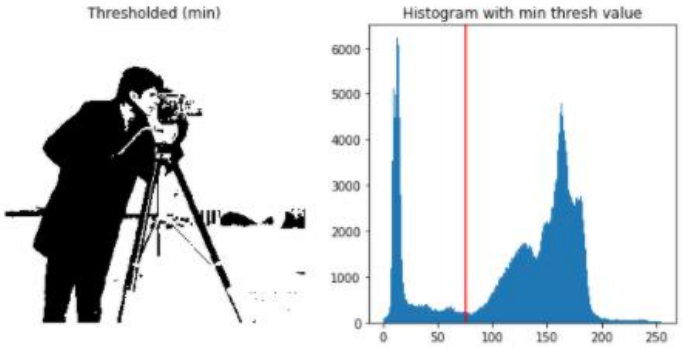
The first image is really dark, so most of the pixels have a low intensity, from 0 to 50. While the second one, it's lighter and has most of the pixels close to 200 and 255.

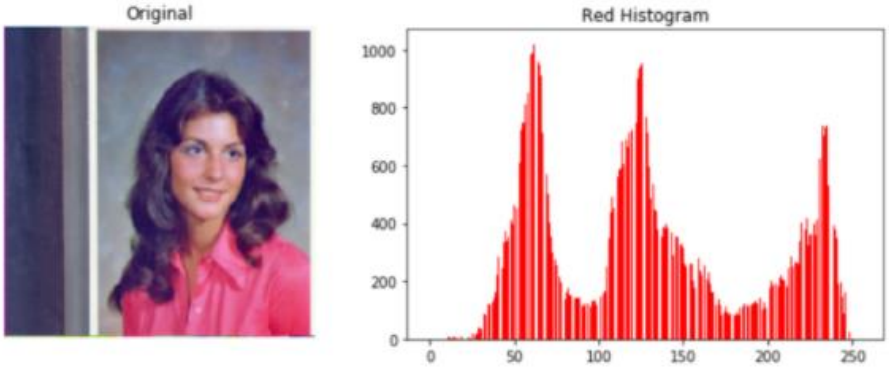


We can also create histograms from RGB-3 colored images. In this case each channel: red, green and blue will have a corresponding histogram.



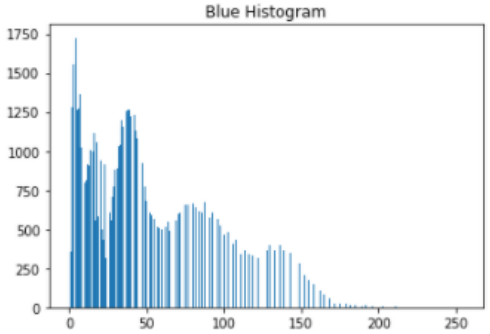
Histograms are used for analysis, and to threshold images (an important topic in computer vision that we will cover later in this course), to alter brightness and contrast, and to equalize an image (which we will also cover later in this course).






```
# Red color of the image
red = image[:, :, 0]

# Obtain the red histogram
plt.hist(red.ravel(), bins=256)
```

Matplotlib has a histogram method. It takes an input array (frequency) and bins as parameters. The successive elements in bin array act as the boundary of each bin. We obtain the red color channel of the image by slicing it. We then use the histogram function. Use ravel to return a continuous flattened array from the color values of the image, in this case red. And pass this ravel and the bins as parameters.




```
blue = image[:, :, 2]

plt.hist(blue.ravel(), bins=256)
plt.title('Blue Histogram')
plt.show()
```

We set bins to 256 because we'll show the number of pixels for every pixel value, that is, from 0 to 255. Meaning you need 256 values to show the histogram.

Flipping out

As a prank, someone has turned an image from a photo album of a trip to Seville upside-down and back-to-front! Now, we need to straighten the image, by flipping it.

Image loaded as `flipped_seville`.

Using the NumPy methods learned in the course, flip the image horizontally and vertically. Then display the corrected image using the `show_image()` function.

NumPy is already imported as `np`.

```
# Flip the image vertically
seville_vertical_flip = np.flipud(flipped_seville)

# Flip the image horizontally
seville_horizontal_flip = np.fliplr(seville_vertical_flip)

# Show the resulting image
show_image(seville_horizontal_flip, 'Seville')
```

Histograms

In this exercise, you will analyze the amount of red in the image. To do this, the histogram of the red channel will be computed for the image shown below:

Image loaded as `image`.

Extracting information from images is a fundamental part of image enhancement. This way you can balance the red and blue to make the image look colder or warmer.

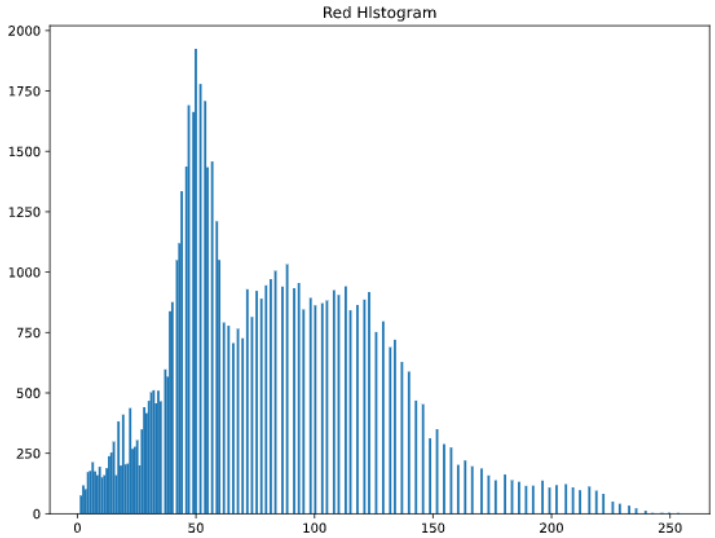
You will use `hist()` to display the 256 different intensities of the red color. And `ravel()` to make these color values an array of one flat dimension.

Matplotlib is preloaded as `plt` and Numpy as `np`.

Remember that if we want to obtain the green color of an image we would do the following:

```
green = image[:, :, 1]
```

```
# Obtain the red channel
red_channel = image[:, :, 0]

# Plot the red histogram with bins in a range of 256
plt.hist(red_channel.ravel(), bins=256)

# Set title and show
plt.title('Red Histogram')
plt.show()
```

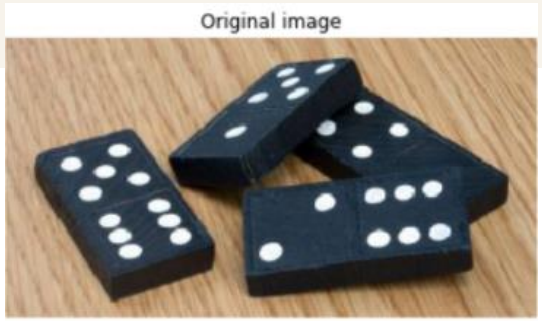


With this histogram we see that the image is quite reddish, meaning it has a sensation of warmness. This is because it has a wide and large distribution of bright red pixels, from 0 to around 150.

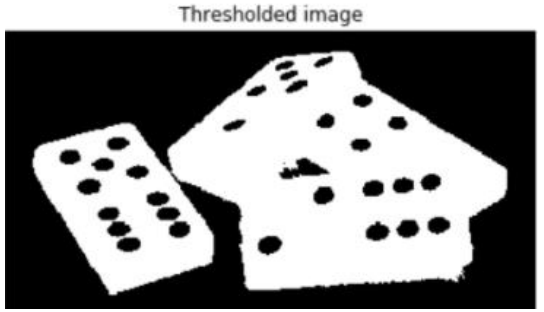
Getting started with thresholding

Thresholding is used to partition the background and foreground of grayscale images, by essentially making them black and white. We compare each pixel to a given threshold value.

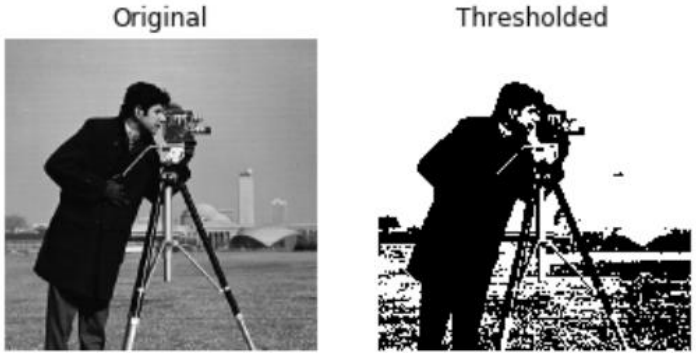
- If pixel > thresh value: 255 (white)
- If pixel < thresh value: 0 (black)



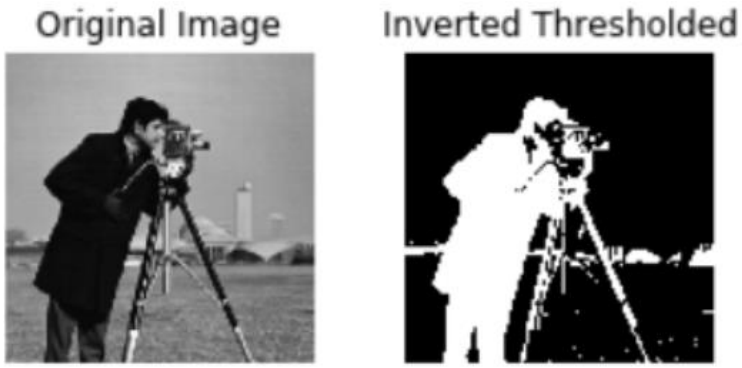
Thresholding let us isolate elements and is used in object detection, facial recognition, and other applications. It works best in high-contrast grayscale images. To threshold color images, we must first convert them to grayscale.




```
# Obtain the optimal threshold value
thresh = 127

# Apply thresholding to the image
binary = image > thresh

# Show the original and thresholded
show_image(image, 'Original')
show_image(binary, 'Thresholded')
```

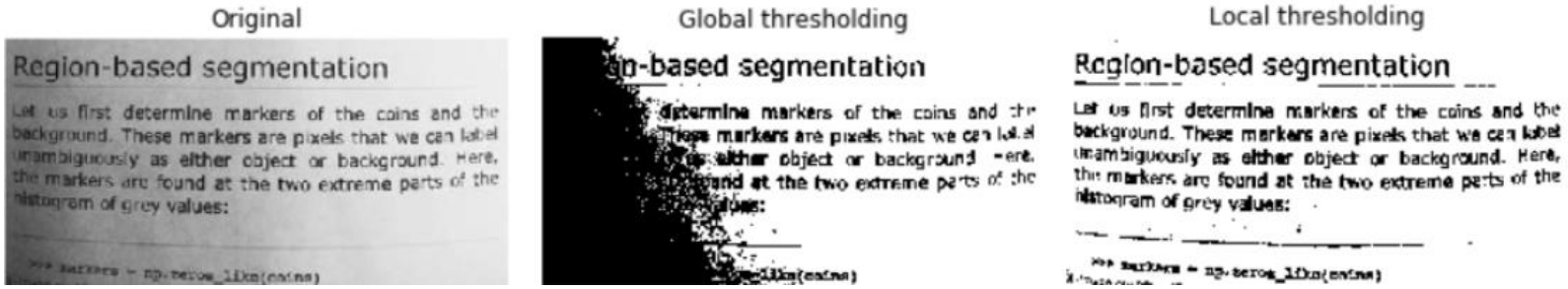



```
# Obtain the optimal threshold value
thresh = 127

# Apply thresholding to the image
inverted_binary = image <= thresh

# Show the original and thresholded
show_image(image, 'Original')
show_image(inverted_binary,
           'Inverted thresholded')
```



There are two categories of thresholding in scikit-image:

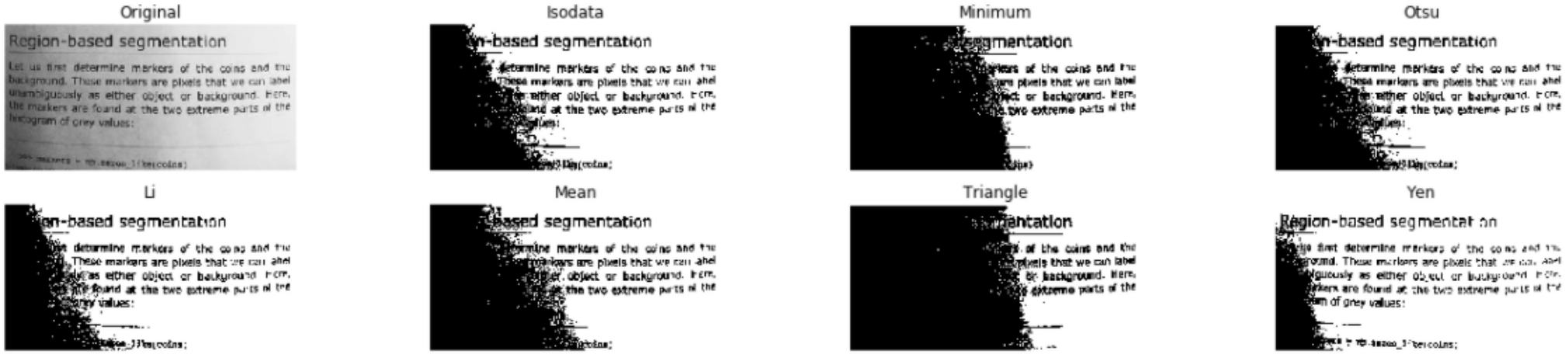
- Global or histogram-based, which is good for images that have relatively uniform backgrounds.
- Adaptive or local, which is best for images where the background is not easily differentiated, with uneven background illumination. Note that local is slower than global thresholding.



Here we have an image to compare. In this case, it seems that local is without a doubt the best option.

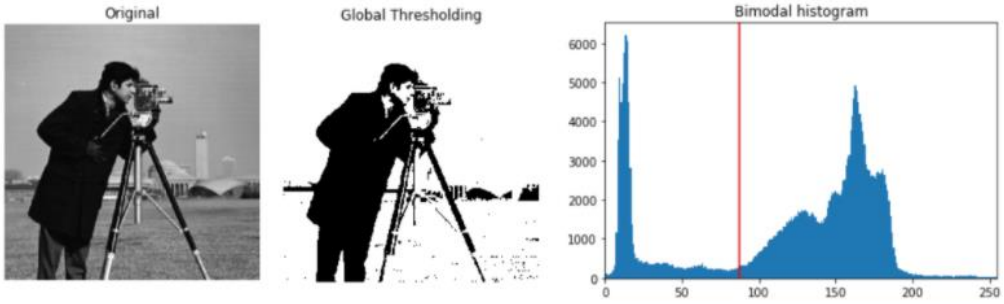
```
from skimage.filters import try_all_threshold

# Obtain all the resulting images
fig, ax = try_all_threshold(image, verbose=False)

# Showing resulting plo
```

(incomplete or missing script/code)



When the background of an image seems uniform, global thresholding works best.

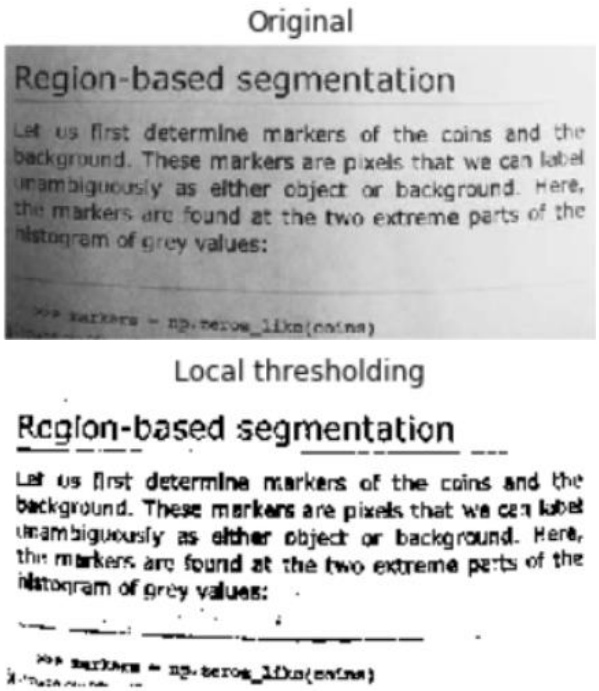
```
# Import the otsu threshold function
from skimage.filters import threshold_otsu

# Obtain the optimal threshold value
thresh = threshold_otsu(image)

# Apply thresholding to the image
binary_global = image > thresh

# Show the original and binarized image
show_image(image, 'Original')
show_image(binary_global, 'Global thresholding')
```



If the image doesn't have high contrast or the background is uneven, local thresholding produces better results.

```
# Import the local threshold function
from skimage.filters import threshold_local

# Set the block size to 35
block_size = 35

# Obtain the optimal local thresholding
local_thresh = threshold_local(text_image, block_size, offset=10)

# Apply local thresholding and obtain the binary image
binary_local = text_image > local_thresh
```

```
# Show the original and binarized image
show_image(image, 'Original')
show_image(binary_local, 'Local thresholding')
```



Apply global thresholding

In this exercise, you'll transform a photograph to binary so you can separate the foreground from the background.

To do so, you need to import the required modules, load the image, obtain the optimal thresh value using `threshold_otsu()` and apply it to the image.

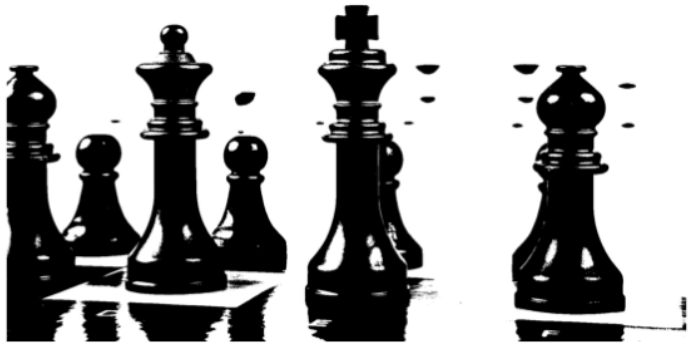
You'll see the resulting binarized image when using the `show_image()` function, previously explained.

Image loaded as `chess_pieces_image`.

Remember we have to turn colored images to grayscale. For that we will use the `rgb2gray()` function learned in previous video. Which has already been imported for you.

```
# Import the otsu threshold function
from skimage.filters import threshold_otsu

# Make the image grayscale using rgb2gray
chess_pieces_image_gray = rgb2gray(chess_pieces_image)

# Obtain the optimal threshold value with otsu
thresh = threshold_otsu(chess_pieces_image_gray)

# Apply thresholding to the image
binary = chess_pieces_image_gray > thresh
```

```
# Show the image
show_image(binary, 'Binary image')
```



You just converted the image to binary and we can separate the foreground from the background.

When the background isn't that obvious

Sometimes, it isn't that obvious to identify the background. If the image background is relatively uniform, then you can use a global threshold value as we practiced before, using `threshold_otsu()`. However, if there's uneven background illumination, adaptive thresholding `threshold_local()` (a.k.a. local thresholding) may produce better results.

You will compare both types of thresholding methods (global and local), to find the optimal way to obtain the binary image we need.

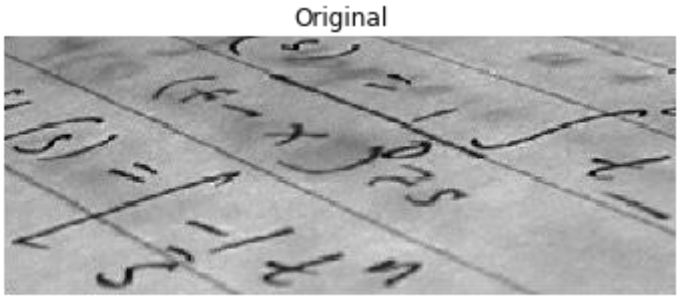
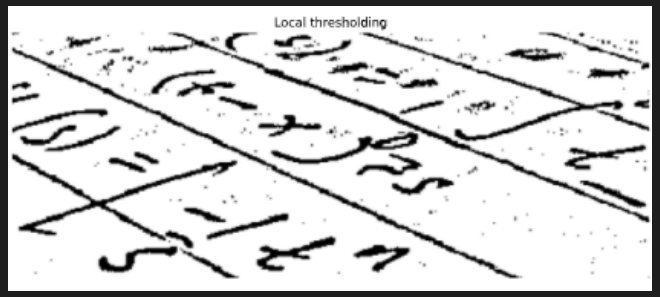


Image loaded as `page_image`.

```
# Import the otsu threshold function
from skimage.filters import threshold_otsu

# Obtain the optimal otsu global thresh value
global_thresh = threshold_otsu(page_image)

# Obtain the binary image by applying global thresholding
binary_global = page_image > global_thresh

# Show the binary image obtained
show_image(binary_global, 'Global thresholding')
```


```
# Import the local threshold function
from skimage.filters import threshold_local

# Set the block size to 35
block_size = 35

# Obtain the optimal local thresholding
local_thresh = threshold_local(page_image, block_size, offset=10)

# Obtain the binary image by applying local thresholding
binary_local = page_image > local_thresh

# Show the binary image
show_image(binary_local, 'Local thresholding')
```



Now you know that you should use local thresholding instead of global if the image has a wide variation of background intensity.

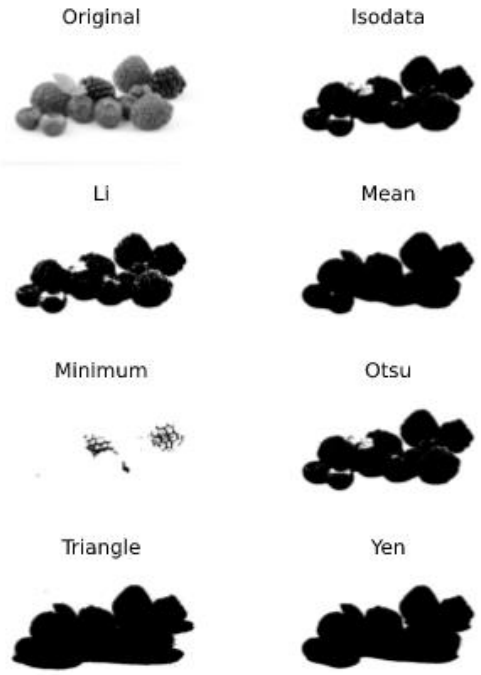
Trying other methods

As we saw in the video, not being sure about what thresholding method to use isn't a problem. In fact, scikit-image provides us with a function to check multiple methods and see for ourselves what the best option is. It returns a figure comparing the outputs of different **global** thresholding methods.

Image loaded as `fruits_image`.

You will apply this function to this image, `matplotlib.pyplot` has been loaded as `plt`. Remember that you can use `try_all_threshold()` to try multiple global algorithms.

```
# Import the try all function
from skimage.filters import try_all_threshold

# Import the rgb to gray convertor function
from skimage.color import rgb2gray

# Turn the fruits_image to grayscale
grayscale = rgb2gray(fruits_image)

# Use the try all method on the resulting grayscale image
fig, ax = try_all_threshold(grayscale, verbose=False)

# Show the resulting plots
plt.show()
```



As you see, this image works good with some global thresholding methods (like the "Yen" and "Mean") and not so well in others, (like the "Minimum").

