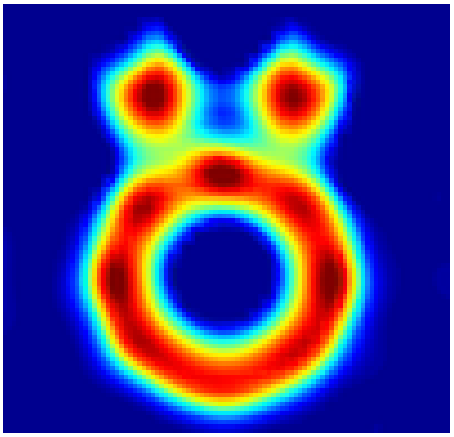
Project 2 - Image Transformation



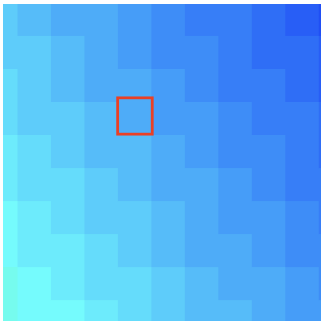
In this project, you are asked to implement functions that will enable you to apply image transformation techniques on a given image.

The objective of this project is to learn about representation of colors, data structures, practice basic mathematical operations on custom data structures, and create fundamental knowledge about basics of the image processing techniques.

Background Information

Representation of Images

In computers, an image is formed with the combination of small blocks called ***pixels***. If you zoom in an image far enough, you will see the pixels on an image. For example the red square shows one of the pixels on the given image.



When we talk about the resolution of an image, we actually talk about how many pixels it has on horizontal and vertical axises. This is also true for imaging or video related equipment such as TVs and cameras. For example an image with **1920×1080** resolution means there are 1920 pixels on the horizontal axis and 1080 pixels on the vertical axis. For a TV, **1920×1080** resolution has a similar meaning: the TV has 1920 pixels horizontally and 1080 pixels vertically to display images.

Just as pixels are used to form an image, each pixel needs to represent **a color** for the image to be meaningful.

Representation of Colors

Colors in computers are represented using the combination of 3 specific colors: ***Red***, ***Green*** and ***Blue***. By defining a minimum and maximum value for each of these colors, we can represent any given color with the combination of these three colors in some amounts. This is the reason we also call these colors **RGB colors**.

For example, for a minimum and maximum values of 0, and 255 for each color, if we set Red value to be 135, Green value to be 104, and Blue value to be 72, we get the following color:

Since a pixel represents the RGB colors, any given image can also be represented using these RGB colors, and the image is actually the combination of these three distinct colors, also called ***channels***: the Red channel, the Green channel and the Blue channel.

For example, for the image given in the header, if we extract each channel; the Red, Green and Blue channels will look like the following:

RGB Colors

As explained before, all colors are represented by defining a minimum and maximum value for each of the RGB colors. Typically these are chosen as 0 and 255, and the order of writing these colors is almost always Red, Green and Blue (hence RGB).

Some RGB color examples:

```
1. Blue  = [   0,   0, 255]
2. Green = [   0, 255,   0]
3. Red   = [ 255,   0,   0]
4. White = [ 255, 255, 255]
5. Black = [   0,   0,   0]
6. Gray  = [  95,  95,  95]
```

HEX Notation

Hex notation is used to represent colors more conveniently, especially on the web. In this notation, each channel is represented with two hex digits in the RGB order to form a 6 hex digit number. For example, for the given brownish color before (`[135, 104, 72]`), the hex representation is given as **876848.** 135 is 0x87, 104 is 0x68 and 72 is 0x48 in hex. This value can also be shown with a preceding hashtag as **#876848.**

Some color examples:

1. `Blue  = 0000FF`
2. `Green = 00FF00`
3. `Red   = FF0000`
4. `White = FFFFFF`
5. `Black = 000000`
6. `Gray  = 5F5F5F`

