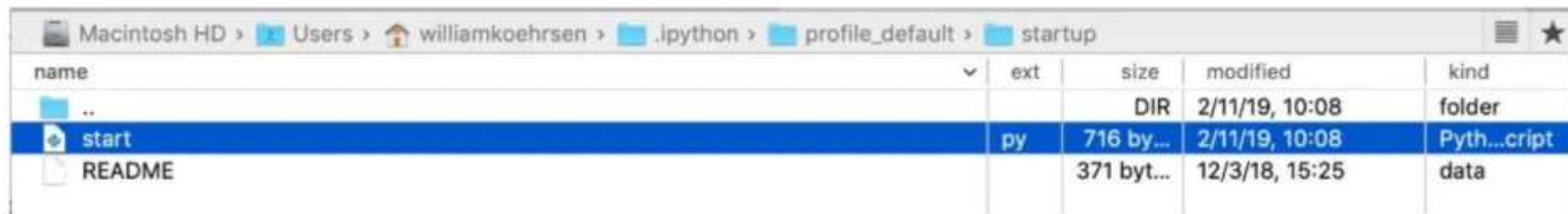


OpenCV в Python

Если вы — частый пользователь IPython или Jupyter Notebooks и вам надоело постоянно импортировать одни и те же библиотеки, то попробуйте этот способ:

- 1.Перейдите к `~/ipython/profile_default`
- 2.Создайте папку `startup`, если она отсутствует
- 3.Добавьте новый файл Python под названием `start.py`
- 4.Добавьте файлы, которые нужно импортировать
- 5.Запустите IPython или Jupyter Notebook, и необходимые библиотеки загрузятся автоматически!

Рассмотрим каждый шаг визуально. Размещение sta



The screenshot shows a Finder window with the path `Macintosh HD > Users > williamkoehrsen > .ipython > profile_default > startup`. The window displays a table of files and folders in the `startup` directory.

name	ext	size	modified	kind
..		DIR	2/11/19, 10:08	folder
start	py	716 by...	2/11/19, 10:08	Pyth...cript
README		371 byt...	12/3/18, 15:25	data

Весь путь сценария Python на `~/ipython/profile_default/startup/start.py`

```
import pandas as pd
import numpy as np

# Pandas options
pd.options.display.max_columns = 30
pd.options.display.max_rows = 20

from IPython import get_ipython
ipython = get_ipython()

# If in ipython, load autoreload extension
if 'ipython' in globals():
    print('\nWelcome to IPython!')
    ipython.magic('load_ext autoreload')
    ipython.magic('autoreload 2')

# Display all cell outputs in notebook
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = 'all'

# Visualization
import plotly.plotly as py
import plotly.graph_objs as go
from plotly.offline import iplot, init_notebook_mode
init_notebook_mode(connected=True)
import cufflinks as cf
cf.go_offline(connected=True)
cf.set_config_file(theme='pearl')

print('Your favorite libraries have been loaded.')
```

如果您是IPython或Jupyter笔记本的频繁用户，并且厌倦了不断导入相同的库，那么请尝试此方法：

转到~/。ipython/profile_default

如果缺少启动文件夹，则创建启动文件夹

添加一个名为的新Python文件start.py

添加要导入的文件

启动IPython或Jupyter Notebook，必要的库将自动加载！

让我们直观地看一下每一步。STA安置

При запуске сессии IPython появится следующее:

```
Williams-MacBook-Pro:~ williamkoehrsen$ ipython
Python 3.6.5 (default, Jun 17 2018, 12:13:06)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.2.0 -- An enhanced Interactive Python. Type '?' for help.

Welcome to IPython!
Your favorite libraries have been loaded.

In [1]:
```

Проверить, загружены ли библиотеки, можно в `globals()`:

```
globals()['pd']
<module 'pandas' from '/usr/local/lib/python3.6/site-
packages/pandas/__init__.py'>

globals()['np']
<module 'numpy' from '/usr/local/lib/python3.6/site-
packages/numpy/__init__.py'>
```

OpenCV — это open source библиотека компьютерного зрения, которая предназначена для анализа, классификации и обработки изображений. Широко используется в таких языках как C, C++, Python и Java.



Часть 1

Теория.

Каждое изображение состоит из набора пикселей. Пиксель — это строительный блок изображения. Если представить изображение в виде сетки, то каждый квадрат в сетке содержит один пиксель, где точке с координатой $(0, 0)$ соответствует верхний левый угол изображения. К примеру, представим, что у нас есть изображение с разрешением 400×300 пикселей. Это означает, что наша сетка состоит из 400 строк и 300 столбцов. В совокупности в нашем изображении есть $400 \times 300 = 120000$ пикселей.