Apply thresholding

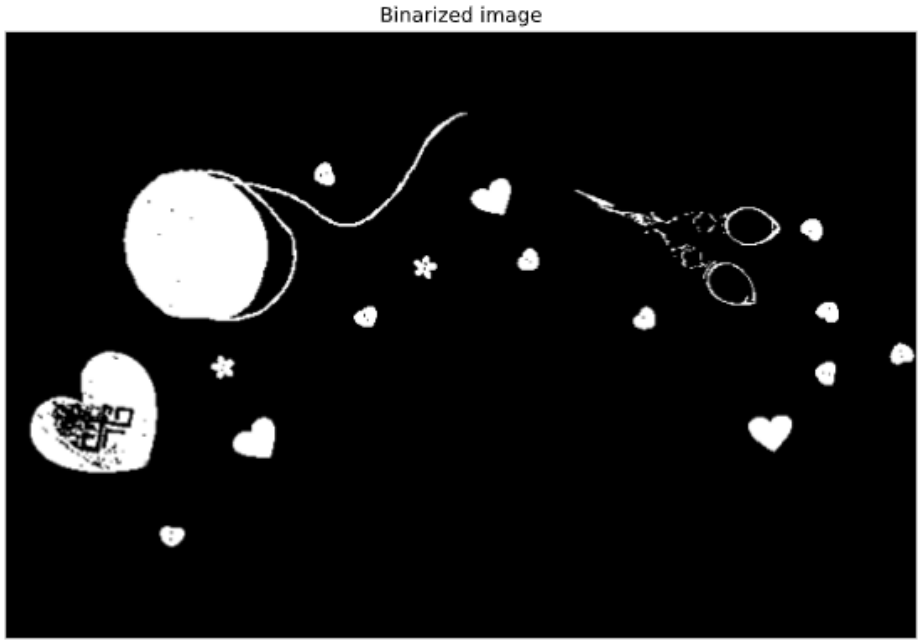
In this exercise, you will decide what type of thresholding is best used to binarize an image of knitting and craft tools. In doing so, you will be able to see the shapes of the objects, from paper hearts to scissors more clearly.

Image loaded as `tools_image`.

What type of thresholding would you use judging by the characteristics of the image? Is the background illumination and intensity even or uneven?

```
# Import threshold and gray convertor functions
from skimage.filters import threshold_otsu
from skimage.color import rgb2gray

# Turn the image grayscale
gray_tools_image = rgb2gray(tools_image)

# Obtain the optimal thresh
thresh = threshold_otsu(gray_tools_image)

# Obtain the binary image by applying thresholding
```

```
binary_image = gray_tools_image > thresh

# Show the resulting binary image
show_image(binary_image, 'Binarized image')
```



By using a global thresholding method, you obtained the precise binarized image. If you would have used local instead nothing would have been segmented.

Try it yourself and see it! In the next chapters, we'll get into image restoration, face detection, and much more. Stay tuned!

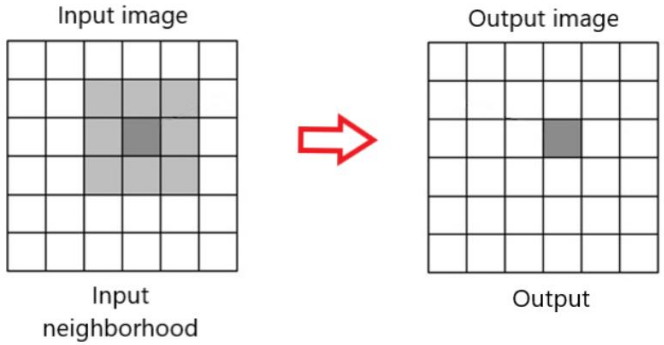
Chapter 2. Filters, Contrast, Transformation and Morphology

You will learn to detect object shapes using edge detection filters, improve medical images with contrast enhancement and even enlarge pictures to five times its original size! You will also apply morphology to make thresholding more accurate when segmenting images and go to the next level of processing images with Python.

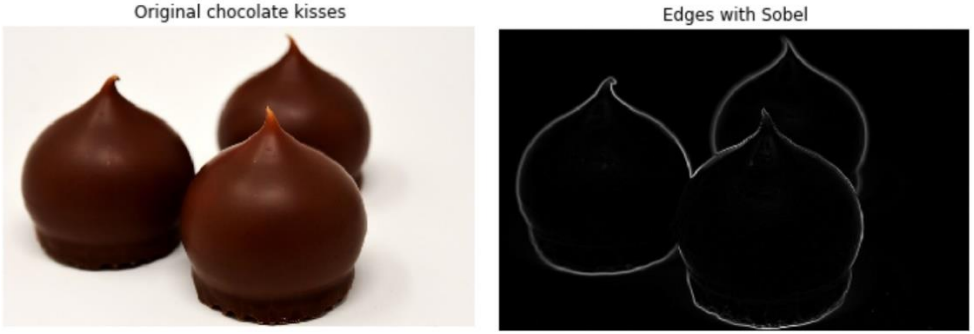
Jump into filtering

Filtering is a technique for modifying or enhancing an image. In essence, a filter is a mathematical function that is applied to images. It can be used to emphasize or remove certain features (like edges), smoothing, sharpening and edge detection.

Neighborhoods
Certain image processing operations involve processing an image in sections, called blocks or neighborhoods, rather than processing the entire image at once. This is the case for filtering, histogram equalization for contrast enhancement, and morphological functions, all three of which use this approach.

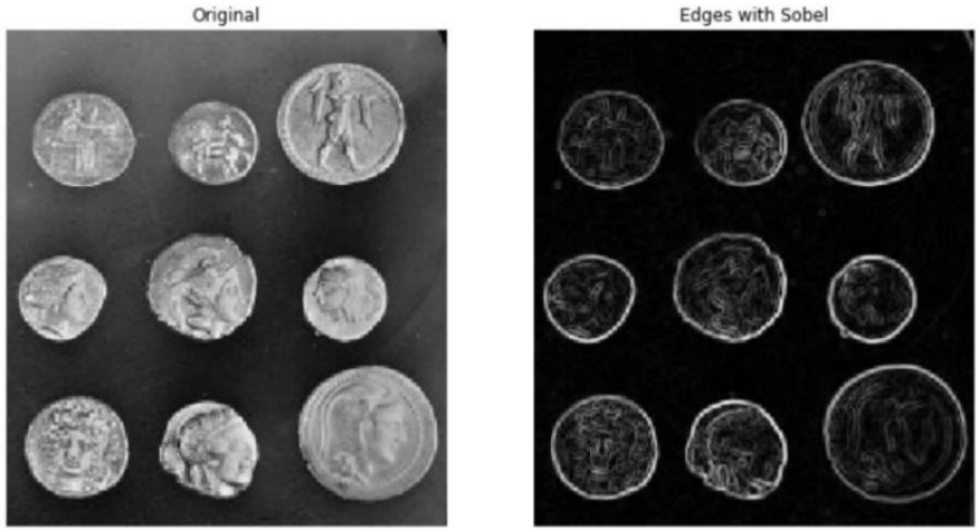


Edge detection
This technique can be used to find the boundaries of objects within images, like in this image, where we spot the chocolate kisses shapes in the image.



Edge detection technique can also segment and extract information like how many coins are in an image. Most of the shape information of an image is enclosed in edges. Edge detection works by detecting discontinuities in brightness. A common edge detection algorithm is Sobel.

```
# Import module and function
from skimage.filters import sobel

# Apply edge detection filter
edge_sobel = sobel(image_coins)

# Show original and resulting image to compare
plot_comparison(image_coins, edge_sobel, "Edge with Sobel")
```



The used coins image is preloaded as image_coins. We apply the filter by passing the image we want to detect the edges from as parameter. This function requires a 2-dimensional grayscale image as input.

So in the case of a colored image, we'll need to convert it to grayscale first.

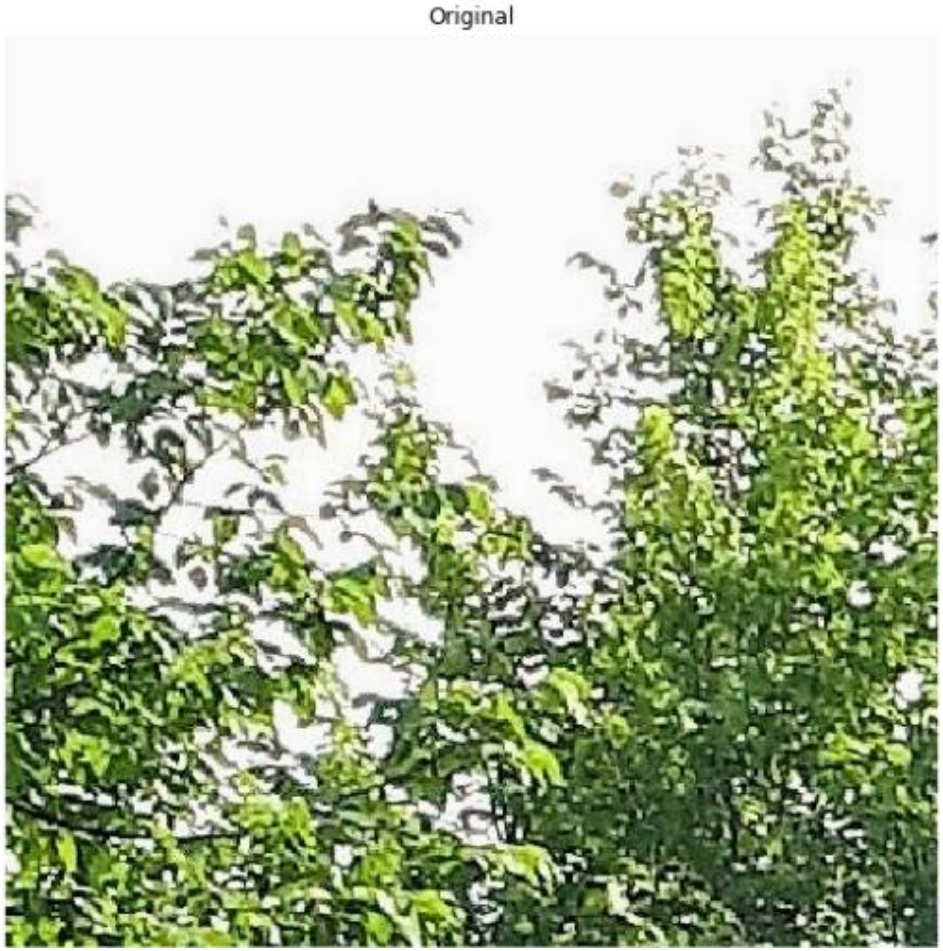
```
def plot_comparison(original, filtered, title_filtered):

    fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(8, 6), sharex=True,
                                   sharey=True)
    ax1.imshow(original, cmap=plt.cm.gray)
    ax1.set_title('original')
    ax1.axis('off')
    ax2.imshow(filtered, cmap=plt.cm.gray)
    ax2.set_title(title_filtered)
    ax2.axis('off')
```

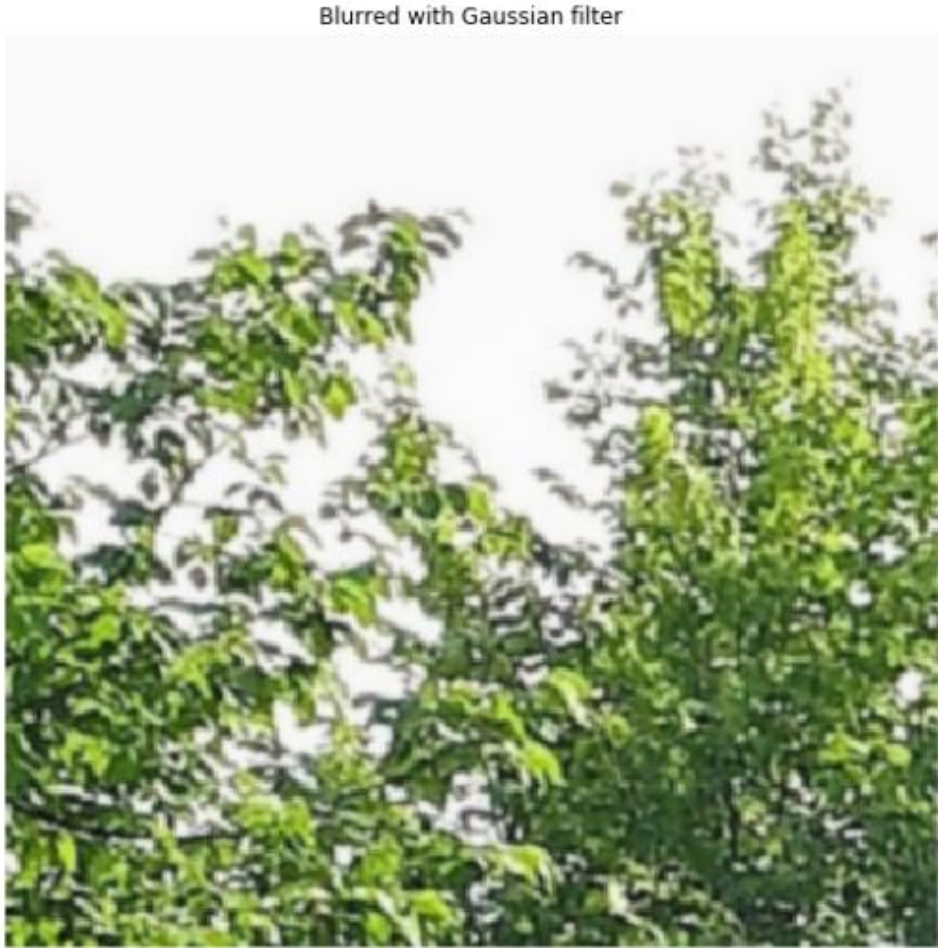
Gaussian smoothing

Another filtering technique is smoothing. We can achieve this with a Gaussian filter. This technique is typically used to blur an image or to reduce noise. Here we see the effect of applying a Gaussian filter, which can especially be seen in the edges of the dogs hair. The Gaussian filter will blur edges and reduce contrast. This is used in other techniques like anti-aliasing filtering.

```
# Import the module and function
from skimage.filters import gaussian

# Apply edge detection filter
gaussian_image = gaussian(amsterdam_pic, multichannel=True)

# Show original and resulting image to compare
plot_comparison(amsterdam_pic, gaussian_image, "Blurred with Gaussian filter")
```

As the image is too large, meaning it has a big resolution, we do not easily see the effect. When zoomed into the image, we can see more clearly how the Guassian filter is blurring the image and removing the noise.





Edge detection

In this exercise, you'll detect edges in an image by applying the Sobel filter. The `show_image()` function has been already loaded for you. Let's see if it spots all the figures in the image.

Image preloaded as `soaps_image`.

```
# Import the color module
from skimage import color

# Import the filters module and sobel function
from skimage.filters import sobel

# Make the image grayscale
soaps_image_gray = color.rgb2gray(soaps_image)

# Apply edge detection filter
edge_sobel = sobel(soaps_image_gray)

# Show original and resulting image to compare
show_image(soaps_image, "Original")
show_image(edge_sobel, "Edges with Sobel")
```

You succesfully detected the edges in the image, as the edges of all the figures in the scene are highlighted.

Blurring to reduce noise

In this exercise you will reduce the sharpness of an image of a building taken during a London trip, through filtering.

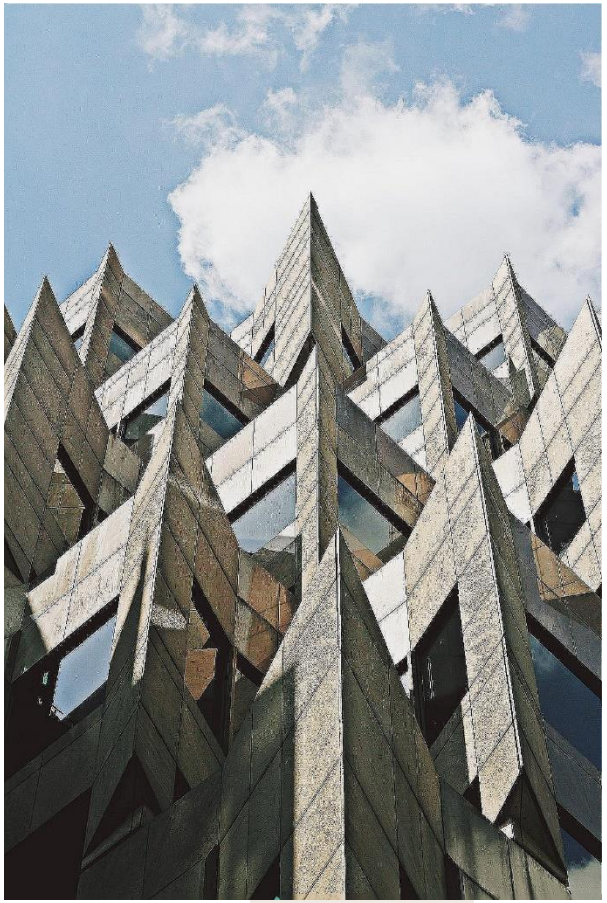


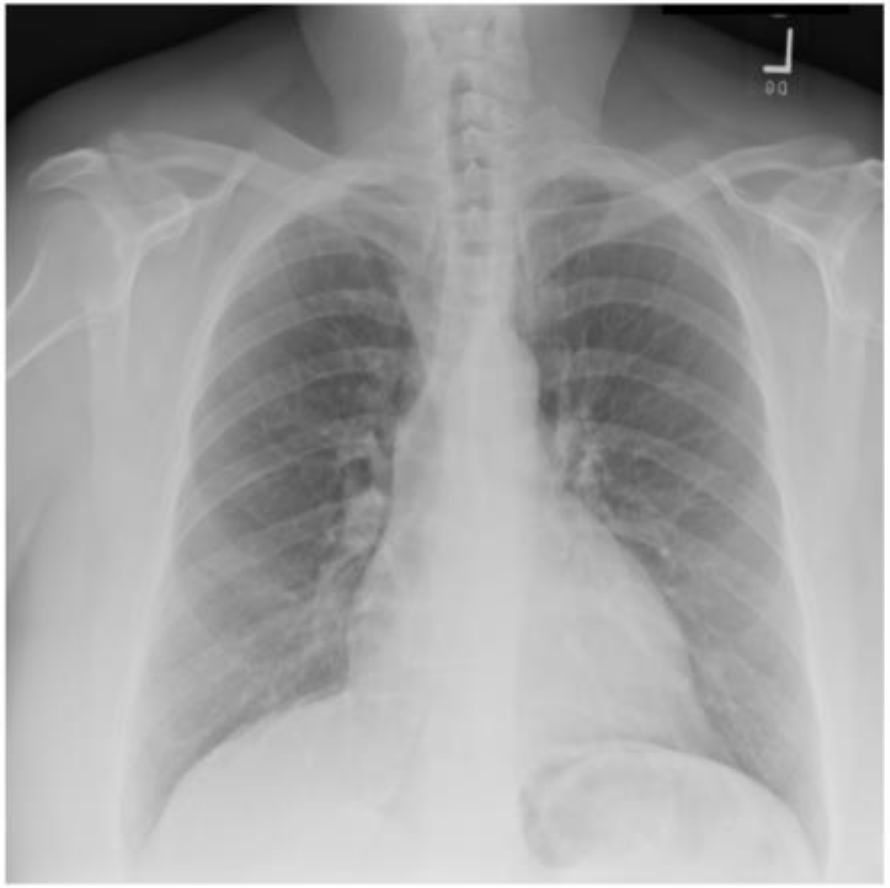
Image loaded as `building_image`.


```
# Import Gaussian filter
from skimage.filters import gaussian

# Apply filter
gaussian_image = gaussian(building_image, multichannel=True)

# Show original and resulting image to compare
show_image(building_image, "Original")
show_image(gaussian_image, "Reduced sharpness Gaussian")
```

You have removed the excessive sharpness in the image.

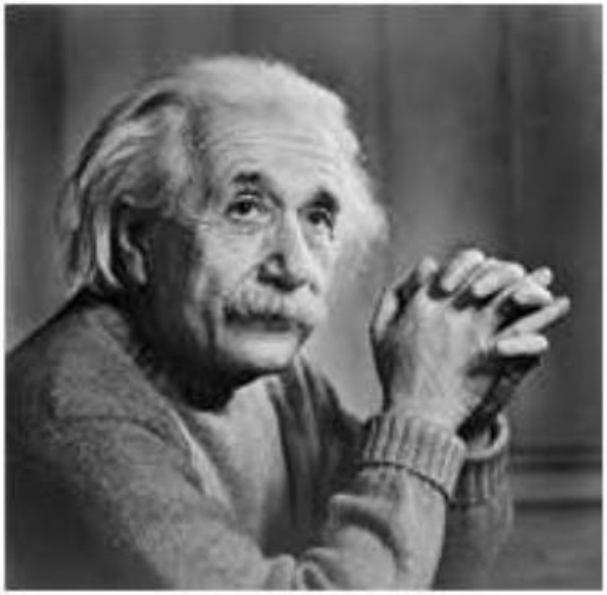
Contrast enhancement

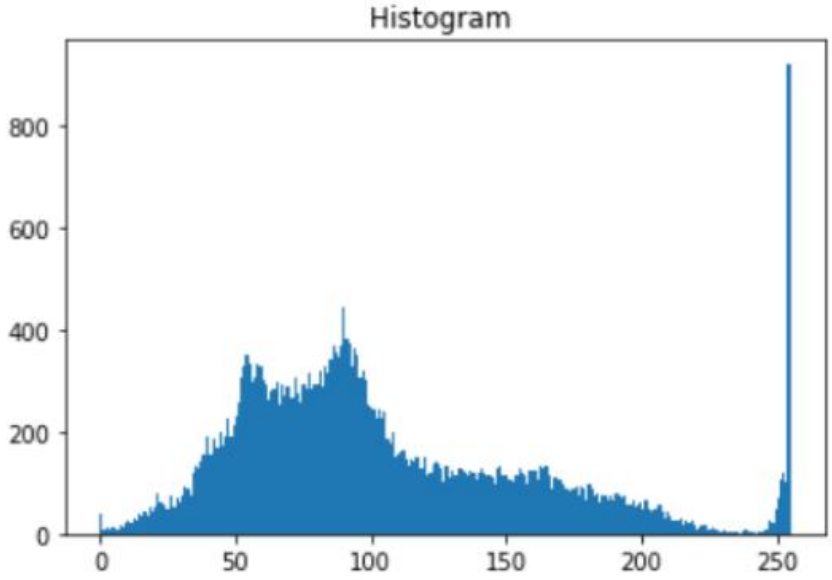
Often medical images like this X-ray can have low contrast, making it hard to spot important details. When we improve the contrast,the details become more visible.



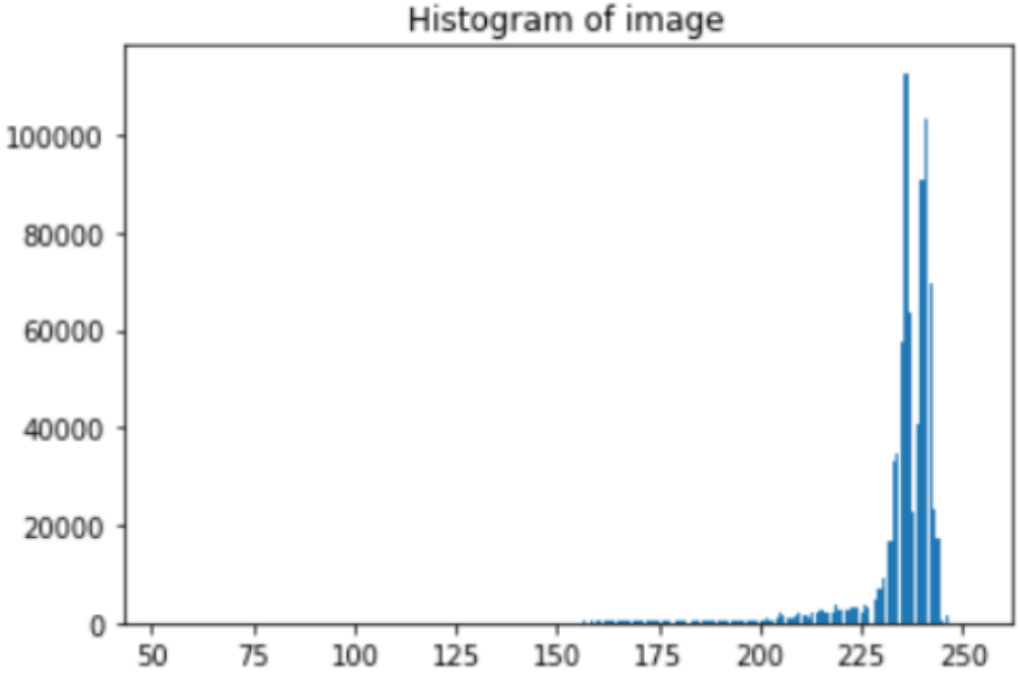
The contrast of an image can be seen as the measure of its dynamic range, or the "spread" of its histogram. The contrast is the difference between the maximum and minimum pixel intensity in the image. The histogram of this image is shown on the right. The maximum value of pixel intensity is 255 while the minimum is 0. 255 - 0 = 255.