You can use the following utilities to play with the colors and look at the RGB values of a given color:

- For Windows: https://docs.microsoft.com/en-us/windows/powertoys/color-picker
- For Mac OS: Digital Color Meter.app

Project Design Rules:

- You will be working on implementing functions that will allow you to apply various image transformation techniques on a given image. Since an image is essentially a 2D matrix, you will be working with a list of lists to represent the rows and columns and each item in the list will hold **pixels**. The pixel's value will be represented as a dictionary that holds the red, green and blue channel values.
  - **pixel** structure should be `{'red' : value, 'green': value, 'blue' : value}`
  - Each channel value is **integer**, and can be **between 0 and 255**.
  - This is the **image format** that you will be dealing with.
  - An example `3x2` image is given below:

```
img = [ [ {'red' : 10, 'green' : 10, 'blue': 10}, {'red' : 20,  'green' :  20, 'blue': 20} ],
        [ {'red' : 30, 'green' : 30, 'blue': 50}, {'red' : 142, 'green' :  20, 'blue': 40} ],
        [ {'red' : 10, 'green' : 20, 'blue': 10}, {'red' : 100, 'green' : 255, 'blue':  0} ] ]
>>> print(img[0][1])
{'red': 20, 'green': 20, 'blue': 20}
```

- You will be graded for each of the given functions so make sure to implement each of the functions **with exact names, parameters and default values** -if given- in your python code.
- You should generally **mutate** the given image, unless it is specifically asked otherwise.
- For any of the functions, the final image should be in valid form, meaning all pixel values should be integer, and should be between 0 and 255. (no floats)
- Image size can go as high as `1000x1000` and as low as `1x1`, but this should not limit you or change anything in your implementation.
- No additional modules should be used except random.
- For file read / write operations, you need to convert the pixels to hex notation representation.
  - Each pixel is represented as a 24-bit hexadecimal string showing the red, green, and blue channels in order.
  - Example representations are given below:
    For a pixel with `{'red' : 10, 'green' : 20, 'blue': 30}` values, the hexadecimal representation is **`0A141E`**. Red is `10` and hex representation is **`0A`**,

Green is `20` and hex representation is **`14`**, and blue is `30` and hex representation is **`1E`**. We concatenate these three in **`RGB`** order (**`RRGGBB`**) which gives us **`0A141E`**.

- For example, for the previously given 3x2 image above, the file contents is (if you open with a text editor):

```
0A0A0A,141414
1E1E53,8E1428
0A140A,64FF00
```

  - Note that there is no comma at the end of the lines.
  - Each line is terminated with an LF character. (`'\n'`)
  - Number of columns should be equal on each line.
  - An example file is also given with the project.
- Displaying an image requires additional work, but you can use the supplied **render.html** file to see the saved image txt file. (Open *render.html* on your browser (Chrome, Firefox, Edge, etc.) and choose the txt file to render the image)
  - After implementing the save function, you can save any of your images and render on your browser to see the result.

Functions:

Function Name	**Explanation**
`generate_random(row, column)`	This function will take two parameters called row and column. It will generate an image with random values for each channel. The random values should be integers between 0 and 255. Hint: You can use the **randint** function from the **random** module. ● `row` : integer `100 ≥ row ≥ 1` ● `column` : integer `100 ≥ column ≥ 1` Return the generated image in the given image format.
`is_valid(img)`	This function will take the image and check if each pixel is valid in the image, meaning the values are all integer and between 0 and 255 (both included). Return `True` if the image is valid, `False` otherwise. Note that the return type is bool.
`read_from_file(filename)`	This function will read the given file with the `filename` as input, convert the hex strings to the given image format. ● `filename` : string