В большинстве изображений пиксели представлены двумя способами: в оттенках серого и в цветовом пространстве RGB. В изображениях в оттенках серого каждый пиксель имеет значение между 0 и 255, где 0 соответствует чёрному, а 255 соответствует белому. А значения между 0 и 255 принимают различные оттенки серого, где значения ближе к 0 более тёмные, а значения ближе к 255 более светлые.



Цветные пиксели обычно представлены в цветовом пространстве RGB(red, green, blue — красный, зелёный, синий), где одно значение для красной компоненты, одно для зелёной и одно для синей. Каждая из трёх компонент представлена целым числом в диапазоне от 0 до 255 включительно, которое указывает как «много» цвета содержится. Исходя из того, что каждая компонента представлена в диапазоне [0,255], то для того, чтобы представить насыщенность каждого цвета, нам будет достаточно 8-битного целого беззнакового числа. Затем мы объединяем значения всех трёх компонент в кортеж вида (красный, зеленый, синий). К примеру, чтобы получить белый цвет, каждая из компонент должна равняться 255: (255, 255, 255). Тогда, чтобы получить чёрный цвет, каждая из компонент должна быть равной 0: (0, 0, 0). Ниже

представленные в виде RGB кортежей:

Black	<code>rgb(0, 0, 0)</code>
White	<code>rgb(255, 255, 255)</code>
Red	<code>rgb(255, 0, 0)</code>
Blue	<code>rgb(0, 0, 255)</code>
Green	<code>rgb(0, 255, 0)</code>
Yellow	<code>rgb(255, 255, 0)</code>
Magenta	<code>rgb(255, 0, 255)</code>
Cyan	<code>rgb(0, 255, 255)</code>
Violet	<code>rgb(136, 0, 255)</code>
Orange	<code>rgb(255, 136, 0)</code>

И импортировать библиотеку. Есть несколько путей импорта, самый распространённый — это использовать выражение:

```
import cv2
```

```
from cv2 import cv2
```

Загрузка, отображение и сохранение изображения

```
def loading_displaying_saving():  
    img = cv2.imread('girl.jpg', cv2.IMREAD_GRAYSCALE)  
    cv2.imshow('girl', img)  
    cv2.waitKey(0)  
    cv2.imwrite('graygirl.jpg', img)
```


Для загрузки изображения мы используем функцию **cv2.imread()**,

где первым аргументом указывается путь к изображению, а вторым аргументом, который является необязательным, мы указываем, в каком цветовом пространстве мы хотим считать наше изображение. Чтобы считать изображение в RGB — `cv2.IMREAD_COLOR`, в оттенках серого — `cv2.IMREAD_GRAYSCALE`.

По умолчанию данный аргумент принимает значение `.IMREAD_COLOR`.

Данная функция возвращает 2D (для изображения в оттенках серого) либо 3D (для цветного изображения) массив NumPy.

Форма массива для цветного изображения: высота x ширина x 3, где 3 — это байты, по одному байту на каждую из компонент. В изображениях в оттенках серого всё немного проще: высота x ширина.

С помощью функции
`cv2.imshow()`

мы отображаем изображение на нашем экране. В качестве первого аргумента мы передаём функции название нашего окна, а вторым аргументом изображение, которое мы загрузили с диска, однако, если мы далее не укажем функцию `cv2.waitKey()`, то изображение моментально закроется.

Данная функция останавливает выполнение программы до нажатия клавиши, которую нужно передать первым аргументом. Для того, чтобы любая клавиша была засчитана передаётся 0. Слева представлено изображение в оттенках серого, а справа в формате RGB:



Доступ к пикселям и манипулирование ими

Для того, чтобы узнать высоту, ширину и количество каналов у изображения можно использовать атрибут `shape`:

```
print("Высота:" + str(img.shape[0]))  
print("Ширина:" + str(img.shape[1]))  
print("Количество каналов:" + str(img.shape[2]))
```

Важно помнить, что у изображений в оттенках серого `img.shape[2]` будет недоступно, так как данные изображения представлены в виде 2D массива.

Чтобы получить доступ к значению пикселя, нам просто нужно указать координаты x и y пикселя, который нас интересует. Также важно помнить, что библиотека OpenCV хранит каналы формата RGB в обратном порядке, в то время как мы думаем в терминах красного, зеленого и синего, то OpenCV хранит их в порядке синего, зеленого и красного цветов:

```
(b, g, r) = img[0, 0]  
print("Красный: {}, Зелёный: {}, Синий: {}".format(r, g, b))
```

Сначала мы берём пиксель, который расположен в точке (0,0). Данный пиксель, да и любой другой пиксель, представлены в виде кортежа. Заметьте, что название переменных расположены в порядке b, g и r. В следующей строке выводим значение каждого канала на экран. Как можно увидеть, доступ к значениям пикселей довольно прост, также просто можно и манипулировать значениями пикселей:

```
img[0, 0] = (255, 0, 0)
(b, g, r) = img[0, 0]
print("Красный: {}, Зелёный: {}, Синий: {}".format(r, g, b))
```