Histograms for contrast enhancement





An image of low contrast has small difference between its dark and light pixel values. Is usually skewed either to the right (being mostly light), to the left (when is mostly dark), or located around the middle (mostly gray).

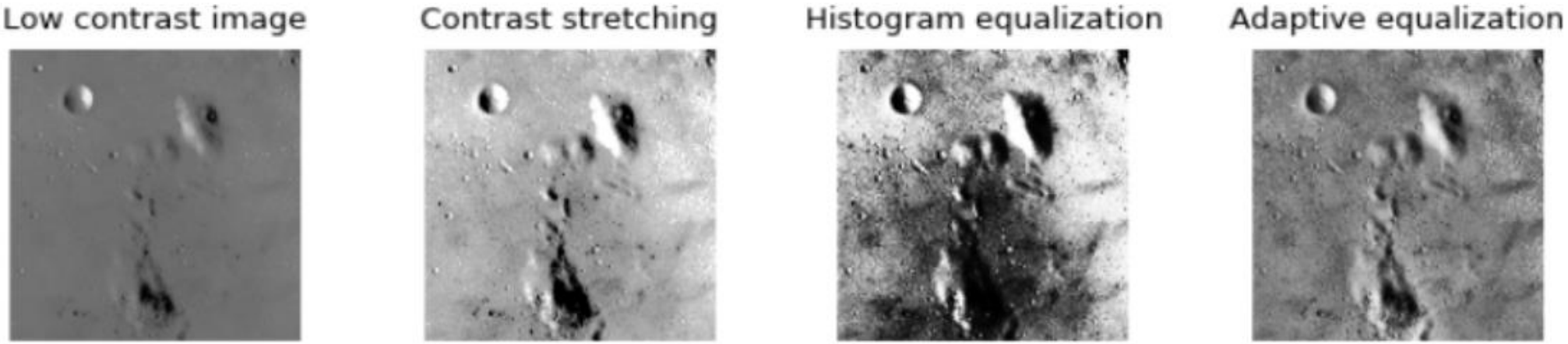


We can enhance contrast through

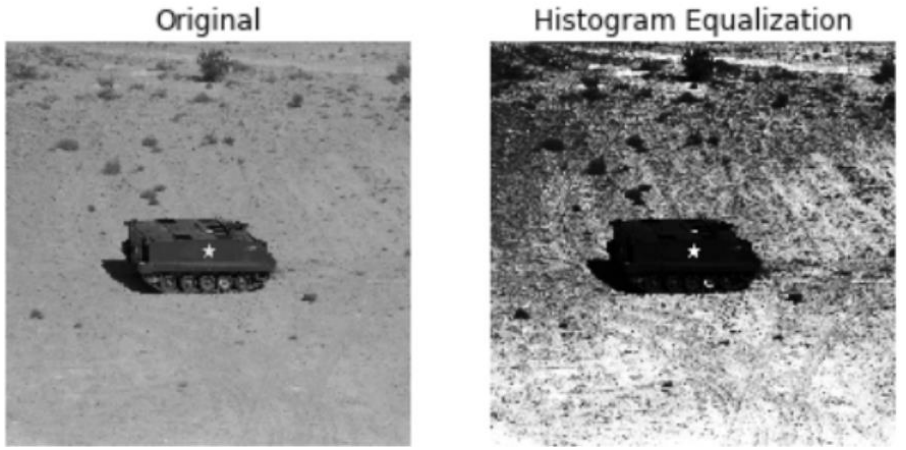
- contrast stretching which is used to stretch the histogram so the full range of intensity values of the image is filled
- histogram equalization, that spreads out the most frequent histogram intensity values using probability distribution.

There are three types of histogram equalization. The standard, the, and the limited adaptive. In scikit-image we can apply

- standard histogram equalization - spreads out the most frequent intensity values
- contrast stretching histogram equalization
- contrast limited adaptive histogram equalization (CLAHE)



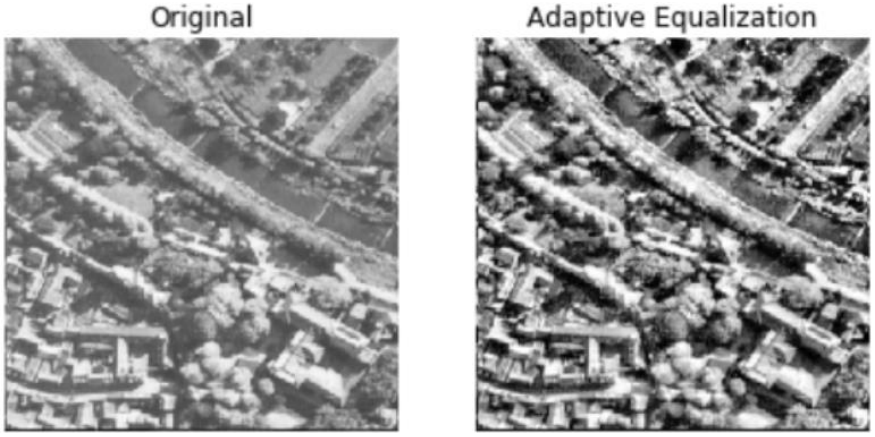

```
from skimage import exposure

# Obtain the equalized image
image_eq = exposure.equalize_hist(image)

# Show original and result
show_image(image, 'Original')
show_image(image_eq, 'Histogram equalized')
```



We get a result that, despite the increased contrast, doesn't look natural. In fact, it doesn't even look like the image has been enhanced at all.

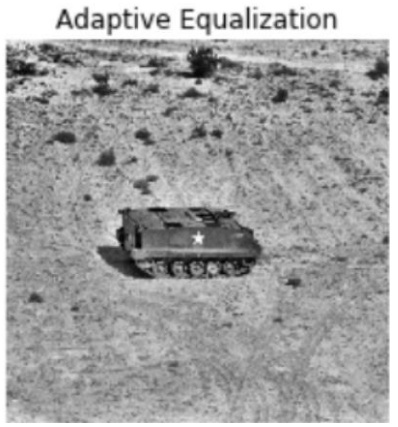
Contrastive Limited Adaptive Histogram Equalization (CLAHE)

Another type of histogram equalization is the adaptive one. This method computes several histograms, each corresponding to a distinct part of the image, and uses them to redistribute the lightness values of the image histogram. This method was developed to prevent over-amplification of noise that adaptive histogram equalization can give rise to. In this image, we see the result of the CLAHE method and it may seem very similar to the standard method.



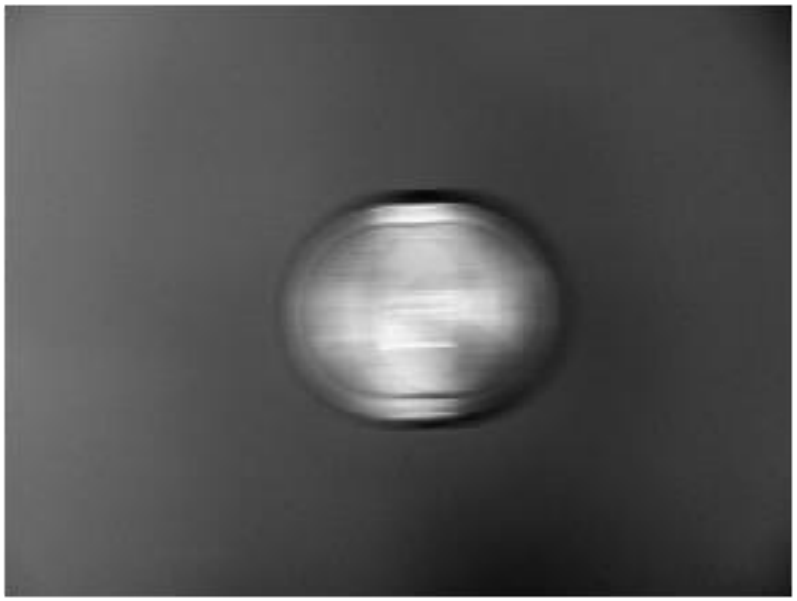
But if you look closer and compare the results, you will see that the adaptive method is not that intense, so it looks more natural. This is because it is not taking the global histogram of the entire image, but operates on small regions called tiles or neighborhoods.

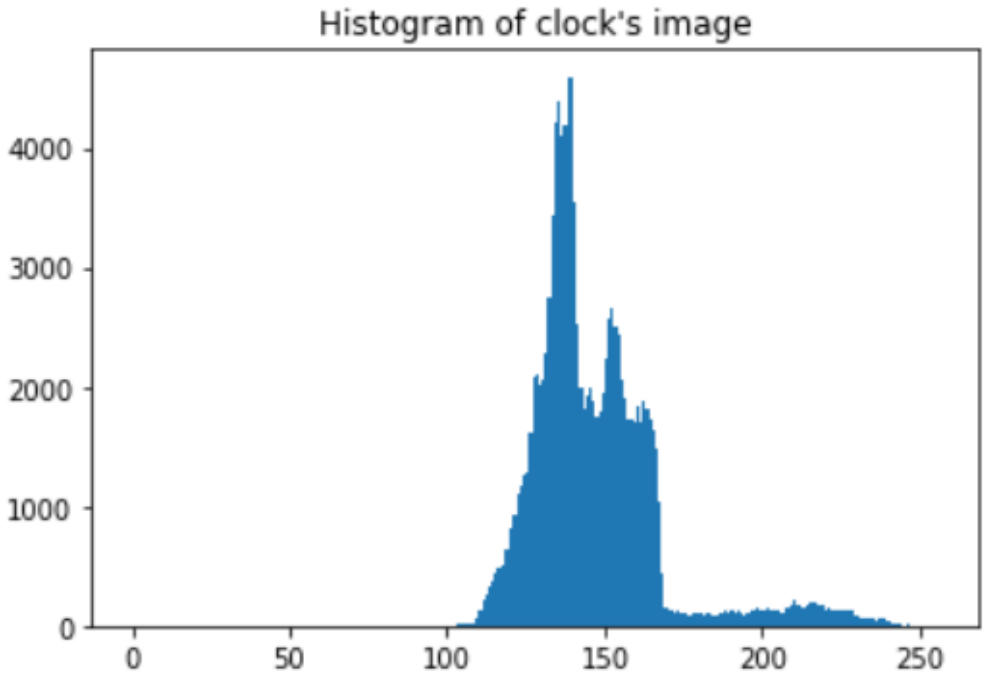
```python
from skimage import exposure

# Apply adaptive Equalization
image_adapteq = exposure.equalize_adapthist(image, clip_limit=0.03)

# Show original and result
show_image(image, 'Original')
show_image(image_adapteq, 'Adaptive equalized')
```



Comparing them, the resulting image is enhanced and we can better detail small objects and figures (like the footprints in the ground).

What's the contrast of this image?





The histogram tell us.

Just as we saw previously, you can calculate the contrast by calculating the **range** of the pixel intensities i.e. by subtracting the minimum pixel intensity value from the **histogram** to the maximum one.

You can obtain the maximum pixel intensity of the image by using the `np.max()` method from NumPy and the minimum with `np.min()` **in the console**.

The image has already been loaded as `clock_image`, NumPy as `np` and the `show_image()` function.

- ○ The contrast is 255 (high contrast).
- ◉ The contrast is 148.
- ○ The contrast is 189.
- ○ The contrast is 49 (low contrast).

```
In [1]: np.max(clock_image) - np.min(clock_image)
Out[1]: 148
```

You calculated the range of the pixels intensities in the histogram, and so, the contrast of the image!

Medical images

You are trying to improve the tools of a hospital by pre-processing the X-ray images so that doctors have a higher chance of spotting relevant details. You'll test our code on a chest X-ray image from the **National Institutes of Health Chest X-Ray Dataset**

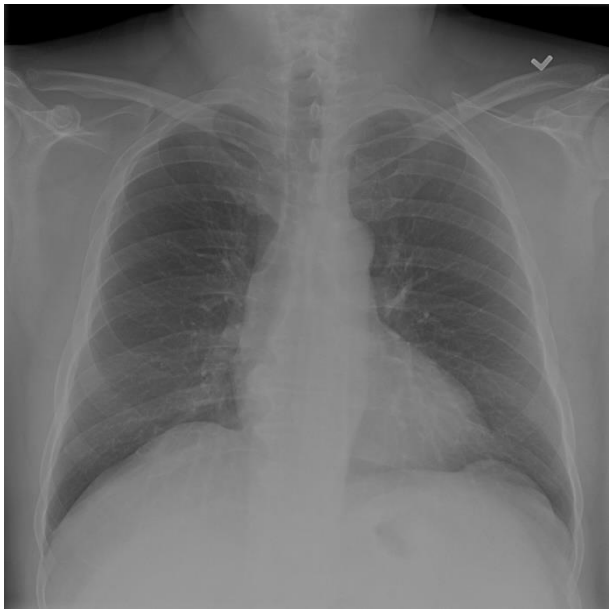
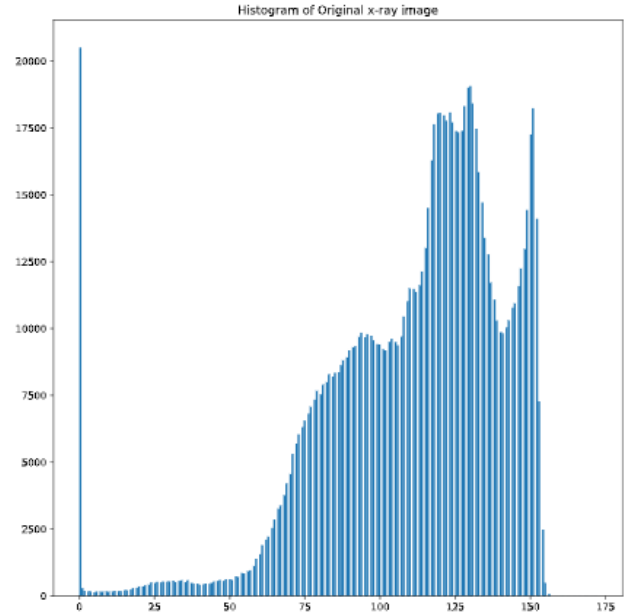
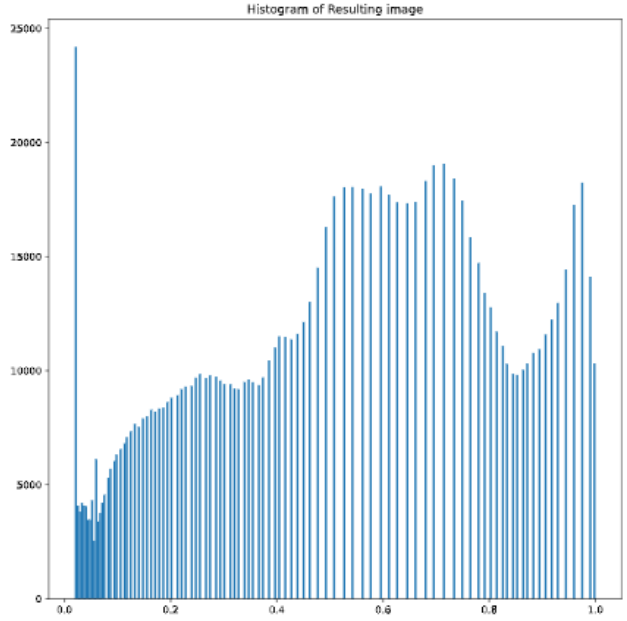


Image loaded as `chest_xray_image`.

First, you'll check the histogram of the image and then apply standard histogram equalization to improve the contrast. Remember we obtain the histogram by using the `hist()` function from Matplotlib, which has been already imported as `plt`.

```
# Import the required module
from skimage import exposure

# Show original x-ray image and its histogram
show_image(chest_xray_image, 'Original x-ray')

plt.title('Histogram of Original x-ray image')
plt.hist(chest_xray_image.ravel(), bins=256)
plt.show()

# Use histogram equalization to improve the contrast
xray_image_eq =  exposure.equalize_hist(chest_xray_image)

# Show the resulting image
show_image(xray_image_eq, 'Resulting image')

plt.title('Histogram of Resulting image')
plt.hist(xray_image_eq.ravel(), bins=256)
plt.show()
```





Now you can apply this code and knowledge to other similar images.

Aerial image

In this exercise, we will improve the quality of an aerial image of a city. The image has low contrast and therefore we can not distinguish all the elements in it. For this we will use the normal or standard technique of Histogram Equalization.

Image loaded as `image_aerial`.

```
# Import the required module
from skimage import exposure

# Use histogram equalization to improve the contrast
image_eq =  exposure.equalize_hist(image_aerial)

# Show the original and resulting image
show_image(image_aerial, 'Original')
show_image(image_eq, 'Resulting image')
```

Now we can see more details of the objects in the image.

Let's add some impact and contrast

Have you ever wanted to enhance the contrast of your photos so that they appear more dramatic? In this exercise, you'll increase the contrast of a cup of coffee. Something you could share with your friends on social media. Don't forget to use **#ImageProcessingDatacamp** as hashtag!

Even though this is not our Sunday morning coffee cup, you can still apply the same methods to any of our photos.

A function called `show_image()`, that displays an image using Matplotlib, has already been defined. It has the arguments `image` and `title`, with `title` being `'Original'` by default.


```
# Import the necessary modules
from skimage import data, exposure

# Load the image
original_image = data.coffee()

# Apply the adaptive equalization on the original image
adapthist_eq_image = exposure.equalize_adapthist(original_image, clip_limit=0.03)

# Compare the original image to the equalized
show_image(original_image)
show_image(adapthist_eq_image, '#ImageProcessingDatacamp')
```

You have increased the contrast of the image using an algorithm for local contrast enhancement, that uses histograms computed over different tile regions of the image. Local details can therefore be enhanced even in regions that are darker or lighter than the rest of the image.

Transformations

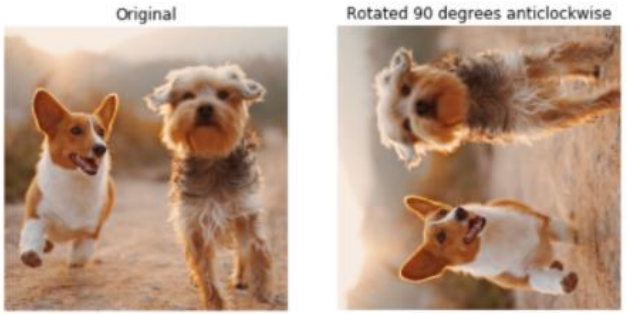
Why transform images?

- need to pass images to a Machine Learning model so it can classify if it's a cat or a dog, and we want the image to be upright.
- to optimize the size of images so it doesn't take long to analyze them.
- need all the images to have the same proportion before processing them further.

Rotating

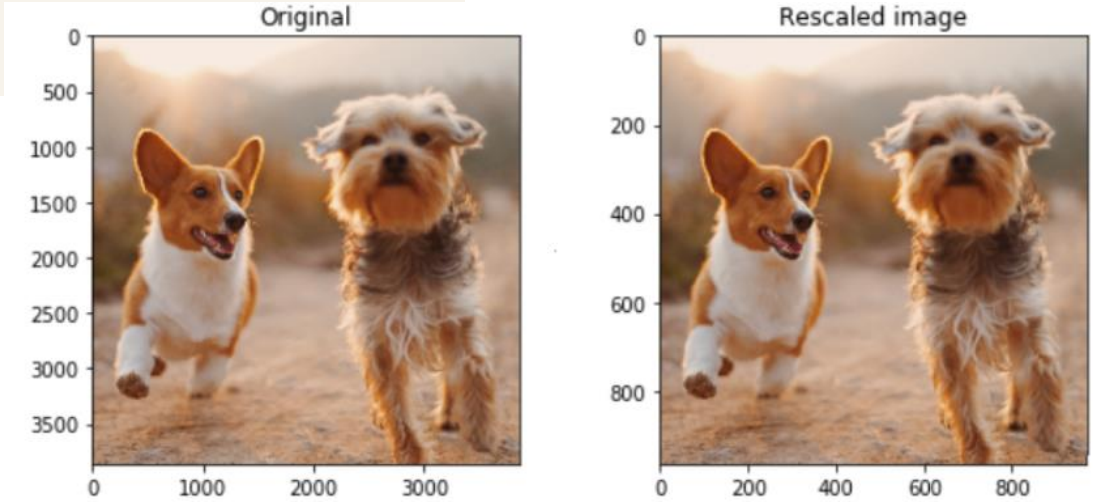
```
from skimage.transform import rotate

# Rotate the image 90 degrees clockwise
image_rotated = rotate(image, -90)

show_image(image, 'Original')
show_image(image_rotated, 'Rotated 90 degrees clockwise')
```

```python
from skimage.transform import rotate

# Rotate an image 90 degrees anticlockwise
image_rotated = rotate(image, 90)

show_image(image, 'Original')
show_image(image_rotated, 'Rotated 90 degrees anticlockwise')
```

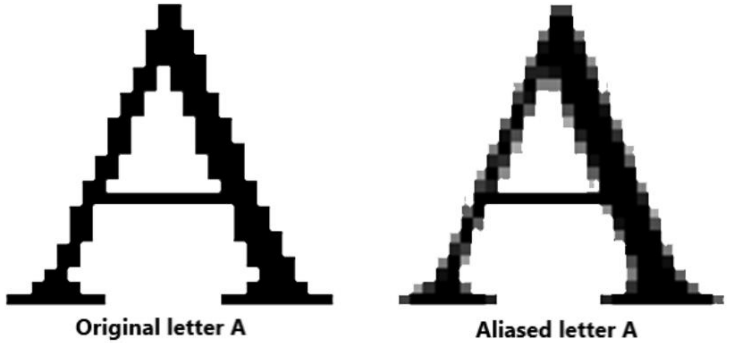


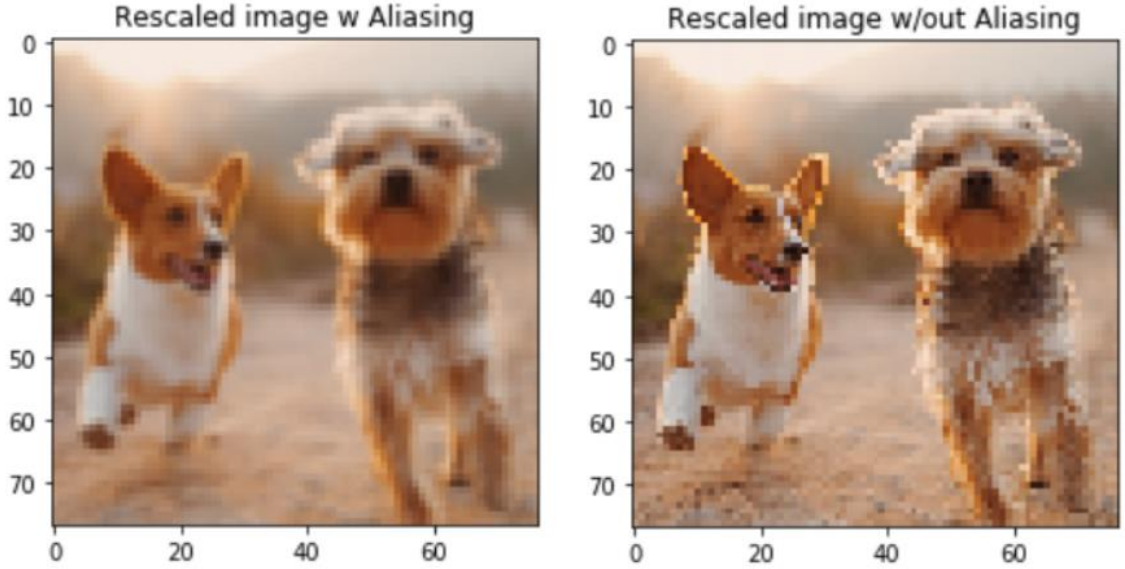
Rescaling

```python
from skimage.transform import rescale

# Rescale the image to be 4 times smaller
image_rescaled = rescale(image, 1/4, anti_aliasing=True, multichannel=True)

show_image(image, 'Original image')
show_image(image_rescaled, 'Rescaled image')
```



Anti-aliasing in digital images
In a digital image, aliasing is a pattern or a rippling effect. Aliasing makes the image look like it has waves or ripples radiating from a certain portion. This happens because the pixelation of the image is poor and does not look smooth.