	Return the image.
`write_to_file(img, filename)`	This function will take two parameters, `img` as image and `filename` as the filename to be written, then write the given image to the given file using hex strings. It should first convert the image to hex strings and should write that. **Note that the format of the new file should be the same as explained above.** ● `filename` : string Return `None` Warning: Do not write to the same file you read, you will overwrite the file contents.
`clear(img)`	This function will set each channel to 0. Note that the given image is mutated at the end of the operation. Return `None`.
`set_value(img, value, channel='rgb')`	This function will set the specified channel to the given `value`. ● `value` : integer `255 ≥ value ≥ 0` ● `channel` : string `∈ {'r','g','b'}` such that ○ if given as `'r'`, `'g'`, or `'b'`: set only red, green, and blue channels, respectively. ○ if given as `'rg'`, `'rb'`, `'gb'`: set two of the specified channels ○ if given as `'rgb'`: set all channels (default) ○ any other letter should be ignored (we will not test this). Note that the given image is mutated at the end of the operation. Return `None`.
`fix(img)`	This function will take an image, go through pixels and correct the pixel values if any of them is not valid. ● If a channel value of any pixel is above 255, set the channel value to 255.

	• If a channel value of any pixel is below 0, set the channel value to 0. • If a channel value of any pixel is not integer round the pixel value. Note that the given image is mutated at the end of the operation. Return `None`.
`rotate90(img)`	This function will rotate the image to the right `90°`, and return the new image. **Note that the image is not mutated, and a new image is returned.** Return the image as a list of lists.
`rotate180(img)`	This function will rotate the image to the right `180°`, and return the new image. **Note that the image is not mutated, and a new image is returned.** Return the image as a list of lists.
`rotate270(img)`	This function will rotate the image to the right `270°`, and return the new image. **Note that the image is not mutated, and a new image is returned.** Return the image as a list of lists.
`mirror_x(img)`	This function will take the mirror of the image with respect to the first dimension. Note that the given image is mutated at the end of the operation. Return `None`
`mirror_y(img)`	This function will take the mirror of the image with respect to the second dimension, and return the new image. Note that the given image is mutated at the end of the operation. Return `None`
`enhance(img, value, channel='rgb')`	This function will enhance the specified channel ***by multiplying*** that channel with the given value. Inputs:

	• `value` : float `value > 0`
	• `channel` : string ∈ `{'r','g','b'}` such that
	  ○ if given as `'r'`, `'g'`, or `'b'`: enhance only red, green, and blue channels, respectively.
	  ○ if given as `'rg'`, `'rb'`, `'gb'`: enhance two of the specified channels
	  ○ if given as `'rgb'`: enhance all channels (default)
	  ○ any other letter should be ignored (we will not test this).
	
	Note that the given image is mutated at the end of the operation.
	
	Return `None`.

`grayscale(img, mode=1)`	This function will convert the image to a grayscale image. Calculate the pixel value based on the formula given in modes and **set each channel to the same value.** • `mode` indicates the method for the grayscale conversion[1]: `mode=1` : (default) uniformly weighted average $p_{i,j} = (r_{i,j} + g_{i,j} + b_{i,j})/3$ `mode=2` : ITU-R BT 601 Standard $p_{i,j} = 0.299r_{i,j} + 0.587g_{i,j} + 0.114b_{i,j}$ `mode=3` : ITU-R BT.709 Standard $p_{i,j} = 0.2126r_{i,j} + 0.7152g_{i,j} + 0.0722b_{i,j}$ `mode=4` : ITU-R BT.2100 Standard $p_{i,j} = 0.2627r_{i,j} + 0.6780g_{i,j} + 0.0593b_{i,j}$ Note that the given image is mutated at the end of the operation. Return `None` [1]https://en.wikipedia.org/wiki/Grayscale
`get_freq(img, channel='rgb', bin_size=16)`	This function will find the frequency of pixels in the given image. `bin_size` is used to group pixels together. Inputs:

<table>
<tr>
<td></td>
<td>

<code>channel</code> : string ∈ <code>{'r','g','b'}</code> such that

if given as <code>'r'</code>, <code>'g'</code>, or <code>'b'</code>: enhance only red, green, and blue channels, respectively.
if given as <code>'rg'</code>, <code>'rb'</code>, <code>'gb'</code>: enhance two of the specified channels
if given as <code>'rgb'</code>: enhance all channels (default)
any other letter should be ignored (we will not test this).
we will not give any empty string.