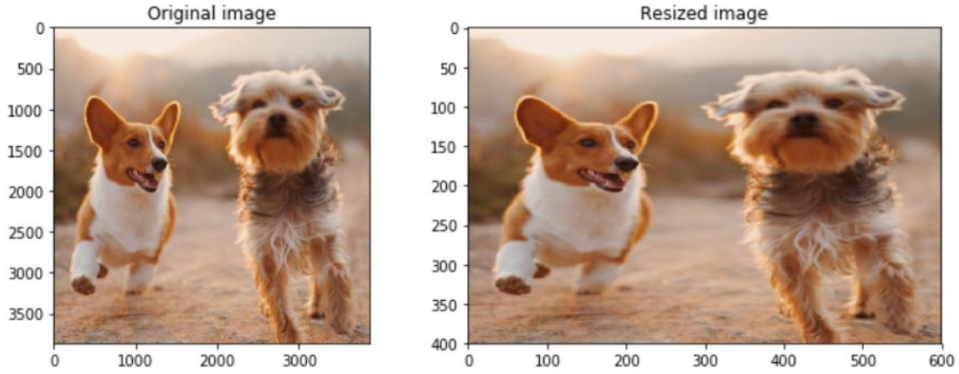




Here, we applied a resizing of 1/30, and we see what the anti_aliasing filter is doing to the image when is set. The first one has the anti_aliasing to True so we see is softer. While the one without it is pixelated.

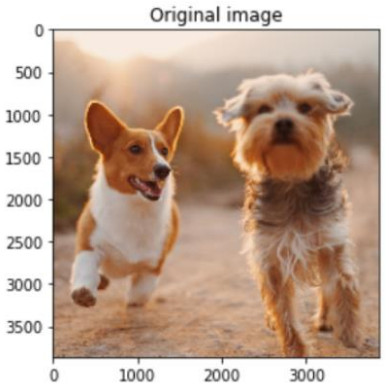
```python
from skimage.transform import resize

# Height and width to resize
height = 400
width = 600

# Resize image
image_resized = resize(image, (height, width), anti_aliasing=True)

# Show the original and resulting images
show_image(image, 'Original image')
show_image(image_resized, 'Resized image')
```

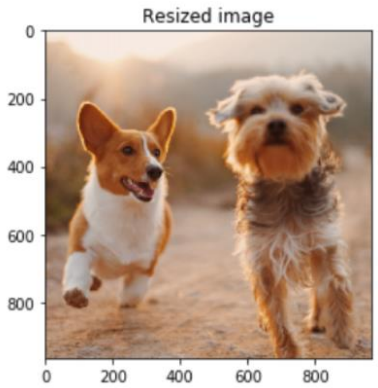
Resizing is used for making images match a certain size. The same purpose as rescale, but allows to specify an output image shape instead of a scaling factor.



Resizing proportionally

```
from skimage.transform import resize

# Set proportional height so its 4 times its size
height = image.shape[0] / 4
width = image.shape[1] / 4

# Resize image
image_resized = resize(image, (height, width), anti_aliasing=True)

show_image(image_resized, 'Resized image')
```





Aliasing, rotating and rescaling

Let's look at the impact of aliasing on images.

Remember that aliasing is an effect that causes different signals, in this case pixels, to become indistinguishable or distorted.

You'll make this cat image upright by rotating it 90 degrees and then rescaling it two times. Once with the anti aliasing filter applied before rescaling and a second time without it, so you can compare them.

Image preloaded as `image_cat`.

```
# Import the module and the rotate and rescale functions
from skimage.transform import rotate, rescale

# Rotate the image 90 degrees clockwise
rotated_cat_image = rotate(image_cat, -90)

# Rescale with anti aliasing
rescaled_with_aa = rescale(rotated_cat_image, 1/4, anti_aliasing=True, multichannel=True)

# Rescale without anti aliasing
rescaled_without_aa = rescale(rotated_cat_image, 1/4, anti_aliasing=False, multichannel=True)

# Show the resulting images
show_image(rescaled_with_aa, "Transformed with anti aliasing")
show_image(rescaled_without_aa, "Transformed without anti aliasing")
```

You rotated and rescaled the image. Seems like the anti-aliasing filter prevents the poor pixelation effect to happen, making it look better but also less sharp.

Enlarging images

Have you ever tried resizing an image to make it larger? This usually results in loss of quality, with the enlarged image looking blurry.

The good news is that the algorithm used by scikit-image works very well for enlarging images up to a certain point.

In this exercise you'll enlarge an image **three times**!!

You'll do this by rescaling the image of a rocket, that will be loaded from the `data` module.

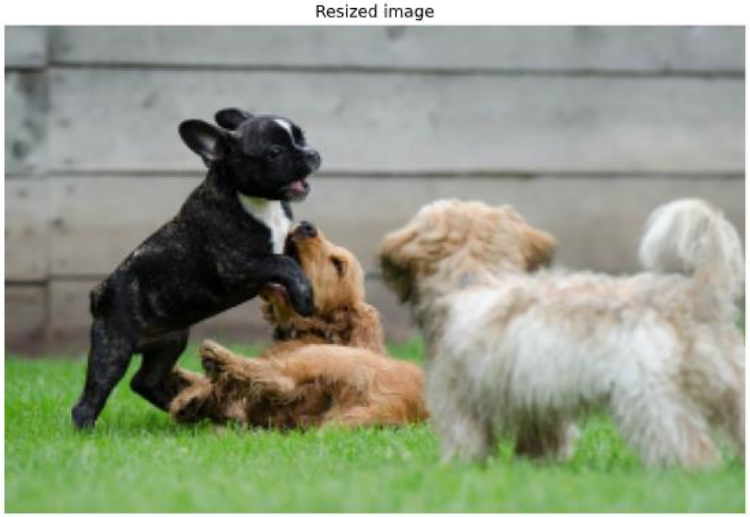

```
# Import the module and function to enlarge images
from skimage.transform import rescale

# Import the data module
from skimage import data

# Load the image from data
rocket_image = data.rocket()

# Enlarge the image so it is 3 times bigger
enlarged_rocket_image = rescale(rocket_image, 3, anti_aliasing=True, multichannel=True)

# Show original and resulting image
show_image(rocket_image)
show_image(enlarged_rocket_image, "3 times enlarged image")
```

The image went from 600 pixels wide to over 2500 and it still does not look poorly pixelated. Nice work!

Proportionally resizing

We want to downscale the images of a veterinary blog website so all of them have the same compressed size. It's important that you do this proportionally, meaning that these are not distorted. First, you'll try it out for one image so you know what code to test later in the rest of the pictures. Remember that by looking at the shape of the image, you can know its width and height.

The image preloaded as `dogs_banner`.




```
# Import the module and function
from skimage.transform import resize

# Set proportional height and width so it is half its size
height = int(dogs_banner.shape[0] / 2.0)
width = int(dogs_banner.shape[1] / 2.0)

# Resize using the calculated proportional height and width
image_resized = resize(dogs_banner, (height, width),
                       anti_aliasing=True)

# Show the original and rotated image
show_image(dogs_banner, 'Original')
show_image(image_resized, 'Resized image')
```

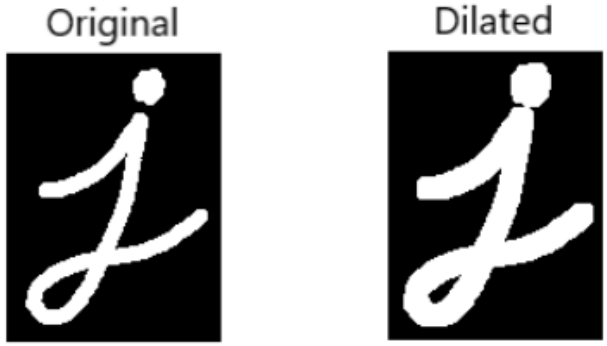
The image is now compressed and ready. We can use this code for future images uploaded to the website.

Morphology

When you try to spot objects in an image, you can do so by its characteristics, like the shape. This what Morphology does. Binary regions produced by simple thresholding can be distorted by noise and texture, as we can see in the image.

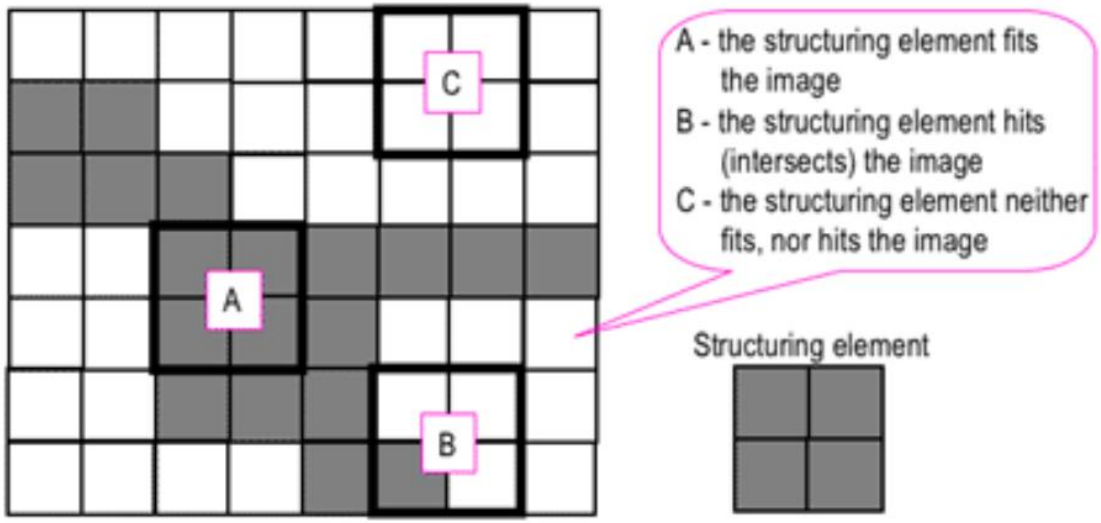
Morphological filtering operations try to remove these imperfections by accounting for the form and structure of the objects in the image. These operations are especially suited to binary images, but some can extend to grayscale ones.

Basic morphological operations are:

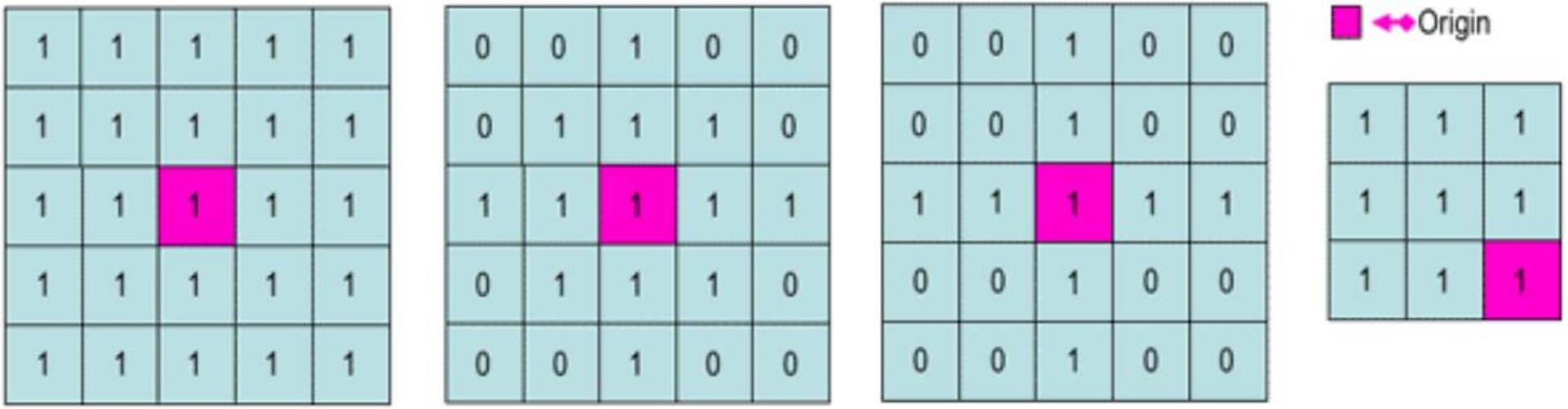
Dilation - adds pixels to the boundaries of objects	**Erosion** - removes pixels on object boundaries
	

The number of pixels added or removed from the objects in an image depends on the size and shape of a structuring element used to process the image.

The structuring element is a small binary image used to probe the input image. We try to "fit" in the image object we want to get its shape.



So if we want to select an apple in a table, we want the structuring element fit in that apple so then expands, probe and obtain the shape.



The dimensions specify the size of the structuring element. Like a square of 5 by 5 pixels. The pattern of ones and zeros specifies the shape of the structuring element. This should be of a similar form to the shape of the

object we want to select. So we see in here different types of shapes, from squares, to diamond. The pink cell is the center or origin of the structuring element. Identifies the pixel being processed.

scikit-image has multiple shapes for this structured element, each one with its own method from the morphology module. If we want square as the structured element, we can obtain it with the square method. Or a rectangle with width and height. This will return the desired shape and if we print we'll see how these are formed with 1s.

```python
from skimage import morphology

square = morphology.square(4)
```

```
[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]
```

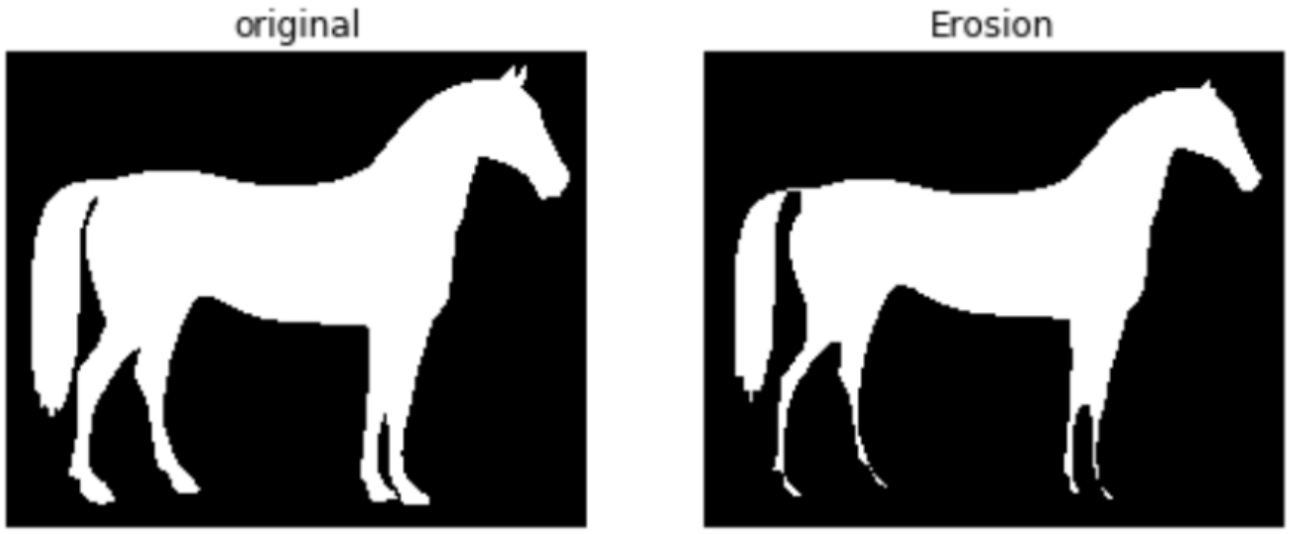
```python
rectangle = morphology.rectangle(4, 2)
```

```
[[1 1]
 [1 1]
 [1 1]
 [1 1]]
```

To apply erosion we can use the binary erosion function. With this we can optionally set a structuring element to use in the operation. Here we import it and load a binary horse image. Set the structuring element to a rectangular-shaped, since it's somewhat similar to the shape we want to obtain, which is a horse. And obtain the eroded image by using this function, passing the image and structuring element as parameters. If not set, the function will use a cross-shaped structured element by default.

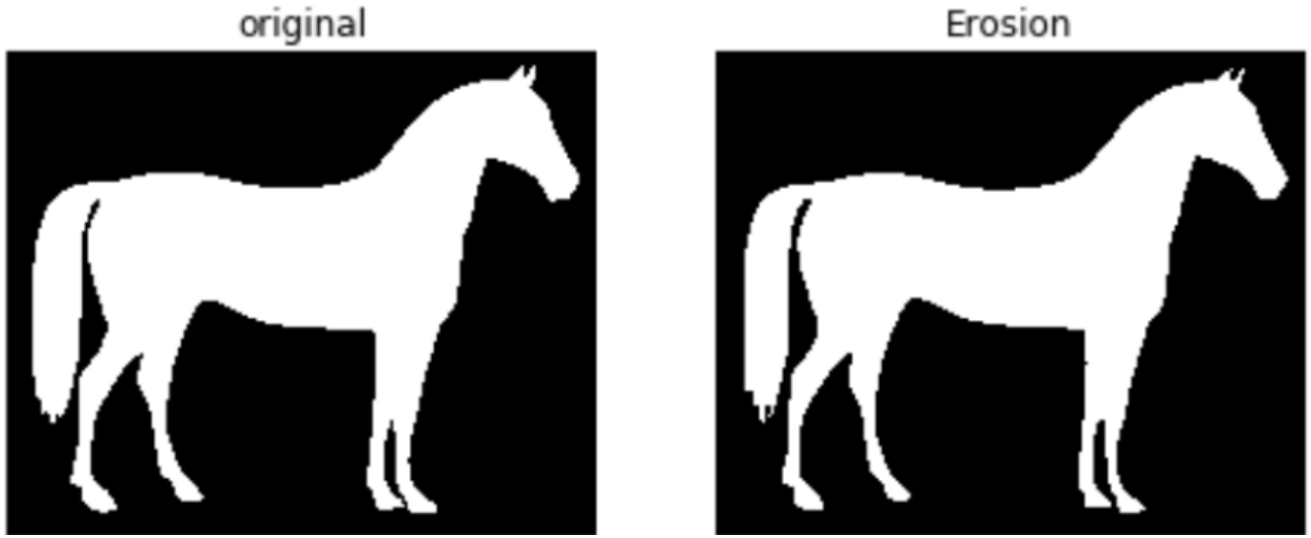
```python
from skimage import morphology

# Set structuring element to the rectangular-shaped
selem = rectangle(12,6)

# Obtain the erosed image with binary erosion
eroded_image = morphology.binary_erosion(image_horse, selem=selem)
```

```
# Show result
plot_comparison(image_horse, eroded_image, 'Erosion')
```



Showing the resulting image, next to the original to compare them, we see that the resulted image is missing some pixels. But still kind of showing the horse shape.

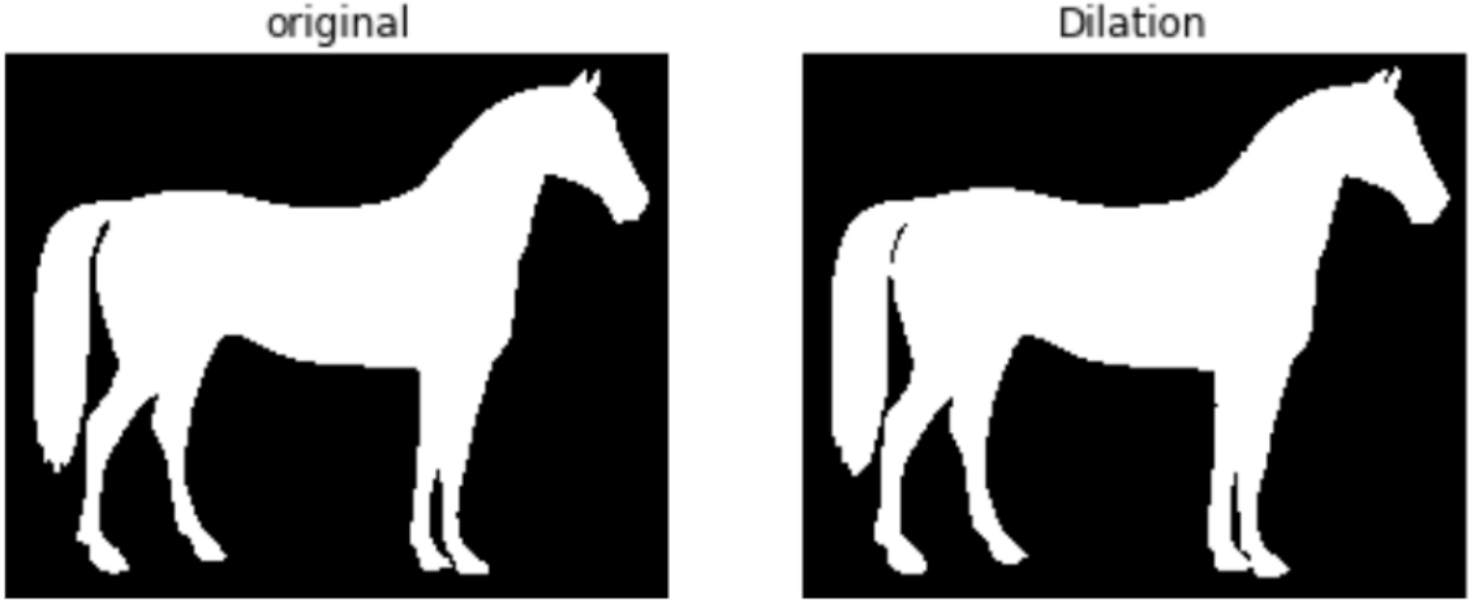
```
# Binary erosion with default selem
eroded_image = morphology.binary_erosion(image_horse)
```



If we apply binary erosion with default structuring element shape, we would obtain this eroded image. It's working better, more accurate than before. So for this image the cross-shaped works great.

Dislation in scikit-image
Dilation operation sort of "expands" the objects in the image. Here, we use binary dilation function, also from the morphology module, on image_horse. Let's use the default structuring element, which is cross-shaped.

```
from skimage import morphology

# Obtain dilated image, using binary dilation
dilated_image = morphology.binary_dilation(image_horse)

# See results
plot_comparison(image_horse, dilated_image, 'Erosion')
```



We see that dilation is indeed adding just a little bit in some parts, like the lower legs and paws. We see that the default structuring element works well.

Handwritten letters

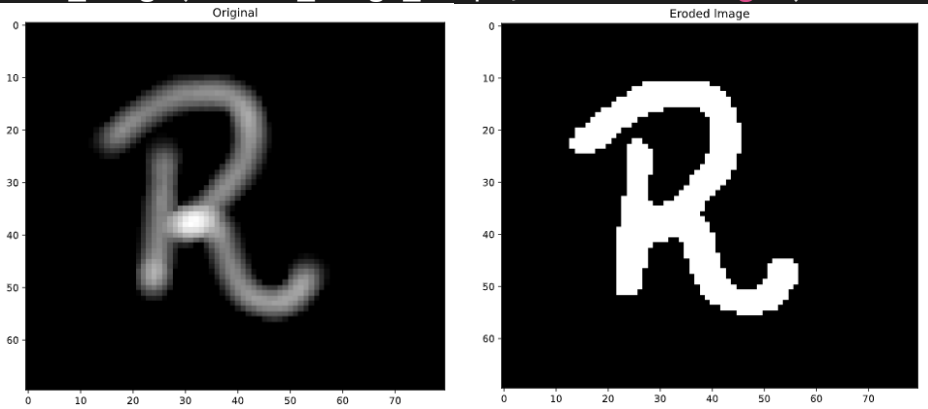
A very interesting use of computer vision in real-life solutions is performing Optical Character Recognition (**OCR**) to distinguish printed or handwritten text characters inside digital images of physical documents.