<code>bin_size</code> : integer <code>x</code> : { <code>x</code> ∈ 2^k <code>where k ∈ [0:8]}</code>

Return a dictionary that holds at most four key, value pairs.

<code>'bin_size'</code> key should hold the <code>bin_size</code> value.
<code>'red'</code> key should hold the frequency list of pixels if the red channel is selected.
<code>'green'</code> key should hold the frequency list of pixels if the green channel is selected.
<code>'blue'</code> key should hold the frequency list of pixels if the blue channel is selected.

Note that the order of the frequencies in the lists are important.
</td>
</tr>
<tr>
<td><code>scale_down(img, N)</code></td>
<td>
This function will scale down the image by N times and return the new image.
You should divide the image to NxN squares starting from top left and take the average of pixels for each square to create the final pixel value for that square.
For a single channel image let's assume the image is:
<pre>1 1 7 7
1 1 7 7
8 8 4 4
8 8 4 4</pre>
after <code>img2 = scale_down(img, 2)</code> command the new image, <code>img2</code>, will be:
<pre>1 7
8 4</pre>

<code>N</code> : integer <code>10 ≥ N ≥ 1</code>

</td>
</tr>
</table>

	If the image length (in any direction) is not a perfect multiple of N, you should duplicate the last pixels for padding.
	
	For example, let's assume you have the same image:
	
	`1 1 7 7`
	`1 1 7 7`
	`8 8 4 4`
	`8 8 4 4`
	
	after `img2 = scale_down(img, 3)` command the new image, `img2`, will be:
	
	`4 6`
	`7 4`
	
	`Explanation:`
	• For the `117,117,884` block, averaging all those values will yield `4.22`, and `round(4.22)` will give `4`.
	• For the `7,7,4` block, since it is missing the right two columns, we duplicate the last pixels which gives us a `777,777,444` block, and averaging all those values will yield `6`.
	• For the `884,,` block, since it is missing the bottom two rows, we duplicate the last pixels which gives us a `884,884,884` block, and averaging all those values will yield `6.66`, and `round(6.66)` will give `7`.
	• For the `4,,` block, since it is missing both the bottom and the right pixels, we duplicate the last pixels to create the block which gives us a `444,444,444` block, and averaging all those values will yield `4`. (Which essentially is itself)
	
	Hint: As a test: `scale_down(img, 1)` should return the same image as a completely new object (a hard copy, not an alias).
	
	Note that the image is not mutated, and a new image is returned.
	
	Return the image as a list of lists.

`scale_up(img, N)`	This function will scale up the image by `N` times duplicating each pixel in both directions along the way. It will then return the new image.

	For a single channel image let's assume the image is:
	
	`1 6`
	`9 3`
	
	after `img2 = scale_up(img, 2)` command the new image, `img2`, will be:
	
	`1 1 6 6`
	`1 1 6 6`
	`9 9 3 3`
	`9 9 3 3`
	
	• `N` : integer `10 ≥ N ≥ 1`
	
	Hint: As a test: `scale_up(img, 1)` should return the same image as a completely new object (a hard copy, not an alias).
	
	Note that the image is not mutated, and a new image is returned.
	
	Return the image as a list of lists.

`apply_window(img, window)`	This function will take a `3x3` window list, and slide around the image to determine the new values for each pixel. In order to do so, you would overlap the window matrix's middle element (i.e. `window[1][1]`) to the target pixel, and multiply and sum each pixel with its corresponding values in the window. This is the new pixel value.
	