Let's try to improve the definition of this handwritten letter so that it's easier to classify.

As we can see it's the letter *R*, already binary, with some noise in it. It's already loaded as `upper_r_image`.

Apply the morphological operation that will discard the pixels near the letter boundaries.

```
# Import the morphology module
from skimage import morphology

# Obtain the eroded shape
eroded_image_shape = morphology.binary_erosion(upper_r_image)

# See results
show_image(upper_r_image, 'Original')
show_image(eroded_image_shape, 'Eroded image')
```



As you can see, erosion is useful for removing minor white noise.

Improving thresholded image

In this exercise, we'll try to reduce the noise of a thresholded image using the dilation morphological operation.

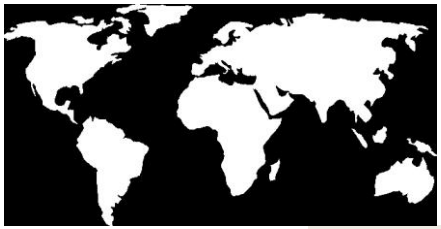
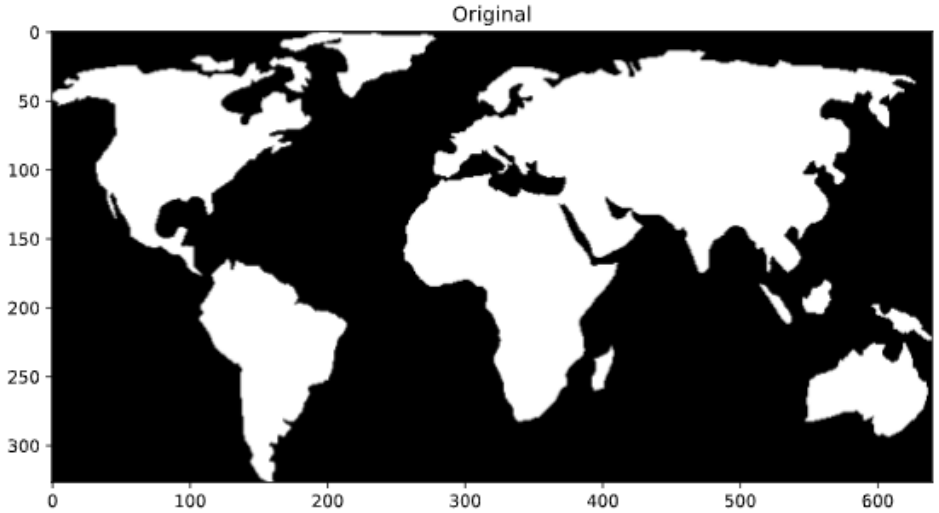


Image already loaded as `world_image`.

This operation, in a way, *expands* the objects in the image.

```
# Import the module
from skimage import morphology

# Obtain the dilated image
dilated_image = morphology.binary_dilation(world_image)

# See results
show_image(world_image, 'Original')
show_image(dilated_image, 'Dilated image')
```

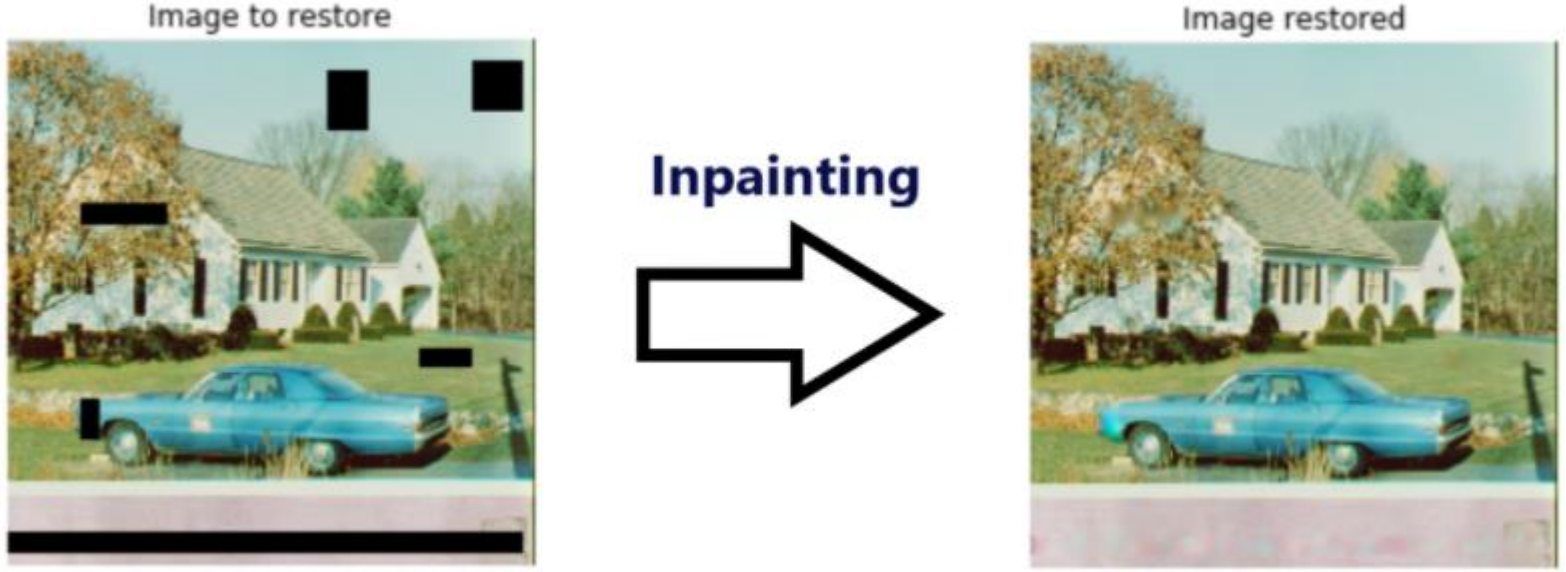


You removed the noise of the segmented image and now it's more uniform.

Chapter 3. Image restoration, Noise, Segmentation and Contours

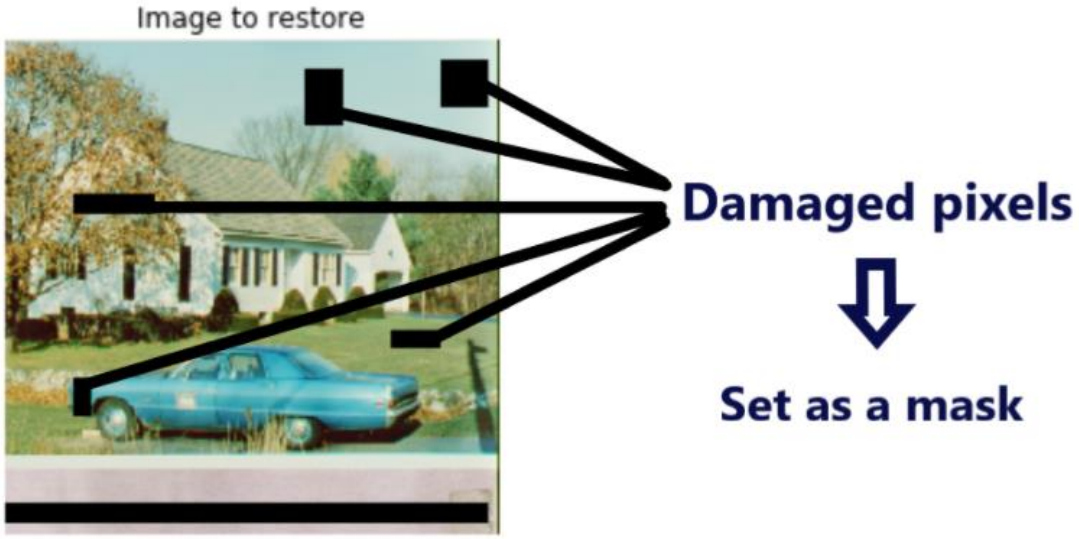
So far, you have done some very cool things with your image processing skills! In this chapter, you will apply image restoration to remove objects, logos, text, or damaged areas in pictures! You will also learn how to apply noise, use segmentation to speed up processing, and find elements in images by their contours.

Image restoration



Besides fixing damaged images, image restoration or reconstruction, is also used for text removing, deleting logos from images and even removing small objects, like tattoos you prefer not to show on a picture.

Reconstructing lost or deteriorated parts of images is known as **inpainting**. The reconstruction is supposed to be performed in a fully automatic way by exploiting the information presented in non-damaged regions of the image.



In scikit-image, we can apply inpainting with the inpaint biharmonic function, from the restoration module. It needs the location of the damaged pixels to be filled, as a mask image on top of the image to work with. A mask image is simply an image where some of the pixel intensity values are zero, and others are non-zero.

```
from skimage.restoration import inpaint

# Obtain the mask
mask = get_mask(defect_image)

# Apply inpainting to the damaged image using the mask
restored_image = inpaint.inpaint_biharmonic(defect_image,
                                            mask,
                                            multichannel=True)

# Show the resulting image
show_image(restored_image)
```

```
# Show the defect and resulting images
show_image(defect_image, 'Image to restore')
show_image(restored_image, 'Image restored')
```

In this example, we can see how the masked pixels get inpainted by the inpainting algorithm based on the biharmonic equation assumption.

Mask

Imagine you have an old picture of your parents you want to fix. In this image, we intentionally added the missing pixels by setting them to black. In case you want to remove an object you can manually delineate it in the mask. And if you want to automatically detect it, you would need to use Thresholding or segmentation to do so. Something we will learn later on. In the right image we see the damaged areas of the image as a mask.

Image to restore

Mask

```python
def get_mask(image):
    ''' Creates mask with three defect regions '''
    mask = np.zeros(image.shape[:-1])

    mask[101:106, 0:240] = 1

    mask[152:154, 0:60] = 1
    mask[153:155, 60:100] = 1
    mask[154:156, 100:120] = 1
    mask[155:156, 120:140] = 1

    mask[212:217, 0:150] = 1
    mask[217:222, 150:256] = 1
    return mask
```

The scikit-image inpainting function requires the mask to be an array of pixels that need to be inpainted. This mask has to be the same shape as one of the image channels. Unknown pixels have to be represented with 1 and known pixels with 0. So we add the missing pixels by copying the image and turning the pixels into a numpy array of zeros, meaning it's empty. We only copy the width and height dimensions of the image, excluding the color dimension, in this case RGB-3. And then, set the 1s in the specific areas we want to be treated as lost.

Let's restore a damaged image

In this exercise, we'll restore an image that has missing parts in it, using the `inpaint_biharmonic()` function.

Loaded as `defect_image`.

We'll work on an image from the `data` module, obtained by `data.astronaut()`. Some of the pixels have been replaced by 1s using a binary mask, on purpose, to simulate a damaged image. Replacing pixels with 1s turns them totally black. The defective image is saved as an array called `defect_image`.

The mask is a black and white image with patches that have the position of the image bits that have been corrupted. We can apply the restoration function on these areas. This mask is preloaded as `mask`.

Remember that inpainting is the process of reconstructing lost or deteriorated parts of images and videos.

```
# Import the module from restoration
from skimage.restoration import inpaint

# Show the defective image
show_image(defect_image, 'Image to restore')

# Apply the restoration function to the image using the mask
restored_image = inpaint.inpaint_biharmonic(defect_image, mask, multichannel=True)
show_image(restored_image)
```

You restored the image successfully. The image looks a lot better now. You can handle colored images that have several missing areas.

Removing logos

As we saw in the video, another use of image restoration is removing objects from an scene. In this exercise, we'll remove the Datacamp logo from an image.

Image loaded as `image_with_logo`.

You will create and set the mask to be able to erase the logo by inpainting this area.

Remember that when you want to remove an object from an image you can either manually delineate that object or run some image analysis algorithm to find it.

```
# Initialize the mask
mask = np.zeros(image_with_logo.shape[:-1])

# Set the pixels where the logo is to 1
mask[210:272, 360:425] = 1

# Apply inpainting to remove the logo
```

```
image_logo_removed = inpaint.inpaint_biharmonic(image_with_logo,
                                                mask,
                                                multichannel=True)

# Show the original and logo removed images
show_image(image_with_logo, 'Image with logo')
show_image(image_logo_removed, 'Image with logo removed')
```

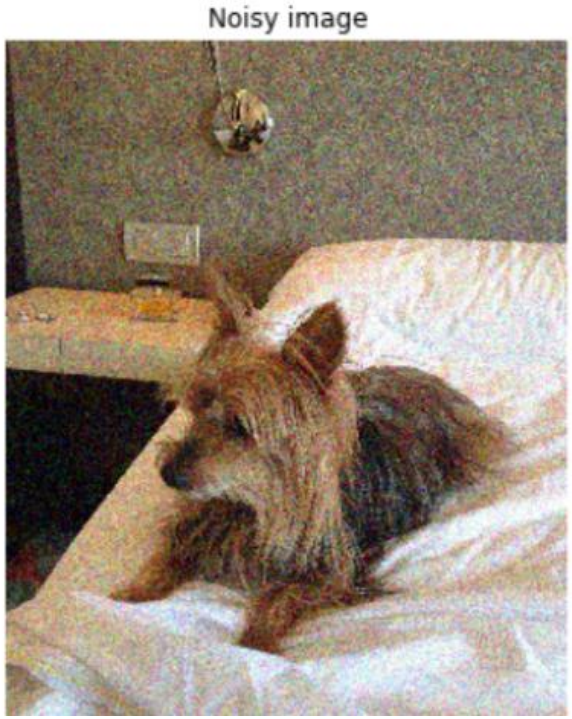
Now you know how you can remove objects or logos from images. Be wise in how you use your new acquired knowledge, magician.

Noise

Images are signals and real-world signals usually contain departures from the ideal signal, which is the perfect image, as we observe with our eyes in real life. Such departures are referred to as noise. We can see how this image has some color grains when zoomed in.

More specifically, noise is the result of errors in the image acquisition process that result in pixel values that do not reflect the true intensities of the real scene. In this image we can see how there is a variation of brightness and color that does not correspond to reality, which is produced by the camera.

```
# Import the module and function
from skimage.util import random_noise

# Add noise to the image
noisy_image = random_noise(dog_image)

# Show original and resulting image
show_image(dog_image)
show_image(noisy_image, 'Noisy image')
```



By using the random_noise function, we obtain the original image with a lot of added noise, that is distributed randomly. This type of noise is known as "salt and pepper".

Most of the times we will want to remove or reduce the noise of images instead of adding it in, by using several algorithms in scikit-image. The higher the resolution of the image, the longer it may take to eliminate the noise.

Some types of denoising algorithms are:

- Total Variation filter - this filter tries to minimize the total variation of the image. It tends to produce “cartoon-like” images, that is, piecewise-constant images. We see that we obtain a denoised image with the edges preserved but a little bit smooth or blurry.
- Bilateral filtering - smooths images while preserving edges. It replaces the intensity of each pixel with a weighted average of intensity values from nearby pixels. The resulting image is less smooth than the one from the total variation filter. And preserves the edges a lot more.
- Wavelet denoising filter
- Non-local means denoising

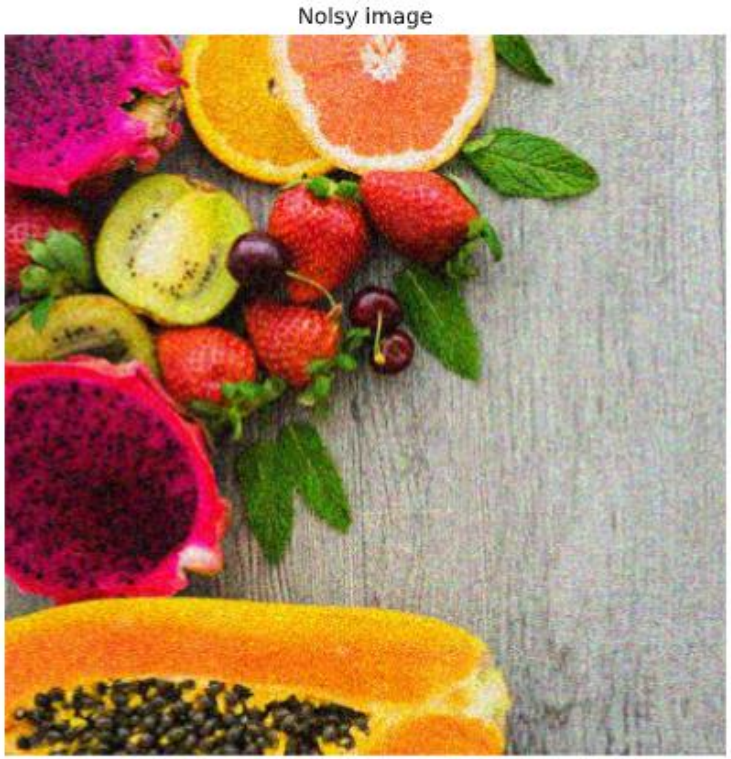

```
from skimage.restoration import denoise_tv_chambolle

# Apply total variation filter denoising
denoised_image = denoise_tv_chambolle(noisy_image,
                                      weight=0.1,
                                      multichannel=True)

# Show denoised image
show_image(noisy_image, 'Noisy image')
show_image(denoised_image, 'Denoised image')
```

```
from skimage.restoration import denoise_bilateral

# Apply bilateral filter denoising
denoised_image = denoise_bilateral(noisy_image, multichannel=True)

# Show original and resulting images
show_image(noisy_image, 'Noisy image')
show_image(denoised_image, 'Denoised image')
```

Let's make some noise!

In this exercise, we'll practice adding noise to a fruit image.

Image preloaded as `fruit_image`.



```
# Import the module and function
from skimage.util import random_noise

# Add noise to the image
noisy_image = random_noise(fruit_image)

# Show original and resulting image
show_image(fruit_image, 'Original')
show_image(noisy_image, 'Noisy image')
```

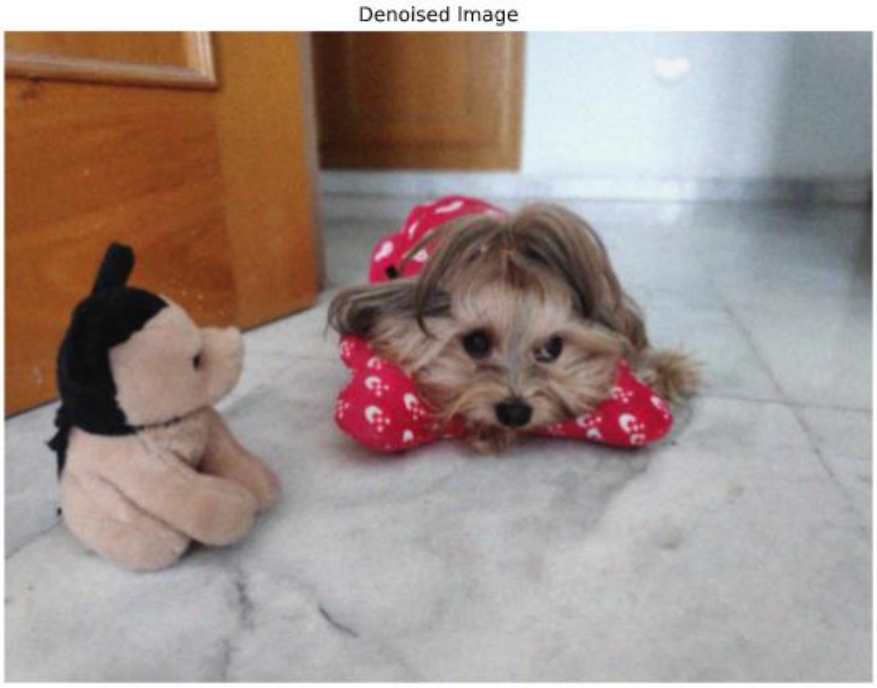
Now you can add noise to any image you work with.
You can always read more about the functions in the scikit-image documentation.

Reducing noise

We have a noisy image that we want to improve by removing the noise in it.

Preloaded as `noisy_image`.

Use total variation filter denoising to accomplish this.

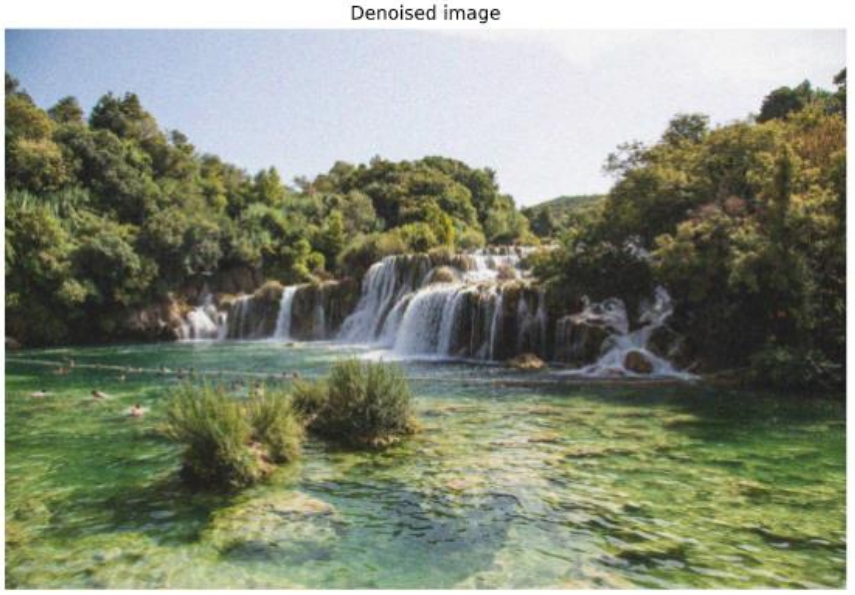


```
# Import the module and function
from skimage.restoration import denoise_tv_chambolle

# Apply total variation filter denoising
denoised_image = denoise_tv_chambolle(noisy_image,
                                      multichannel=True)

# Show the noisy and denoised images
show_image(noisy_image, 'Noisy')
show_image(denoised_image, 'Denoised image')
```

You fixed the image by applying the TV denoising function with the parameter default values. Feel free to read more about them in the scikit-image documentation.

Reducing noise while preserving edges

In this exercise, you will reduce the noise in this landscape picture.



Preloaded as `landscape_image`.