	For example, for a single channel `3x3` image:
	
	`1 2 3 4`
	`5 6 7 8`
	`9 10 11 12`
	
	and a window of:
	
	`0 1 0`
	`1 -4 1`
	`0 1 0`
	
	`1 2 3 4`
	`5 **6** 7 8`
	`9 10 11 12`
	
	For **`pixel[1][1] (6):`** To find the new value of the pixel, we overlap the window's middle element

to the target pixel, and multiply and sum corresponding values to find the new value.

```
new pixel value = 1*0 + 2*1 + 3*0 + 5*1 +
6*-4 + 7*1 + 9*0 + 10*1 + 11*0
```

which yields to 0. So the new value of `pixel[1][1]` will be 0.

```
 1  2  3  4
 5  6  7  8
 9 10 11 12
```

For **`pixel[1][2] (7):`** To find the new value of the pixel, we overlap the window's middle element to the target pixel, and multiply and sum corresponding values to find the new value.

```
new pixel value = 2*0 + 3*1 + 4*0 + 6*1 +
7*-4 + 8*1 + 10*0 + 11*1 + 12*0
```

which yields to 0. So the new value of `pixel[1][2]` will be 0.

For the corner cases where there is no overlapping pixel value, you will *duplicate the nearest pixel value*.

```
1  1  2
1  1  2  3  4
5  5  6  7  8
   9 10 11 12
```

For **`pixel[0][0] (1):`** To find the new value of the pixel, we overlap the window's middle element to the target pixel, and multiply and sum corresponding values to find the new value.

```
new pixel value = 1*0 + 1*1 + 2*0 + 1*1 +
1*-4 + 2*1 + 5*0 + 5*1 + 6*0
```

which yields to 5. So the new value of `pixel[0][0]` will be 5.

If you repeat these operations for each pixel, for this example, you should get:

```
5  4  4  3
1  0  0  0
```

	0 0 0 0 General image rules apply here too: ● You should round the target pixel value (i.e. `round()`) ● Pixel value cannot be less than 0, (set to 0 if it is) ● Pixel value cannot be greater than 255, (set to 255 if it is) **Note that the image is not mutated, and a new image is returned.** Return the image as a list of lists.

```
# Generate random image
>>> orig = generate_random(3,2)
>>> print(orig)

[[{'red': 10, 'green': 15, 'blue': 20}, {'red': 25, 'green': 30, 'blue': 35}],
[{'red': 40, 'green': 45, 'blue': 50}, {'red': 55, 'green': 60, 'blue': 65}],
[{'red': 70, 'green': 75, 'blue': 80}, {'red': 85, 'green': 90, 'blue': 95}]]

# Scale down (for copying).
>>> img = scale_down(orig, 1)
>>> print(img)

[[{'red': 10, 'green': 15, 'blue': 20}, {'red': 25, 'green': 30, 'blue': 35}],
[{'red': 40, 'green': 45, 'blue': 50}, {'red': 55, 'green': 60, 'blue': 65}],
[{'red': 70, 'green': 75, 'blue': 80}, {'red': 85, 'green': 90, 'blue': 95}]]

# They both have the same contents
>>> print(img == orig)
True

# But they are not the same object
>>> print(img is orig)
False

# Enhance. Only the Red channel is enhanced. (i.e. round(10 * 1.82) )
>>> enhance(img, 1.82, 'r')
>>> print(img)

[[{'red': 18, 'green': 15, 'blue': 20}, {'red': 46, 'green': 30, 'blue': 35}],
[{'red': 73, 'green': 45, 'blue': 50}, {'red': 100, 'green': 60, 'blue': 65}],
[{'red': 127, 'green': 75, 'blue': 80}, {'red': 155, 'green': 90, 'blue': 95}]]
```

```
# Enhance. Green and Blue channels are enhanced. (i.e. round(20 * 1.26) and
round(10 * 1.26) )
>>> enhance(img, 1.26, 'gb')
>>> print(img)

[[{'red': 18, 'green': 19, 'blue': 25}, {'red': 46, 'green': 38, 'blue': 44}],
[{'red': 73, 'green': 57, 'blue': 63}, {'red': 100, 'green': 76, 'blue': 82}],
[{'red': 127, 'green': 94, 'blue': 101}, {'red': 155, 'green': 113, 'blue':
120}]]

# Scale up. (Will create/return a new object)
>>> img = scale_up(img, 3)
>>> print(img)

[[{'red': 18, 'green': 19, 'blue': 25}, {'red': 18, 'green': 19, 'blue': 25},
{'red': 18, 'green': 19, 'blue': 25}, {'red': 46, 'green': 38, 'blue': 44},
{'red': 46, 'green': 38, 'blue': 44}, {'red': 46, 'green': 38, 'blue': 44}],
[{'red': 18, 'green': 19, 'blue': 25}, {'red': 18, 'green': 19, 'blue': 25},
{'red': 18, 'green': 19, 'blue': 25}, {'red': 46, 'green': 38, 'blue': 44},
{'red': 46, 'green': 38, 'blue': 44}, {'red': 46, 'green': 38, 'blue': 44}],
[{'red': 18, 'green': 19, 'blue': 25}, {'red': 18, 'green': 19, 'blue': 25},
{'red': 18, 'green': 19, 'blue': 25}, {'red': 46, 'green': 38, 'blue': 44},
{'red': 46, 'green': 38, 'blue': 44}, {'red': 46, 'green': 38, 'blue': 44}],
[{'red': 73, 'green': 57, 'blue': 63}, {'red': 73, 'green': 57, 'blue': 63},
{'red': 73, 'green': 57, 'blue': 63}, {'red': 100, 'green': 76, 'blue': 82},
{'red': 100, 'green': 76, 'blue': 82}, {'red': 100, 'green': 76, 'blue': 82}],
[{'red': 73, 'green': 57, 'blue': 63}, {'red': 73, 'green': 57, 'blue': 63},
{'red': 73, 'green': 57, 'blue': 63}, {'red': 100, 'green': 76, 'blue': 82},
{'red': 100, 'green': 76, 'blue': 82}, {'red': 100, 'green': 76, 'blue': 82}],
[{'red': 73, 'green': 57, 'blue': 63}, {'red': 73, 'green': 57, 'blue': 63},
{'red': 73, 'green': 57, 'blue': 63}, {'red': 100, 'green': 76, 'blue': 82},
{'red': 100, 'green': 76, 'blue': 82}, {'red': 100, 'green': 76, 'blue': 82}],