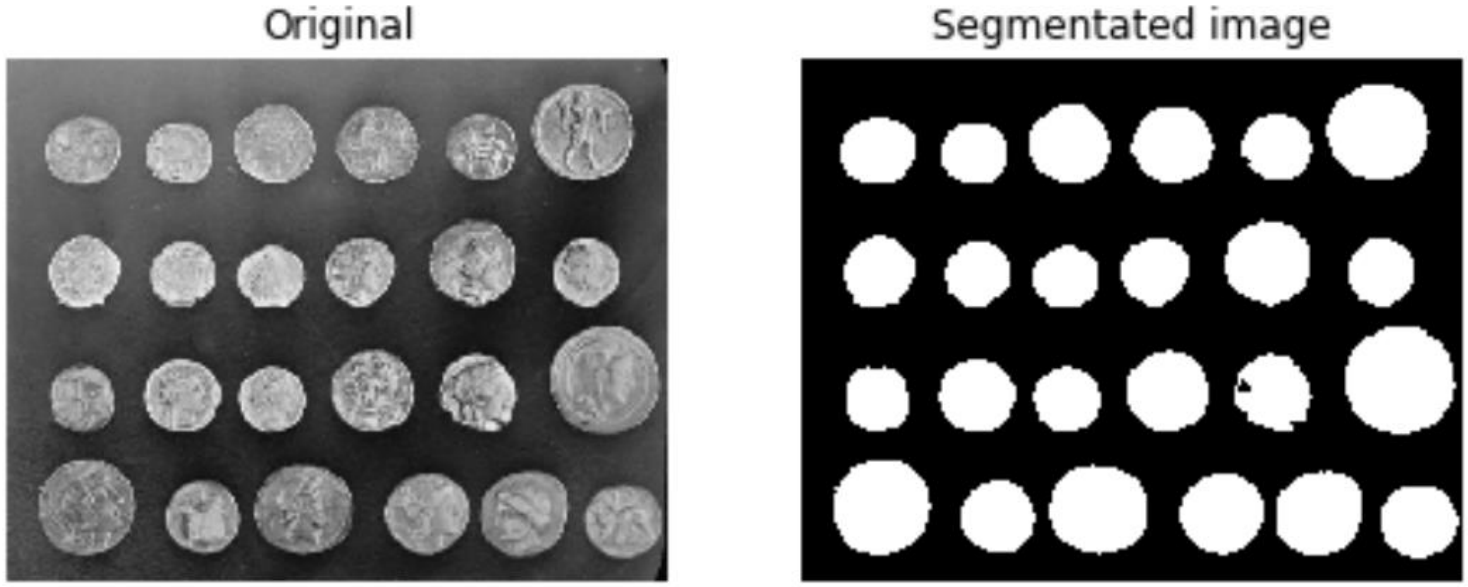
Since we prefer to preserve the edges in the image, we'll use the bilateral denoising filter.

```
# Import bilateral denoising function
from skimage.restoration import denoise_bilateral

# Apply bilateral filter denoising
denoised_image = denoise_bilateral(landscape_image,
                                   multichannel=True)

# Show original and resulting images
show_image(landscape_image, 'Noisy image')
show_image(denoised_image, 'Denoised image')
```

You denoised the image without losing sharpness.
In this case denoise_bilateral() worked well with the default optional parameters.

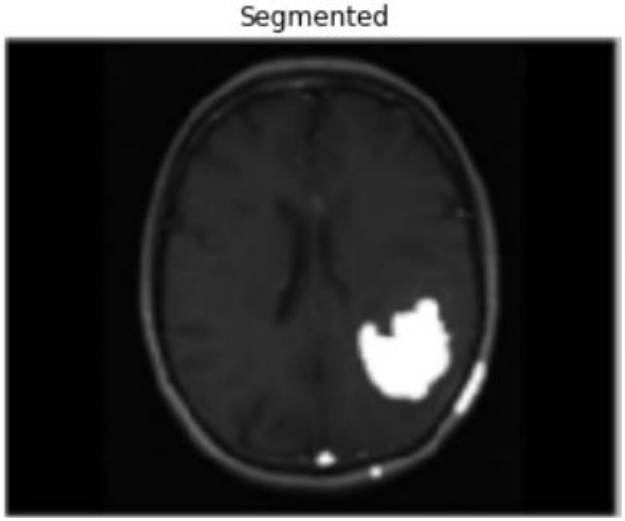
Superpixels & segmentation

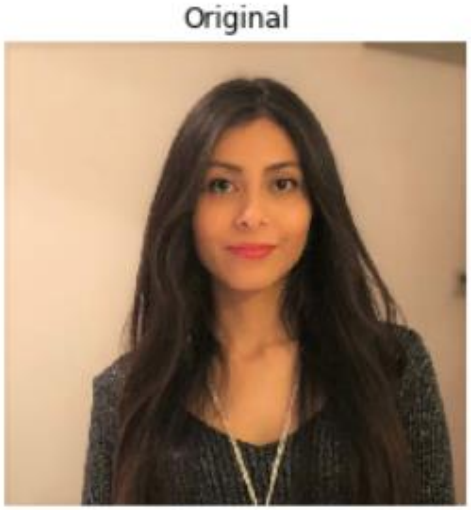
The goal is to partition images into regions, or segments, to simplify and/or change the representation into something more meaningful and easier to analyze.



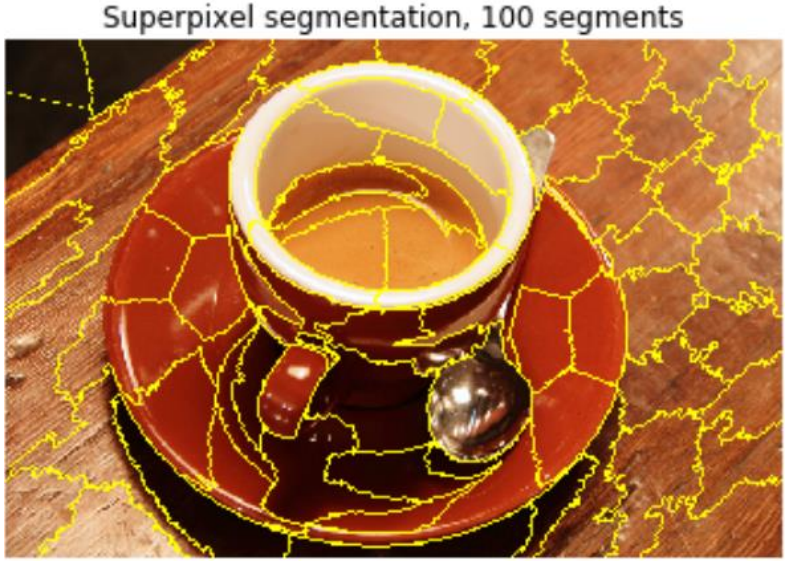
For example, before a tumor is analyzed in a computed tomography, it has to be detected and somehow isolated from the rest of the image. Or before recognizing a face, it has to also be picked out from its

background. Previously we learned about Thresholding, which is the simplest method of segmentation. Separating foreground from background. Now we'll learn about separating more than that.





A single pixel, standing alone by itself, is not a natural representation. We can explore more logical meanings in an image that's formed by bigger regions or grouped pixels. These are known as superpixels. A **superpixel** is a group of connected pixels with similar colors or gray levels. These carry more meaning than their simple pixel grid counterparts.



Superpixel segmentation is dividing an image into superpixels. It has been applied to many computer vision tasks, like visual tracking and image classification. Some advantages for using them are

- can compute features on more meaningful regions
- can reduce an image from thousands of pixels down to some regions for subsequent algorithms, so you have computational efficiency.

Two types of segmentation are

- Supervised - some prior knowledge is used to guide the algorithm. Like the kind of thresholding in which we specify the threshold value ourselves.
- Unsupervised - no prior knowledge is required. These algorithms try to subdivide images into meaningful regions automatically. The user may still be able to tweak certain settings to obtain the desired output. Like the otsu thresholding we used in first chapter.

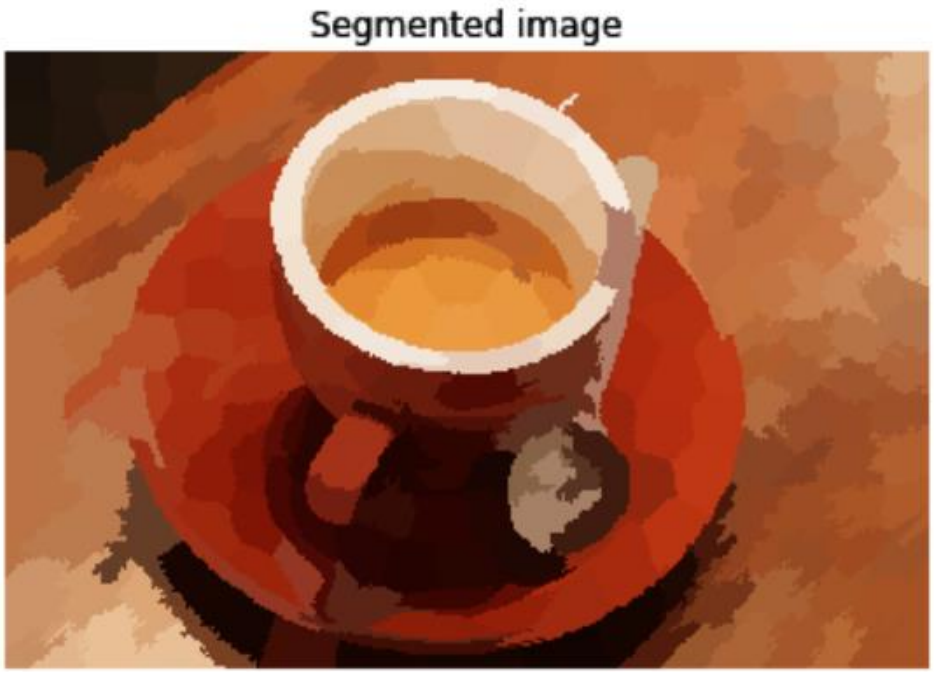
Unsupervised segmentation (SLIC)

Simple Linear Iterative Clustering (SLIC) segments the image using a machine learning algorithm called K-Means clustering. It takes in all the pixel values of the image and tries to separate them into a predefined number of sub-regions.

```
# Import the modules
from skimage.segmentation import slic
from skimage.color import label2rgb

# Obtain the segments
segments = slic(image)

# Put segments on top of original image to compare
segmented_image = label2rgb(segments, image, kind='avg')

show_image(image)
show_image(segmented_image, "Segmented image")
```

You can see how local regions with similar color and texture distributions are part of the same superpixel group. If we want more segments, let's say 300, we can specify this with an optional parameter, n_segments. Its default value is 100 segments.

```
# Import the modules
from skimage.segmentation import slic
from skimage.color import label2rgb

# Obtain the segmentation with 300 regions
segments = slic(image, n_segments= 300)

# Put segments on top of original image to compare
segmented_image = label2rgb(segments, image, kind='avg')

show_image(segmented_image)
```



Number of pixels

Let's calculate the total number of pixels in this image.

Image preloaded as `face_image`

The total amount of pixel is its resolution. Given by Height×Width.

Use `.shape` from NumPy which is preloaded as `np`, in the console to check the width and height of the image.

- `face_image` is 191 * 191 = 36,481 pixels
- `face_image` is 265 * 191 = 50,615 pixels
- `face_image` is 1265 * 1191 = 1,506,615 pixels
- `face_image` is 2265 * 2191 = 4,962,615 pixels

```
In [1]: face_image.shape
Out[1]: (265, 191, 3)
```

The image is 50,615 pixels in total.

Superpixel segmentation

In this exercise, you will apply unsupervised segmentation to the same image, before it's passed to a face detection machine learning model. So you will reduce this image from 265×191=50,615 pixels down to 400 regions. The `show_image()` function has been preloaded for you as well.

Already preloaded as `face_image`.


```
# Import the slic function from segmentation module
from skimage.segmentation import slic

# Import the label2rgb function from color module
from skimage.color import label2rgb

# Obtain the segmentation with 400 regions
segments = slic(face_image, n_segments= 400)

# Put segments on top of original image to compare
segmented_image = label2rgb(segments, face_image, kind='avg')

# Show the segmented image
show_image(segmented_image, "Segmented image, 400 superpixels")
```

You reduced the image from 50,615 pixels to 400 regions! Much more computationally efficient for, for example, face detection machine learning models.

Finding contours

A contour is a closed shape of points or line segments, representing the boundaries of these objects. Once we find the contours, we can do things like identifying the total points in domino tokens, exactly what we do in this example, where we count a total of 29! So we can measure size, classify shapes or determining the number of objects in an image.

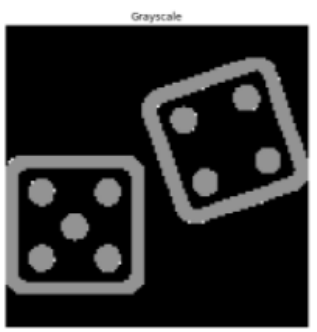
Total points in domino tokens: 29.

The input to a contour-finding function should be a binary image, which we can produce by first applying thresholding. In such binary image, the objects we wish to detect should be white, while the background remains black.

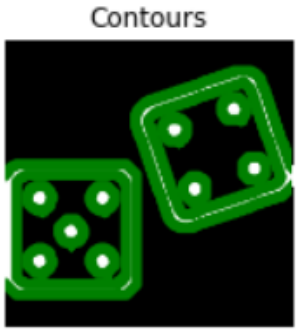
We can obtain a binary image applying **thresholding** or using **edge detection**

```
# Make the image grayscale
image = color.rgb2gray(image)
```

```
# Obtain the thresh value
thresh = threshold_otsu(image)

# Apply thresholding
thresholded_image = image > thresh
```

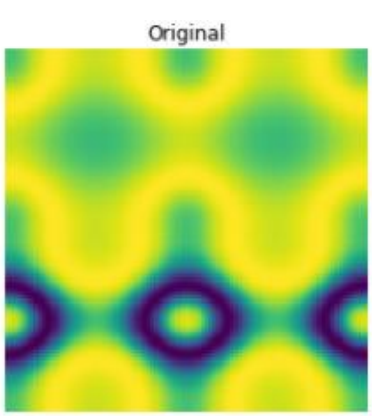


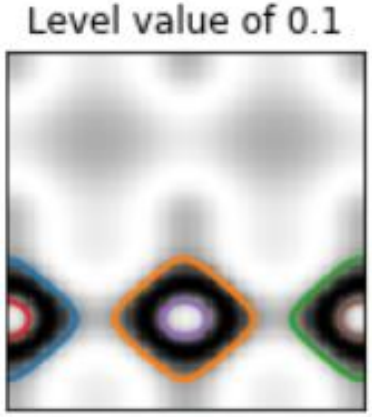

```
# Import the measure module
from skimage import measure

# Find contours at a constant value of 0.8
contours = measure.find_contours(thresholded_image, 0.8)
```

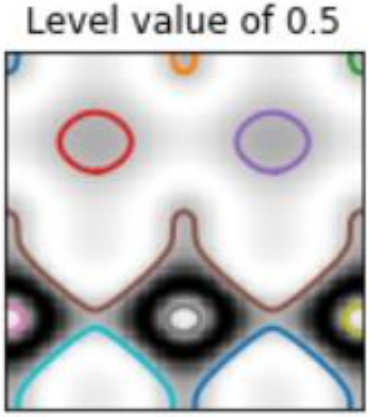


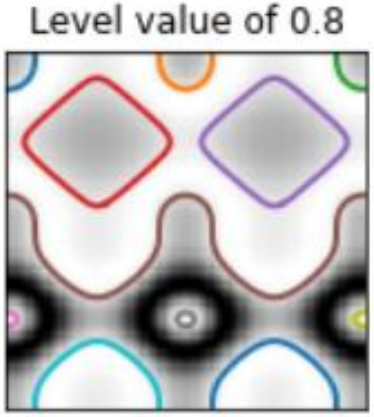
Constant level value

The level value varies between 0 and 1, the closer to 1 the more sensitive the method is to detecting contours, so more complex contours will be detected. We have to find the value that best detects the contours we care for.









```python
from skimage import measure
from skimage.filters import threshold_otsu

# Make the image grayscale
image = color.rgb2gray(image)
# Obtain the optimal thresh value of the image
thresh = threshold_otsu(image)

# Apply thresholding and obtain binary image
thresholded_image = image > thresh

# Find contours at a constant value of 0.8
contours = measure.find_contours(thresholded_image, 0.8)
```

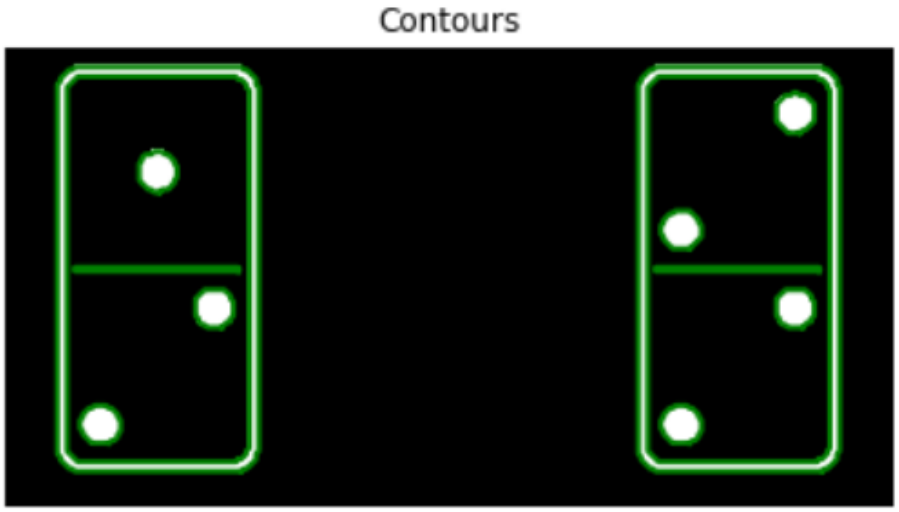
```python
for contour in contours:
    print(contour.shape)
```

```
(433, 2)
(433, 2)
(401, 2)
(401, 2)
(123, 2)
(123, 2)
(59, 2)
(59, 2)
(59, 2)
(57, 2)
(57, 2)
(59, 2)
```

A contour's shape

After executing these steps we obtain a list of contours. Each contour is an ndarray of shape (n, 2), consisting of n row and column coordinates along the contour. In this way, a contour is like an outline formed by multiple points joined together. The bigger the contour, the more points joined together and the wider the perimeter formed. Here we can see the shapes of the contours found in the domino's tokens image.

```
(433, 2)
(433, 2) --> Outer border
(401, 2)
(401, 2) --> Inner border
(123, 2)
(123, 2) --> Divisory line of tokens
(59, 2)
(59, 2)
(59, 2)
(57, 2)
(57, 2)
(59, 2)
(59, 2) --> Dots
```



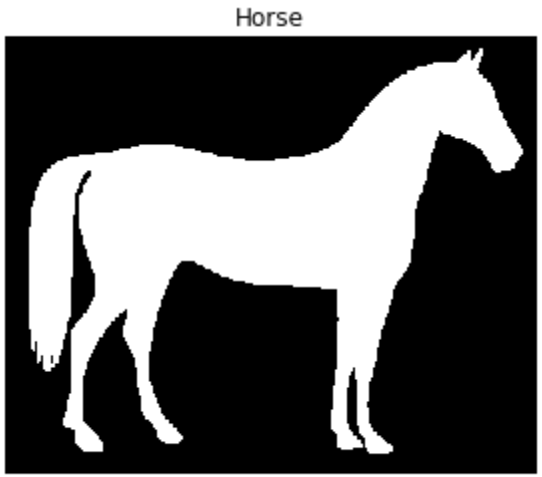
Contouring shapes

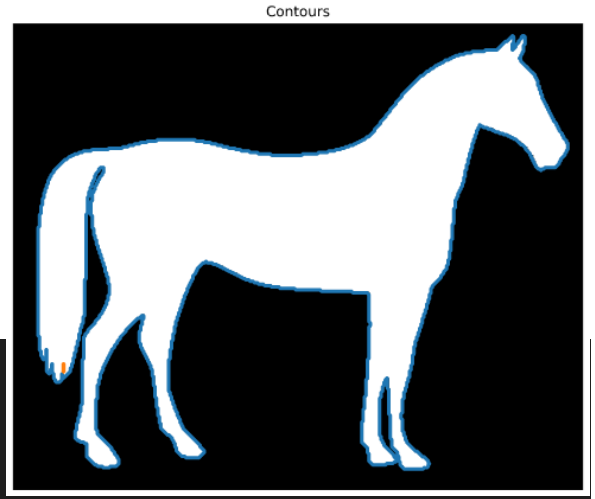
In this exercise we'll find the contour of a horse.

For that we will make use of a **binarized** image provided by scikit-image in its `data` module. Binarized images are easier to process when finding contours with this algorithm. Remember that contour finding only supports 2D image arrays.

Once the contour is detected, we will display it together with the original image. That way we can check if our analysis was correct!

`show_image_contour(image, contours)` is a preloaded function that displays the image with all contours found using Matplotlib. Remember you can use the `find_contours()` function from the measure module, by passing the thresholded image and a constant value.






```
# Import the modules
from skimage import measure, data

# Obtain the horse image
horse_image = data.horse()

# Find the contours with a constant level value of 0.8
contours = measure.find_contours(horse_image, 0.8)
```

```
# Shows the image with contours found
show_image_contour(horse_image, contours)
```

You were able to find the horse contours! In the next exercise you will do some image preparation first and binarize the image yourself before finding the contours.

Find contours of an image that is not binary

Let's work a bit more on how to prepare an image to be able to find its contours and extract information from it.

We'll process an image of two purple dice loaded as `image_dice` and determine what number was rolled for each dice.

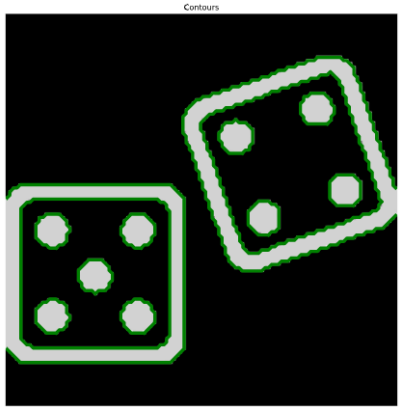
In this case, the image is not grayscale or binary yet. This means we need to perform some image pre-processing steps before looking for the contours. First, we'll transform the image to a 2D array grayscale image and next apply thresholding. Finally, the contours are displayed together with the original image.

`color`, `measure` and `filters` modules are already imported so you can use the functions to find contours and apply thresholding.

We also import the `io` module to load the `image_dice` from local memory, using `imread`. **Read more here.**

```
# Make the image grayscale
image_dice = color.rgb2gray(image_dice)

# Obtain the optimal thresh value
thresh = filters.threshold_otsu(image_dice)

# Apply thresholding
binary = image_dice > thresh
```

```
# Find contours at a constant value of 0.8
contours = measure.find_contours(binary, 0.8)

# Show the image
show_image_contour(image_dice, contours)
```



You made the image a 2D array by slicing, applied thresholding and succesfully found the contour. Now you can apply it to any image you work on in the future.

Count the dots in a dice's image

Now we have found the contours, we can extract information from it.

In the previous exercise, we prepared a purple dices image to find its contours:

This time we'll determine what number was rolled for the dice, by counting the dots in the image.

The contours found in the previous exercise are preloaded as `contours`.

Create a list with all contour's shapes as `shape_contours`. You can see all the contours shapes by calling `shape_contours` in the console, once you have created it.

Check that most of the contours aren't bigger in size than 50. If you count them, they are the exact number of dots in the image.

`show_image_contour(image, contours)` is a preloaded function that displays the image with all contours found using Matplotlib.

```
# Create list with the shape of each contour
shape_contours = [cnt.shape[0] for cnt in contours]

# Set 50 as the maximum size of the dots shape
max_dots_shape = 50

# Count dots in contours excluding bigger than dots size
dots_contours = [cnt for cnt in contours if np.shape(cnt)[0] < max_dots_shape]

# Shows all contours found
show_image_contour(binary, contours)

# Print the dice's number
print("Dice's dots number: {}. ".format(len(dots_contours)))
```

```
<script.py> output:
    Dice's dots number: 9.
```

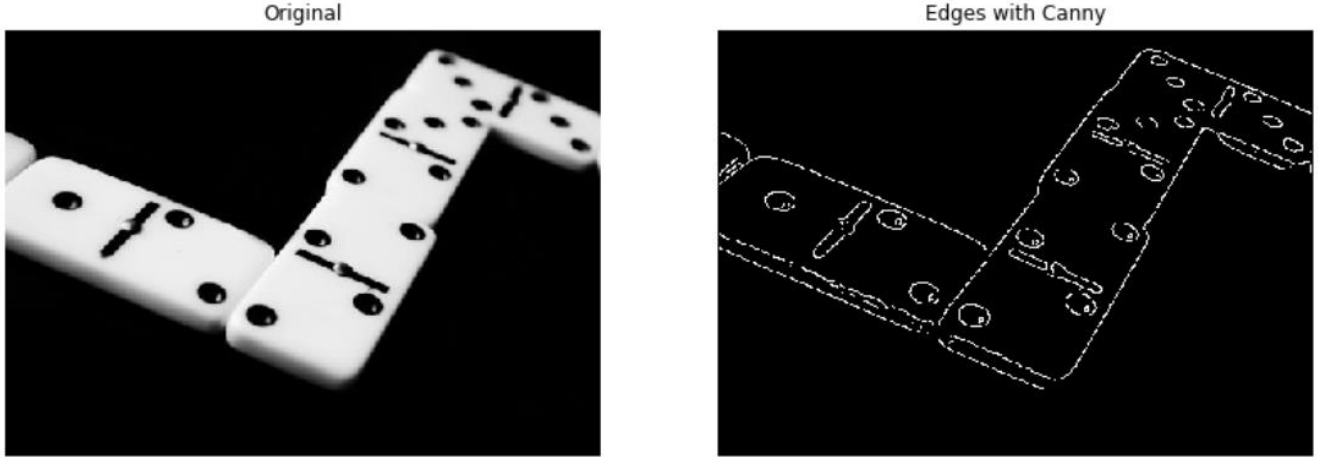
You calculated the dice's number in the image by classifing its contours.