[{'red': 127, 'green': 94, 'blue': 101}, {'red': 127, 'green': 94, 'blue':
101}, {'red': 127, 'green': 94, 'blue': 101}, {'red': 155, 'green': 113,
'blue': 120}, {'red': 155, 'green': 113, 'blue': 120}, {'red': 155, 'green':
113, 'blue': 120}], [{'red': 127, 'green': 94, 'blue': 101}, {'red': 127,
'green': 94, 'blue': 101}, {'red': 127, 'green': 94, 'blue': 101}, {'red': 155,
'green': 113, 'blue': 120}, {'red': 155, 'green': 113, 'blue': 120}, {'red':
155, 'green': 113, 'blue': 120}], [{'red': 127, 'green': 94, 'blue': 101},
{'red': 127, 'green': 94, 'blue': 101}, {'red': 127, 'green': 94, 'blue': 101},
{'red': 155, 'green': 113, 'blue': 120}, {'red': 155, 'green': 113, 'blue':
120}, {'red': 155, 'green': 113, 'blue': 120}]]

# Scale down. (Will create/return a new object)
>>> img = scale_down(img, 3)
>>> print(img)
```

```
[[{'red': 18, 'green': 19, 'blue': 25}, {'red': 46, 'green': 38, 'blue': 44}],
[{'red': 73, 'green': 57, 'blue': 63}, {'red': 100, 'green': 76, 'blue': 82}],
[{'red': 127, 'green': 94, 'blue': 101}, {'red': 155, 'green': 113, 'blue':
120}]]

# Mirror the image in X direction.
>>> mirror_x(img)
>>> print(img)

[[{'red': 46, 'green': 38, 'blue': 44}, {'red': 18, 'green': 19, 'blue': 25}],
[{'red': 100, 'green': 76, 'blue': 82}, {'red': 73, 'green': 57, 'blue': 63}],
[{'red': 155, 'green': 113, 'blue': 120}, {'red': 127, 'green': 94, 'blue':
101}]]

# Mirror the image in Y direction.
>>> mirror_y(img)
>>> print(img)

[[{'red': 155, 'green': 113, 'blue': 120}, {'red': 127, 'green': 94, 'blue':
101}], [{'red': 100, 'green': 76, 'blue': 82}, {'red': 73, 'green': 57, 'blue':
63}], [{'red': 46, 'green': 38, 'blue': 44}, {'red': 18, 'green': 19, 'blue':
25}]]

# Rotate the image 90 degrees to the right. (Will create/return a new object).
>>> img2 = rotate90(img)
>>> print(img2)

[[{'red': 46, 'green': 38, 'blue': 44}, {'red': 100, 'green': 76, 'blue': 82},
{'red': 155, 'green': 113, 'blue': 120}], [{'red': 18, 'green': 19, 'blue':
25}, {'red': 73, 'green': 57, 'blue': 63}, {'red': 127, 'green': 94, 'blue':
101}]]

# Rotate the image 180 degrees to the right. (Will create/return a new object).
>>> img2 = rotate180(img)
>>> print(img2)

[[{'red': 18, 'green': 19, 'blue': 25}, {'red': 46, 'green': 38, 'blue': 44}],
[{'red': 73, 'green': 57, 'blue': 63}, {'red': 100, 'green': 76, 'blue': 82}],
[{'red': 127, 'green': 94, 'blue': 101}, {'red': 155, 'green': 113, 'blue':
120}]]

# Rotate the image 270 degrees to the right. (Will create/return a new object).
>>> img2 = rotate270(img)
>>> print(img2)
```

```
[[{'red': 127, 'green': 94, 'blue': 101}, {'red': 73, 'green': 57, 'blue': 63},
{'red': 18, 'green': 19, 'blue': 25}], [{'red': 155, 'green': 113, 'blue':
120}, {'red': 100, 'green': 76, 'blue': 82}, {'red': 46, 'green': 38, 'blue':
44}]]

# Convert to grayscale.
>>> grayscale(img)
>>> print(img)

[[{'red': 129, 'green': 129, 'blue': 129}, {'red': 107, 'green': 107, 'blue':
107}], [{'red': 86, 'green': 86, 'blue': 86}, {'red': 64, 'green': 64, 'blue':
64}], [{'red': 43, 'green': 43, 'blue': 43}, {'red': 21, 'green': 21, 'blue':
21}]]


# Clear the image.
>>> clear(img)
>>> print(img)

[[{'red': 0, 'green': 0, 'blue': 0}, {'red': 0, 'green': 0, 'blue': 0}],
[{'red': 0, 'green': 0, 'blue': 0}, {'red': 0, 'green': 0, 'blue': 0}],
[{'red': 0, 'green': 0, 'blue': 0}, {'red': 0, 'green': 0, 'blue': 0}]]