Chapter 4. Advanced Operations, Detecting Faces and Features

After completing this chapter, you will have a deeper knowledge of image processing as you will be able to detect edges, corners, and even faces! You will learn how to detect not just front faces but also face profiles, cat, or dogs. You will apply your skills to more complex real-world applications. Learn to master several widely used image processing techniques with very few lines of code!

Finding the edges with Canny

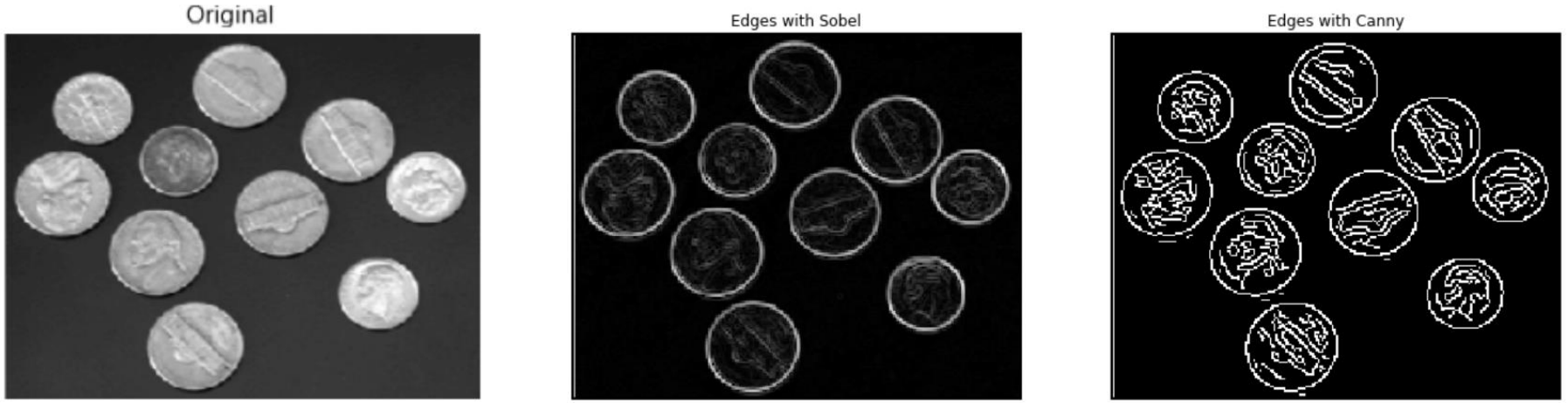
Edge detection is extensively used when we want to divide the image into areas corresponding to different objects. Most of the shape information of an image is enclosed in edges. In this image we see how the edges hold information about the domino's tokens.



Representing an image by its edges has the advantage that the amount of data is reduced significantly while retaining most of the image information, like the shapes.

the Canny edge detection. This is widely considered to be the standard edge detection method in image processing. And produces higher accuracy detecting edges and less execution time compared with Sobel algorithm.

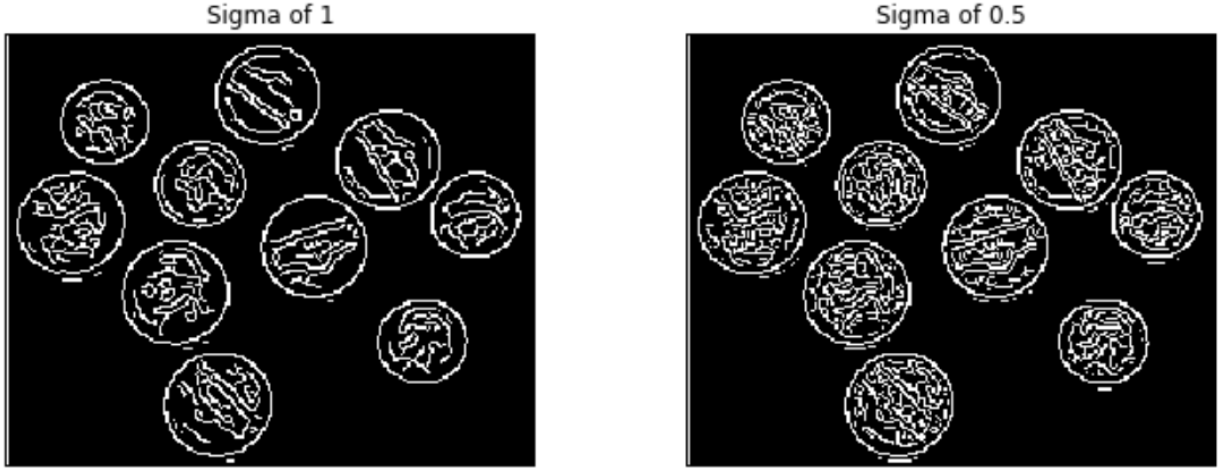
```python
from skimage.feature import canny

# Convert image to grayscale
coins = color.rgb2gray(coins)

# Apply Canny detector
canny_edges = canny(coins)

# Show resulted image with edges
show_image(canny_edges, "Edges with Canny")
```



We see how the edges are highlighted with thick white lines and that some details are more pronounced than the rest of the image. We can also spot the boundaries and shapes of coins; by knowing that for each closed circle or ellipse, there's a coin.

```python
# Apply Canny detector with a sigma of 0.5
canny_edges_0_5 = canny(coins, sigma=0.5)

# Show resulted images with edges
show_image(canny_edges, "Sigma of 1")
show_image(canny_edges_0_5, "Sigma of 0.5")
```

Apply a gaussian filter to remove noise in the image. Set the intensity of this Gaussian filter to be applied in the image by using the sigma attribute. The lower the value of this sigma, the less of gaussian filter effect is applied on the image, so it will spot more edges. On the other hand, if you set a higher value, more noise will be removed and the result is going to be a less edgy image. The default value of this parameter is 1. In this example we set it to 0.5, let's see the effect in the image.



The resulting image has a lot more edges than the previous one and this is because noise was removed before continuing with the rest of the steps in the algorithm.

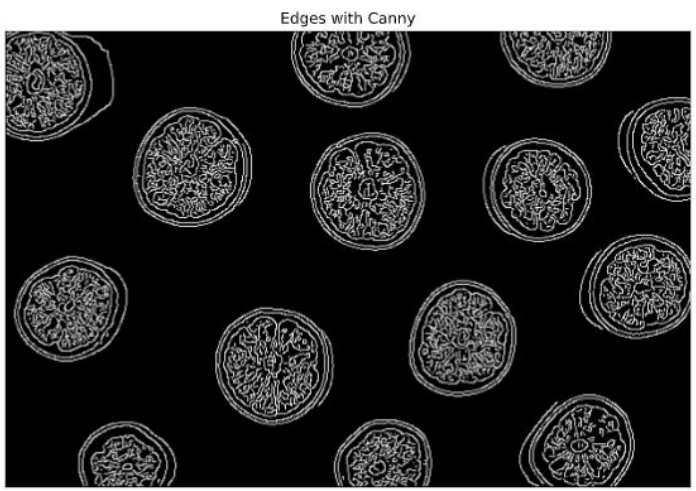
Edges

In this exercise you will identify the shapes in a grapefruit image by detecting the edges, using the Canny algorithm.

Image preloaded as `grapefruit`.

The `color` module has already been preloaded for you.

```
# Import the canny edge detector
from skimage.feature import canny

# Convert image to grayscale
grapefruit = color.rgb2gray(grapefruit)

# Apply canny edge detector
canny_edges = canny(grapefruit)

# Show resulting image
show_image(canny_edges, "Edges with Canny")
```



You can see the shapes and details of the grapefruits of the original image being highlighted.

Less edgy

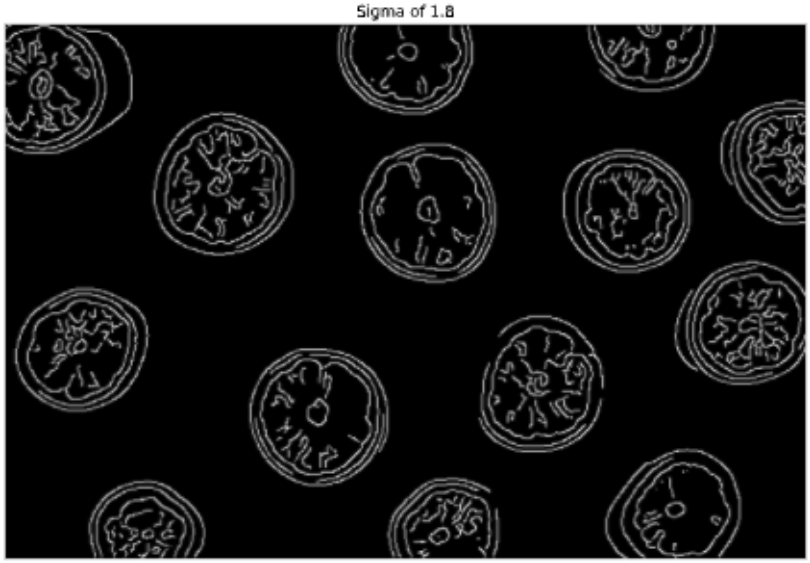
Let's now try to spot just the outer shape of the grapefruits, the circles. You can do this by applying a more intense Gaussian filter to first make the image smoother. This can be achieved by specifying a bigger sigma in the canny function.

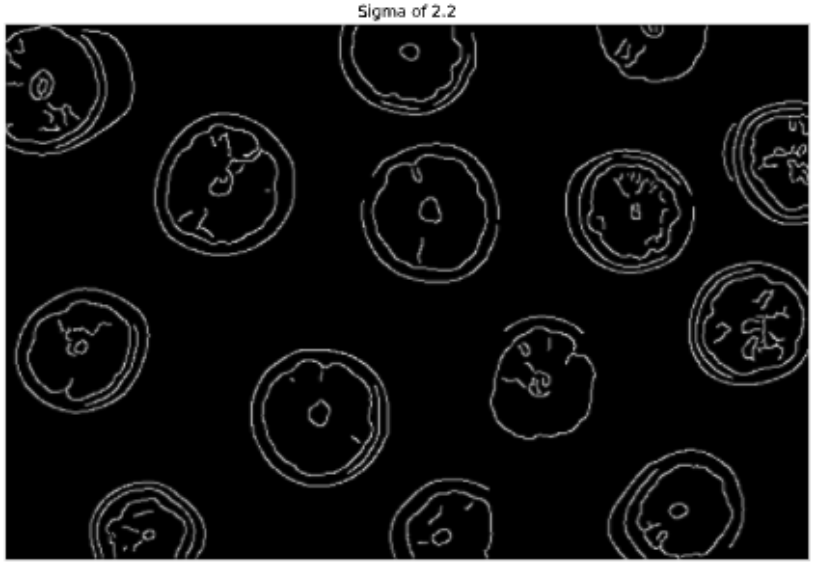
In this exercise, you'll experiment with sigma values of the `canny()` function.

Image preloaded as `grapefruit`.

The `show_image` has already been preloaded.

```
# Apply canny edge detector with a sigma of 1.8
edges_1_8 = canny(grapefruit, sigma=1.8)

# Apply canny edge detector with a sigma of 2.2
edges_2_2 = canny(grapefruit, sigma=2.2)

# Show resulting images
show_image(edges_1_8, "Sigma of 1.8")
show_image(edges_2_2, "Sigma of 2.2")
```



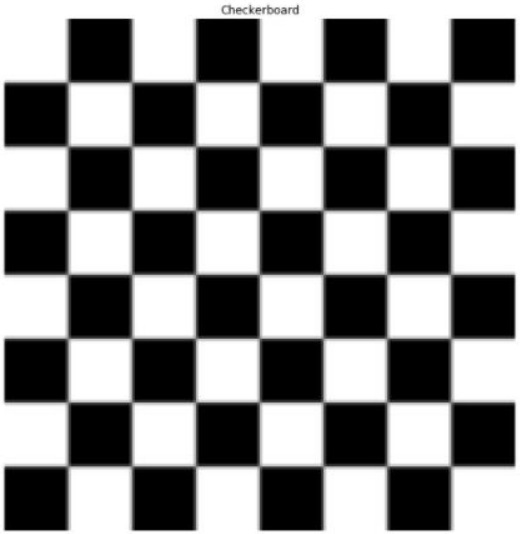


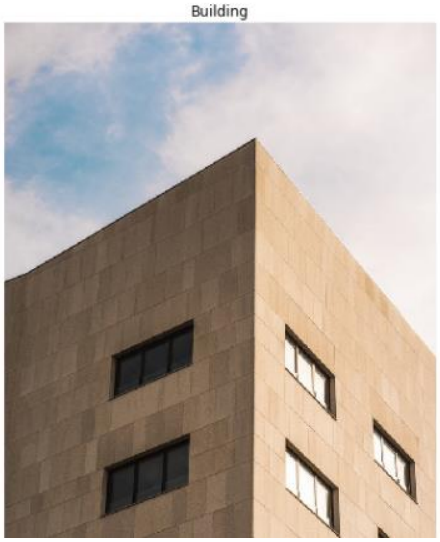
The bigger the sigma value, the less edges are detected because of the gaussian filter pre applied.

Right around the corner

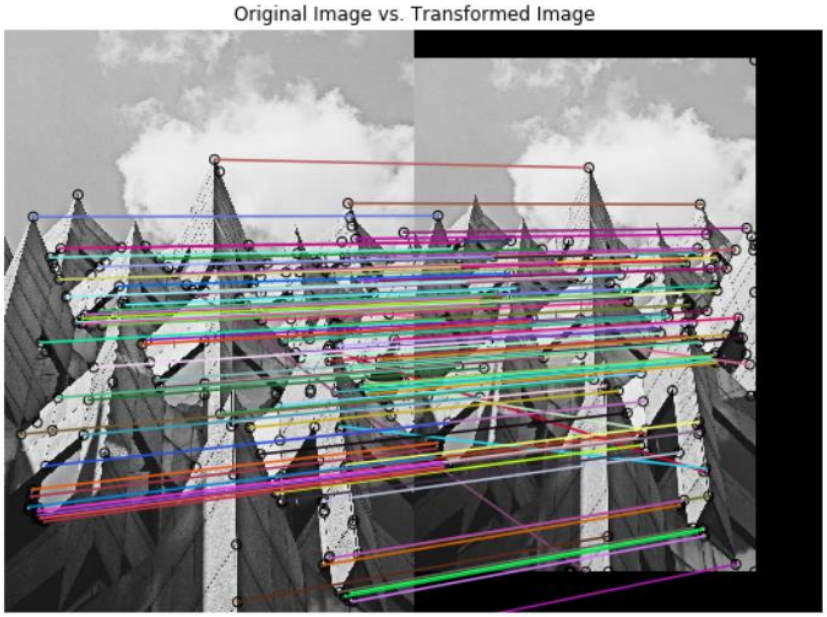
Corner detection is an approach used to extract certain types of features and infer the contents of an image. It's frequently used in motion detection, image registration, video tracking, panorama stitching, 3D modelling, and object recognition.
Features are the points of interest which provide rich image content information. Points of interest are points in the image which are invariant to rotation, translation, intensity, and scale changes. (Basically, robust and

reliable). There are different interest points such as corners and edges. So corner detection is basically detecting (one type of) interest points in an image.
A corner can be defined as the intersection of two edges. Intuitively, it can also be a junction of contours. We can see some obvious corners in this checkerboard image. Or in this building image on the right.

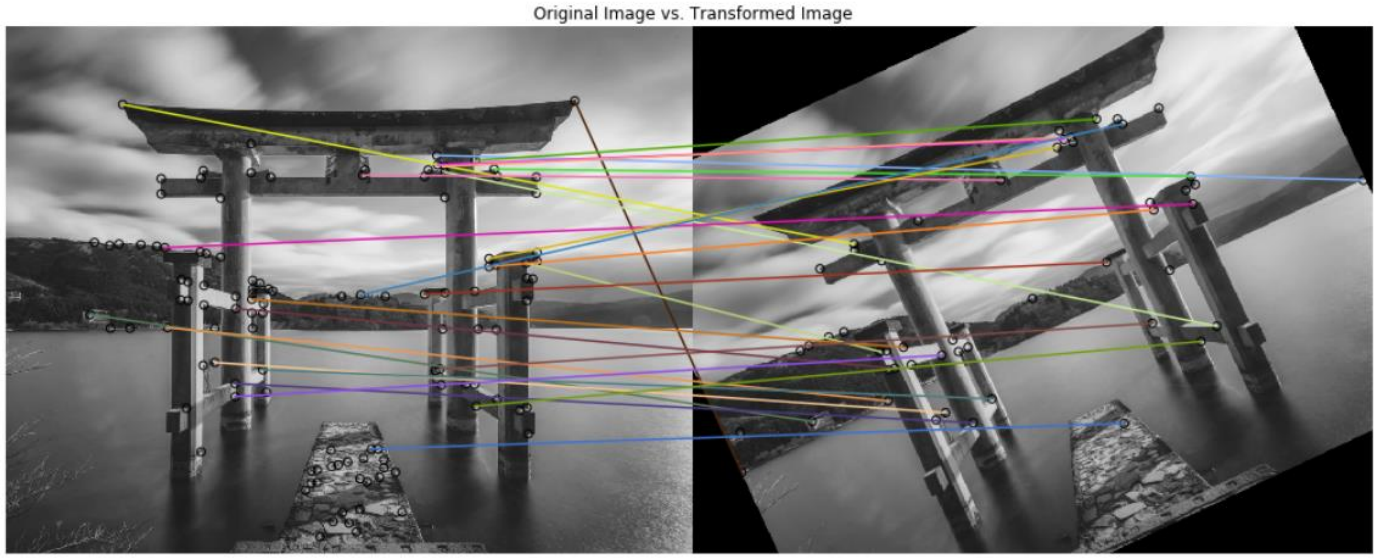




So by detecting corners as interest points, we can match objects from different perspectives. Like in this image, where we detect the corners of the original image on the left and then match them in a downscaled image in the right.

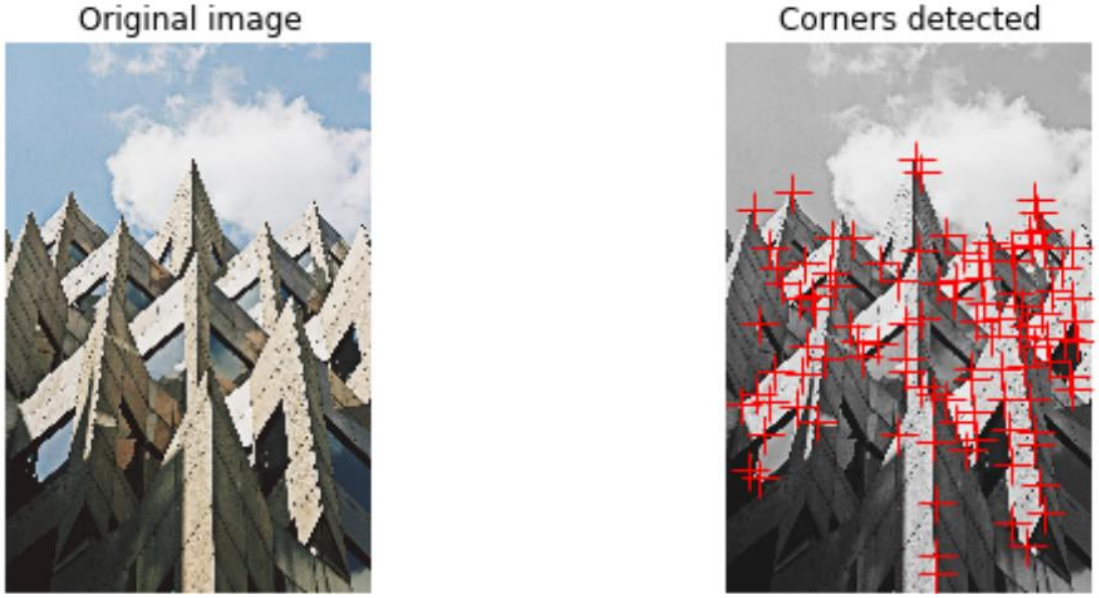


Here is another example of corner matching, this time, in a rotated image. We see how the relevant points. are still being matched.



Harris corner detector

Harris Corner Detector is a corner detection operator that is widely used in computer vision algorithms. Here, we see an original image of a building, and on the right we see the corners detected by the Harris algorithm, marked in red.



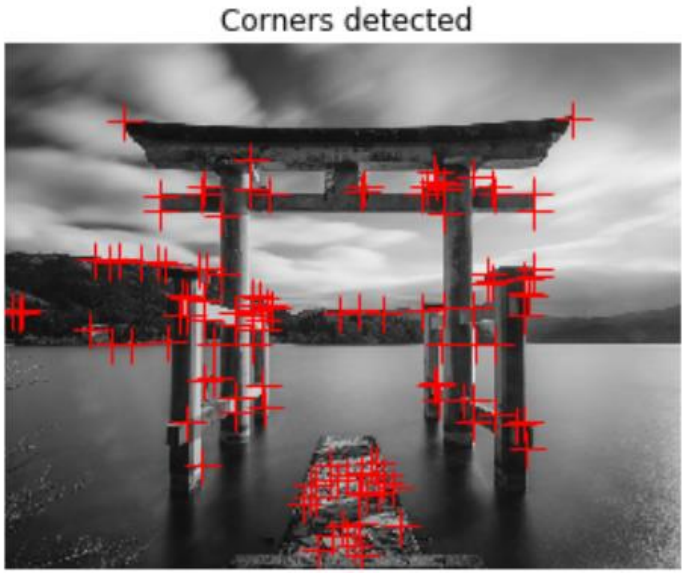
```python
from skimage.feature import corner_harris

# Convert image to grayscale
image = rgb2gray(image)

# Apply the Harris corner detector on the image
measure_image = corner_harris(image)

# Show the Harris response image
show_image(measure_image)
```

Harris response image

```python
# Finds the coordinates of the corners
coords = corner_peaks(corner_harris(image), min_distance=5)

print("A total of", len(coords), "corners were detected.")
```

```
A total of 122 corners were found from measure response image.
```

```
# Show image with marks in detected corners
show_image_with_detected_corners(image, coords)
```




```
def show_image_with_corners(image, coords, title="Corners detected"):
    plt.imshow(image, interpolation='nearest', cmap='gray')
    plt.title(title)
    plt.plot(coords[:, 1], coords[:, 0], '+r', markersize=15)
    plt.axis('off')
    plt.show()
```

Perspective

In this exercise, you will detect the corners of a building using the Harris corner detector.