# Set value. All channels will be set to the same value since we passed 'rgb'
>>> set_value(img, 80, 'rgb')
>>> print(img)

[[{'red': 80, 'green': 80, 'blue': 80}, {'red': 80, 'green': 80, 'blue': 80}],
[{'red': 80, 'green': 80, 'blue': 80}, {'red': 80, 'green': 80, 'blue': 80}],
[{'red': 80, 'green': 80, 'blue': 80}, {'red': 80, 'green': 80, 'blue': 80}]]

# Set value. Only the Red channel is set to the given value.
>>> set_value(img, 55, 'r')
>>> print(img)

[[{'red': 55, 'green': 80, 'blue': 80}, {'red': 55, 'green': 80, 'blue': 80}],
[{'red': 55, 'green': 80, 'blue': 80}, {'red': 55, 'green': 80, 'blue': 80}],
[{'red': 55, 'green': 80, 'blue': 80}, {'red': 55, 'green': 80, 'blue': 80}]]

## These are example window values that cause different effects on an image.
## They are all used in the apply_window(img, w) function.
## So implementing one function will let you test all different window values
# Generate random image
>>> img = generate_random(2,3)
>>> print(img)
```

```
[[{'red': 26, 'green': 12, 'blue': 131}, {'red': 137, 'green': 67, 'blue': 44},
{'red': 68, 'green': 141, 'blue': 23}], [{'red': 53, 'green': 234, 'blue': 24},
{'red': 37, 'green': 137, 'blue': 148}, {'red': 34, 'green': 95, 'blue': 240}]]

# identity filter 3x3
>>> w = [[0,0,0], [0,1,0], [0,0,0]]
>>> img2 = apply_window(img, w)
>>> print(img2)

# for this window, the resulting image should look like the original image
since summing all the multiplications of all the neighboring pixels will yield
the target pixel (all others are 0, except the target
>>> print(img == img2)
True

# box blur filter 3x3
>>> w = [[1/9, 1/9, 1/9], [1/9, 1/9, 1/9], [1/9, 1/9, 1/9]]
>>> img2 = apply_window(img, w)

[[{'red': 58, 'green': 87, 'blue': 90}, {'red': 65, 'green': 101, 'blue': 90},
{'red': 72, 'green': 114, 'blue': 90}], [{'red': 53, 'green': 145, 'blue': 78},
{'red': 53, 'green': 128, 'blue': 114}, {'red': 54, 'green': 111, 'blue':
150}]]

# gaussian blur filter 3x3
>>> w = [[1/16, 2/16, 1/16], [2/16, 4/16, 2/16], [1/16, 2/16, 1/16]]
>>> img2 = apply_window(img, w)

[[{'red': 53, 'green': 72, 'blue': 96}, {'red': 79, 'green': 92, 'blue': 80},
{'red': 73, 'green': 118, 'blue': 75}], [{'red': 50, 'green': 164, 'blue': 69},
{'red': 53, 'green': 131, 'blue': 120}, {'red': 47, 'green': 110, 'blue':
170}]]

# edge filter (ver 1) 3x3
>>> w = [[0, -1, 0], [-1, 4, -1], [0, -1, 0]]
>>> img2 = apply_window(img, w)

[[{'red': 0, 'green': 0, 'blue': 194}, {'red': 255, 'green': 0, 'blue': 0},
{'red': 0, 'green': 120, 'blue': 0}], [{'red': 43, 'green': 255, 'blue': 0},
{'red': 0, 'green': 15, 'blue': 136}, {'red': 0, 'green': 0, 'blue': 255}]]

# edge filter (ver 2) 3x3
>>> w = [[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]]
>>> img2 = apply_window(img, w)
```

```
[[{'red': 0, 'green': 0, 'blue': 255}, {'red': 255, 'green': 0, 'blue': 0},
{'red': 0, 'green': 244, 'blue': 0}], [{'red': 2, 'green': 255, 'blue': 0},
{'red': 0, 'green': 81, 'blue': 255}, {'red': 0, 'green': 0, 'blue': 255}]]

# sharpening filter 3x3
>>> w = [[0, -1, 0], [-1, 5, -1], [0, -1, 0]]
>>> img2 = apply_window(img, w)

[[{'red': 0, 'green': 0, 'blue': 255}, {'red': 255, 'green': 0, 'blue': 0},
{'red': 33, 'green': 255, 'blue': 0}], [{'red': 96, 'green': 255, 'blue': 0},
{'red': 0, 'green': 152, 'blue': 255}, {'red': 0, 'green': 7, 'blue': 255}]]
```