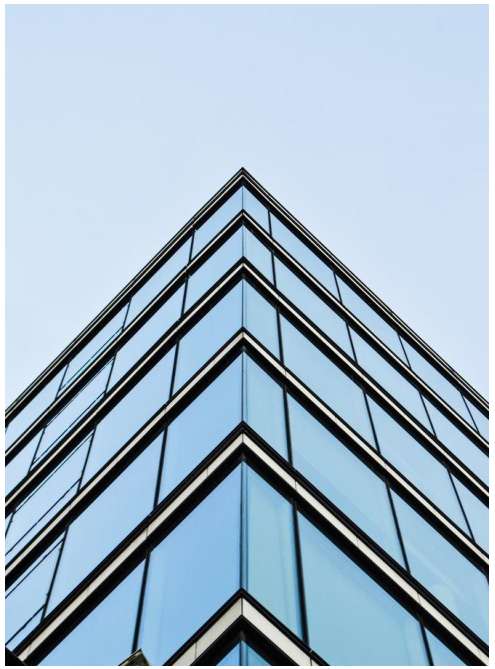


Image preloaded as `building_image`.

The functions `show_image()` and `show_image_with_corners()` have already been preloaded for you. As well as the `color` module for converting images to grayscale.

```
# Import the corner detector related functions and module
from skimage.feature import corner_harris, corner_peaks

# Convert image from RGB-3 to grayscale
building_image_gray = color.rgb2gray(building_image)

# Apply the detector  to measure the possible corners
measure_image = corner_harris(building_image_gray)

# Find the peaks of the corners
coords = corner_peaks(measure_image, min_distance=2)

# Show original and resulting image with corners detected
show_image(building_image, "Original")
show_image_with_corners(building_image, coords)
```

You made the Harris algorithm work fine.

Less corners

In this exercise, you will test what happens when you set the minimum distance between corner peaks to be a higher number. Remember you do this with the `min_distance` attribute parameter of the `corner_peaks()` function.

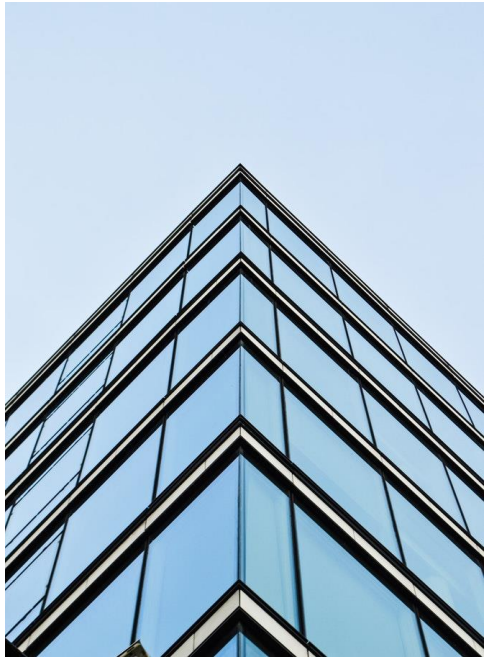
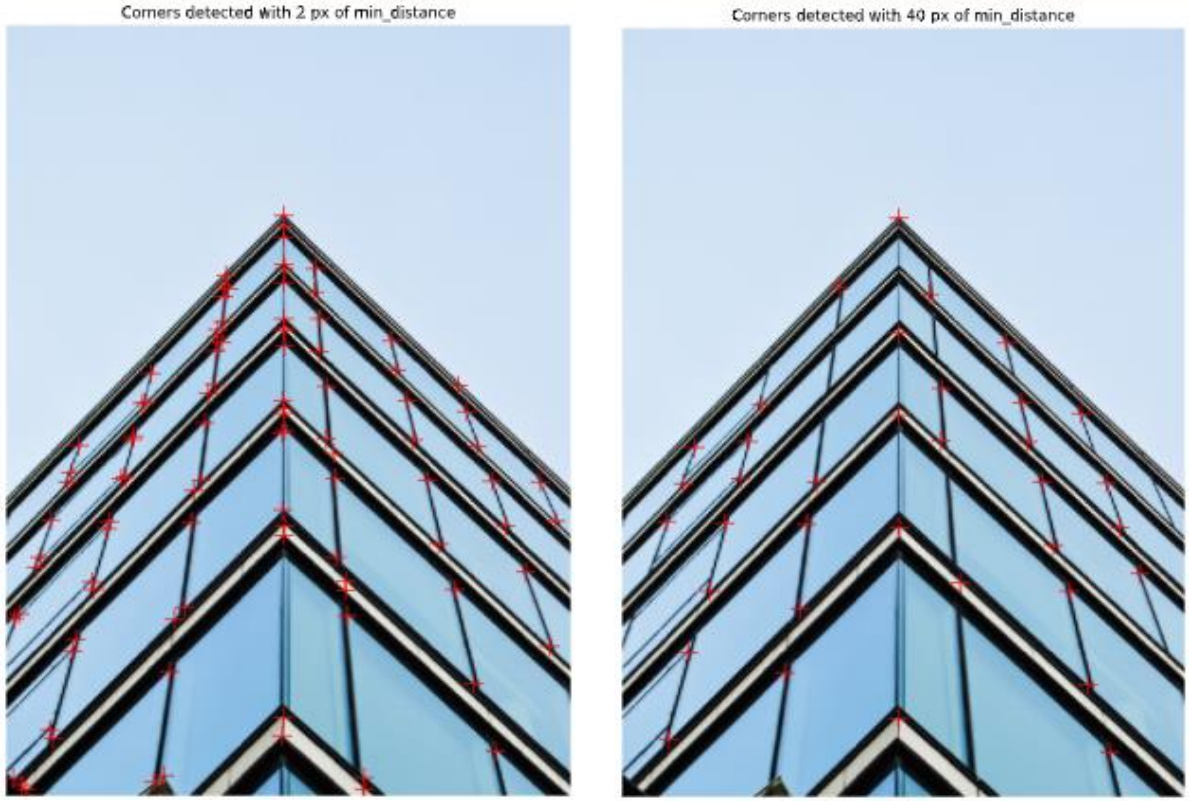


Image preloaded as `building_image`.

The functions `show_image()`, `show_image_with_corners()` and required packages have already been preloaded for you. As well as all the previous code for finding the corners. The Harris measure response image obtained with `corner_harris()` is preloaded as `measure_image`.

```
# Find the peaks with a min distance of 2 pixels
coords_w_min_2 = corner_peaks(measure_image, min_distance=2)
print("With a min_distance set to 2, we detect a total", len(coords_w_min_2), "corners in the image.")

# Find the peaks with a min distance of 40 pixels
coords_w_min_40 = corner_peaks(measure_image, min_distance=40)
print("With a min_distance set to 40, we detect a total", len(coords_w_min_40), "corners in the image.")

# Show original and resulting image with corners detected
show_image_with_corners(building_image, coords_w_min_2, "Corners detected with 2 px of min_distance")
show_image_with_corners(building_image, coords_w_min_40, "Corners detected with 40 px of min_distance")
```

```
<script.py> output:
    With a min_distance set to 2, we detect a total 98 corners in the image.
    With a min_distance set to 40, we detect a total 36 corners in the image.
```



With a 40-pixel distance between the corners there are a lot less corners than with 2 pixels.

Face detection

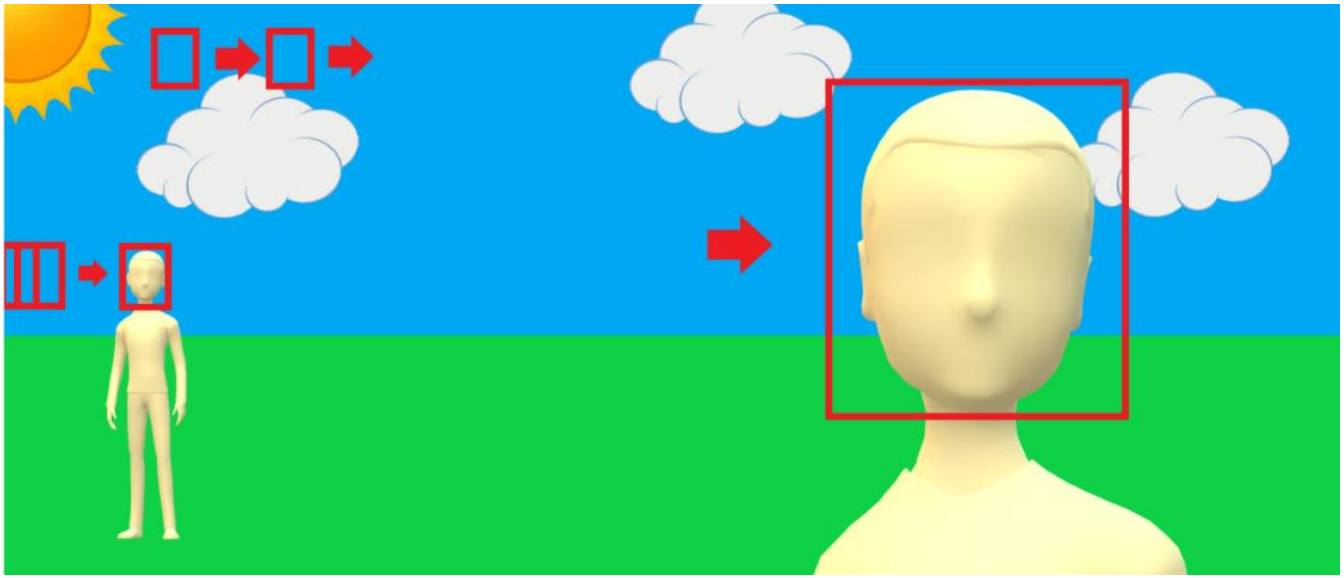
Several social networks platforms and smart-phones are using face detection to know if there is someone in a picture and if so, apply filters, add focus in the face area, or recommend you to tag friends. You can even automatically blur faces for privacy protection.

Face detection can be useful in other cases as well. Human faces are able to convey many different emotions such as happiness, sadness and many others. That's why face detection is the key first step before recognizing emotions.

With scikit-image, we can detect faces using a cascade of machine learning classifiers, with just a couple of lines! A cascade of classifiers is like multiple classifiers in one. You can also use it for other things, like cats, objects, or profile faces, from a side view.

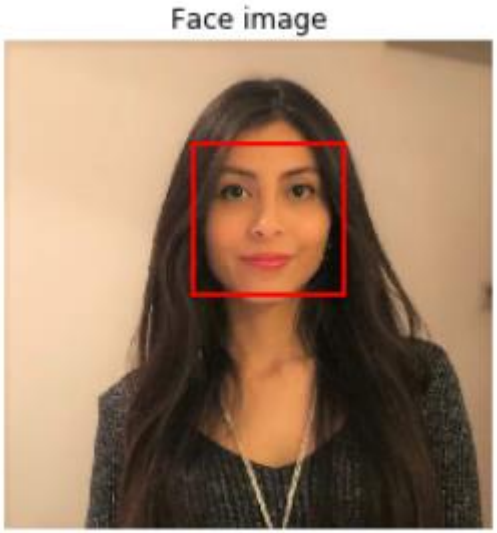
```
# Import the classifier class
from skimage.feature import Cascade

# Load the trained file from the module root.
trained_file = data.lbp_frontal_face_cascade_filename()

# Initialize the detector cascade.
detector = Cascade(trained_file)
```

To apply the detector on images, we need to use the detect_multi_scale method, from the same cascade class. This method searches for the object, in this case a face. It creates a window that will be moving through the image until it finds something similar to a human face.
Searching happens on multiple scales. The window will have a minimum size, to spot the small or far-away faces. And a maximum size to also find the larger faces in the image.



This method takes the input image as the first parameter, a scale factor, by which the searching window is multiplied in each step, a step ratio, in which 1 represents an exhaustive search and usually is slow. By setting this parameter to higher values the results will be worse but the computation will be much faster. Usually, values in the interval 1 to 1.5 give good results. Then the minimum and maximum window size are defined. These specify the interval for the search windows that are applied to the input image to detect the faces.

```
# Apply detector on the image
detected = detector.detect_multi_scale(img=image,
                                       scale_factor=1.2,
                                       step_ratio=1,
                                       min_size=(10, 10),
                                       max_size=(200, 200))

print(detected)
# Show image with detected face marked
show_detected_face(image, detected)
```




```
Detected face: [{'r': 115, 'c': 210, 'width': 167, 'height': 167}]
```

The detector will return the coordinates of the box that contains the face.

With this function I draw a rectangle around detected faces.

```
def show_detected_face(result, detected, title="Face image"):
    plt.imshow(result)
    img_desc = plt.gca()
    plt.set_cmap('gray')
    plt.title(title)
    plt.axis('off')

    for patch in detected:
        img_desc.add_patch(
            patches.Rectangle(
                (patch['c'], patch['r']),
                patch['width'],
                patch['height'],
                fill=False,color='r',linewidth=2)
        )
    plt.show()
```

Is someone there?

In this exercise, you will check whether or not there is a person present in an image taken at night.

Image preloaded as `night_image`.

The `Cascade` of classifiers class from `feature` module has been already imported. The same is true for the `show_detected_face()` function, that is used to display the face marked in the image and crop so it can be shown separately.

```
# Load the trained file from data
trained_file = data.lbp_frontal_face_cascade_filename()

# Initialize the detector cascade
detector = Cascade(trained_file)

# Detect faces with min and max size of searching window
detected = detector.detect_multi_scale(img = night_image,
                                       scale_factor=1.2,
                                       step_ratio=1,
                                       min_size=(10, 10),
                                       max_size=(200, 200))

# Show the detected faces
show_detected_face(night_image, detected)
```

```
<script.py> output:
    {'r': 774, 'c': 131, 'width': 40, 'height': 40}
```

The detector found the face even when it's very small and pixelated. Note though that you would ideally want a well-illuminated image for detecting faces.

Multiple faces

In this exercise, you will detect multiple faces in an image and show them individually. Think of this as a way to create a dataset of your own friends' faces!

Image preloaded as `friends_image`.

The `Cascade` of classifiers class from `feature` module has already been imported, as well as the `show_detected_face()` function which is used to display the face marked in the image and crop it so it can be shown separately.

```
# Load the trained file from data
trained_file = data.lbp_frontal_face_cascade_filename()

# Initialize the detector cascade
detector = Cascade(trained_file)

# Detect faces with scale factor to 1.2 and step ratio to 1
detected = detector.detect_multi_scale(img=friends_image,
                                       scale_factor=1.2,
                                       step_ratio=1,
                                       min_size=(10, 10),
                                       max_size=(200, 200))
# Show the detected faces
show_detected_face(friends_image, detected)
```

```
<script.py> output:
    {'r': 202, 'c': 402, 'width': 45, 'height': 45}
    {'r': 207, 'c': 152, 'width': 47, 'height': 47}
    {'r': 217, 'c': 311, 'width': 39, 'height': 39}
    {'r': 219, 'c': 533, 'width': 48, 'height': 48}
    {'r': 218, 'c': 440, 'width': 52, 'height': 52}
    {'r': 242, 'c': 237, 'width': 41, 'height': 41}
    {'r': 202, 'c': 31, 'width': 36, 'height': 36}
```

The detector gave you a list with all the detected faces.

Segmentation and face detection

Previously, you learned how to make processes more computationally efficient with unsupervised superpixel segmentation. In this exercise, you'll do just that!

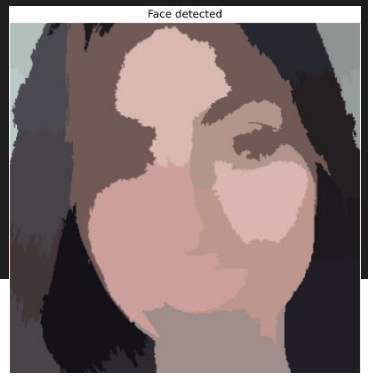
Using the `slic()` function for segmentation, pre-process the image before passing it to the face detector.

Image preloaded as `profile_image`.

The `Cascade` class, the `slic()` function from `segmentation` module, and the `show_detected_face()` function for visualization have already been imported. The detector is already initialized and ready to use as `detector`.

```
# Obtain the segmentation with default 100 regions
segments = slic(profile_image)

# Obtain segmented image using label2rgb
segmented_image = label2rgb(segments, profile_image, kind='avg')

# Detect the faces with multi scale method
detected = detector.detect_multi_scale(img=segmented_image,
                                       scale_factor=1.2,
                                       step_ratio=1,
                                       min_size=(10, 10), max_size=(1000, 1000))

# Show the detected faces
show_detected_face(segmented_image, detected)
```

```
<script.py> output:
    {'r': 110, 'c': 169, 'width': 340, 'height': 340}
```



You applied segementation to the image before passing it to the face detector and it's finding the face even when the image is relatively large.
This time you used 1000 by 1000 pixels as the maximum size of the searching window because the face in this case was indeed rather larger in comparison to the image.

Real-world applications

Some cases where we might need to combine several techniques are, for example

- converting to images to grayscale before detecting edges or corners.
- detecting faces to later on blur them by applying a gaussian filter.
- reducing noise and restoring a damaged image.
- approximation of objects' sizes

Let's look at how we would solve a privacy protection case by detecting faces and then anonymizing them.
We'll first need to detect faces, using the cascade of classifiers detector and then apply a gaussian filter to the cropped faces.
For each detected face, as the variable d, in the detected list, we'll use the coordinates to crop it out of the image, in other words, extract it.

```
# Import Cascade of classifiers and gaussian filter
from skimage.feature import Cascade
from skimage.filters import gaussian
```

```
# Detect the faces
detected = detector.detect_multi_scale(img=image,
                                       scale_factor=1.2, step_ratio=1,
                                       min_size=(50, 50), max_size=(100, 100))
# For each detected face
for d in detected:
    # Obtain the face cropped from detected coordinates
    face = getFace(d)
```

```
def getFace(d):
    ''' Extracts the face rectangle from the image using the
    coordinates of the detected.'''
    # X and Y starting points of the face rectangle
    x, y  = d['r'], d['c']

    # The width and height of the face rectangle
    width, height = d['r'] + d['width'],  d['c'] + d['height']

    # Extract the detected face
    face= image[x:width, y:height]
    return face
```

```
# Detect the faces
detected = detector.detect_multi_scale(img=image,
                                       scale_factor=1.2, step_ratio=1,
                                       min_size=(50, 50), max_size=(100, 100))
# For each detected face
for d in detected:
    # Obtain the face cropped from detected coordinates
    face = getFace(d)

    # Apply gaussian filter to extracted face
    gaussian_face = gaussian(face, multichannel=True, sigma = 10)

    # Merge this blurry face to our final image and show it
    resulting_image = mergeBlurryFace(image, gaussian_face)
```

```python
def mergeBlurryFace(original, gaussian_image):
    # X and Y starting points of the face rectangle
    x, y  = d['r'], d['c']
    # The width and height of the face rectangle
    width, height = d['r'] + d['width'],  d['c'] + d['height']

    original[ x:width, y:height] =  gaussian_image
    return original
```

So it results in an image that no longer contains people's faces in it and in this way, personal data is anonymized.

Blurred faces

The classifier was only trained to detect the front side of faces, not profile faces. So, if someone is turning the head too much to a side, it won't recognize it. If you wnat to do that, you'll need to train the classifier with xml files of profile faces, that you can find available online. Like some provided by the OpenCV image processing library.

Privacy protection

Let's look at a real-world application of what you have learned in the course.

In this exercise, you will detect **human** faces in the image and for the sake of privacy, you will anonymize data by blurring people's faces in the image automatically.

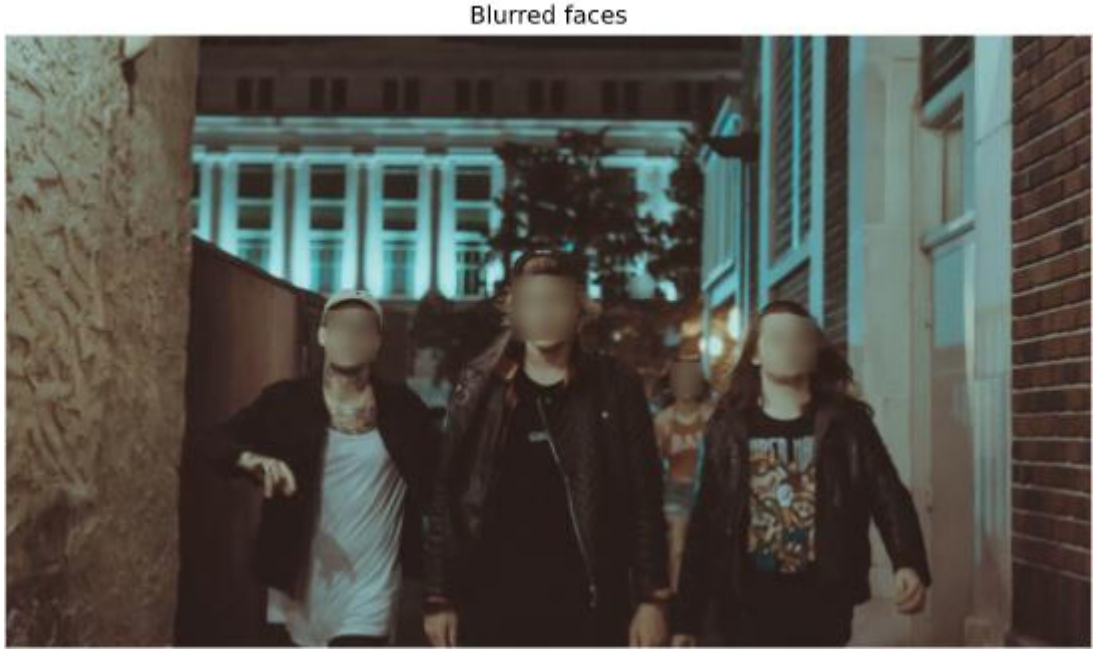
Image preloaded as `group_image`.

You can use the gaussian filter for the blurriness.

The face detector is ready to use as `detector` and all packages needed have been imported.

```
# Detect the faces
```

```
detected = detector.detect_multi_scale(img=group_image,
                                       scale_factor=1.2, step_ratio=1,
                                       min_size=(10, 10), max_size=(100, 100))
# For each detected face
for d in detected:
    # Obtain the face rectangle from detected coordinates
    face = getFaceRectangle(d)

    # Apply gaussian filter to extracted face
    blurred_face = gaussian(face, multichannel=True, sigma = 8)

    # Merge this blurry face to our final image and show it
    resulting_image = mergeBlurryFace(group_image, blurred_face)
show_image(resulting_image, "Blurred faces")
```



You solved this important issue by applying what you have learned in the course.

Help Sally restore her graduation photo

You are going to combine all the knowledge you acquired throughout the course to complete a final challenge: reconstructing a very damaged photo.

Help Sally restore her favorite portrait which was damaged by noise, distortion, and missing information due to a breach in her laptop.

Sally's damaged portrait is already loaded as `damaged_image`.

You will be fixing the problems of this image by:

- Rotating it to be uprightusing `rotate()`
- Applying noise reduction with `denoise_tv_chambolle()`
- Reconstructing the damaged parts with `inpaint_biharmonic()` from the `inpaint` module.

`show_image()` is already preloaded.

```
# Import the necessary modules
from skimage.restoration import denoise_tv_chambolle, inpaint
from skimage.transform import rotate

# Transform the image so it's not rotated
upright_img = rotate(damaged_image, 20)

# Remove noise from the image, using the chambolle method
```

```
upright_img_without_noise = denoise_tv_chambolle(upright_img,weight=0.1, multichannel=True)

# Reconstruct the image missing parts
mask = get_mask(upright_img)
result = inpaint.inpaint_biharmonic(upright_img_without_noise, mask, multichannel=True)

show_image(result)
```

You have learned a lot about image processing methods and algorithms: You performed rotation, removed annoying noise, and fixed the missing pixels of the damaged image.

Course completed!

Recap the topics you have learned:

- Improved contrast
- Restored images
- Applied filters
- Rotated, flipped and resized
- Segmented: supervised and unsupervised
- Applied morphological operators
- Created and reduced noise
- Detected edges, corners and faces
- Combination of the above to solve problems

Next steps:

- Tinting gray scale images
- Matching
- Approximation
- etc

Happy learning!