В первой строке мы устанавливаем значение пикселя (0, 0) равным (255, 0, 0), затем мы снова берём значение данного пикселя и выводим его на экран, в результате на консоль вывелось следующее:

```
Красный: 251, Зелёный: 43, Синий: 65
Красный: 0, Зелёный: 0, Синий: 255
```

```
import cv2
```

```
def loading_displaying_saving():  
    img = cv2.imread('images/girl.jpg', cv2.IMREAD_GRAYSCALE)  
    cv2.imshow('girl', img)  
    cv2.waitKey(0)  
    cv2.imwrite('graygirl.jpg', img)
```

```
def accessing_and_manipulating():  
    img = cv2.imread('images/girl.jpg')  
    print("Высота:" + str(img.shape[0]))  
    print("Ширина:" + str(img.shape[1]))  
    print("Количество каналов:" + str(img.shape[2]))  
    (b, g, r) = img[0, 0]  
    print("Красный: {}, Зелёный: {}, Синий: {}".format(r, g, b))  
    img[0, 0] = (255, 0, 0)  
    (b, g, r) = img[0, 0]  
    print("Красный: {}, Зелёный: {}, Синий: {}".format(r, g, b))
```


Часть 2



Приступаем к базовым преобразованиям изображений: изменение размера, смещение вдоль осей, кадрирование(обрезка), поворот.

Изменение размера изображения

Первый метод, который мы изучим — это как поменять высоту и ширину у изображения. Для этого в opencv есть такая функция как `resize()`:

```
def resizing():  
    res_img = cv2.resize(img, (500, 900), cv2.INTER_NEAREST)
```

Данная функция первым аргументом принимает изображение, размер которого мы хотим изменить, вторым — кортеж, который должен содержать в себе ширину и высоту для нового изображения, третьим — метод интерполяции(необязательный). Интерполяция — это алгоритм, который находит неизвестные промежуточные значения по имеющемуся набору известных значений. Фактически, это то, как будут заполняться новые пиксели при модификации размера изображения.

К примеру, интерполяция методом ближайшего соседа (`cv2.INTER_NEAREST`) просто берёт для каждого пикселя итогового изображения один пиксель исходного, который наиболее близкий к его положению — это самый простой и быстрый способ. Кроме этого метода в `opencv` существуют следующие: `cv2.INTER_AREA`, `cv2.INTER_LINEAR` (используется по умолчанию), `cv2.INTER_CUBIC` и `cv2.INTER_LANCZOS4`. Наиболее предпочтительным методом интерполяции для сжатия изображения является `cv2.INTER_AREA`, для увеличения — `cv2.INTER_LINEAR`. От данного метода зависит качество конечного изображения, но как показывает практика, если мы уменьшаем/увеличиваем изображение меньше, чем в 1.5 раза, то не важно каким методом интерполяции мы воспользовались — качество будет схожим. Данное утверждение можно проверить на практике. Напишем следующий код:

```
res_img_nearest = cv2.resize(img, (int(w / 1.4), int(h / 1.4)),
                               cv2.INTER_NEAREST)
res_img_linear = cv2.resize(img, (int(w / 1.4), int(h / 1.4)),
                             cv2.INTER_LINEAR)
```

Слева — изображение с интерполяцией методом ближайшего соседа, справа — изображение с билинейной интерполяцией:



Очень важно при изменении размера изображения учитывать соотношение сторон. Соотношение сторон — это пропорциональное соотношение ширины и высоты изображения. Если мы забудем о данном понятии, то получим изображение такого плана:



Поэтому текущую функцию для изменения размера необходимо модифицировать:

```
def resizing(new_width=None, new_height=None, interp=cv2.INTER_LINEAR):
    h, w = img.shape[:2]

    if new_width is None and new_height is None:
        return img

    if new_width is None:
        ratio = new_height / h
        dimension = (int(w * ratio), new_height)

    else:
        ratio = new_width / w
        dimension = (new_width, int(h * ratio))

    res_img = cv2.resize(img, dimension, interpolation=interp)
```

Соотношение сторон мы вычисляем в переменной `ratio`. В зависимости от того, какой параметр не равен `None`, мы берём установленную нами новую высоту/ширину и делим на старую высоту/ширину. Далее, в переменной `dimension` мы определяем новые размеры изображения и передаём в функцию `cv2.resize()`.

Смещение изображения вдоль осей

С помощью функции `cv2.warpAffine()` мы можем перемещать изображение влево и вправо, вниз и вверх, а также любую комбинацию из перечисленного:

```
def shifting():  
    h, w = img.shape[:2]  
    translation_matrix = np.float32([[1, 0, 200], [0, 1, 300]])  
    dst = cv2.warpAffine(img, translation_matrix, (w, h))  
    cv2.imshow('Изображение, сдвинутое вправо и вниз', dst)  
    cv2.waitKey(0)
```

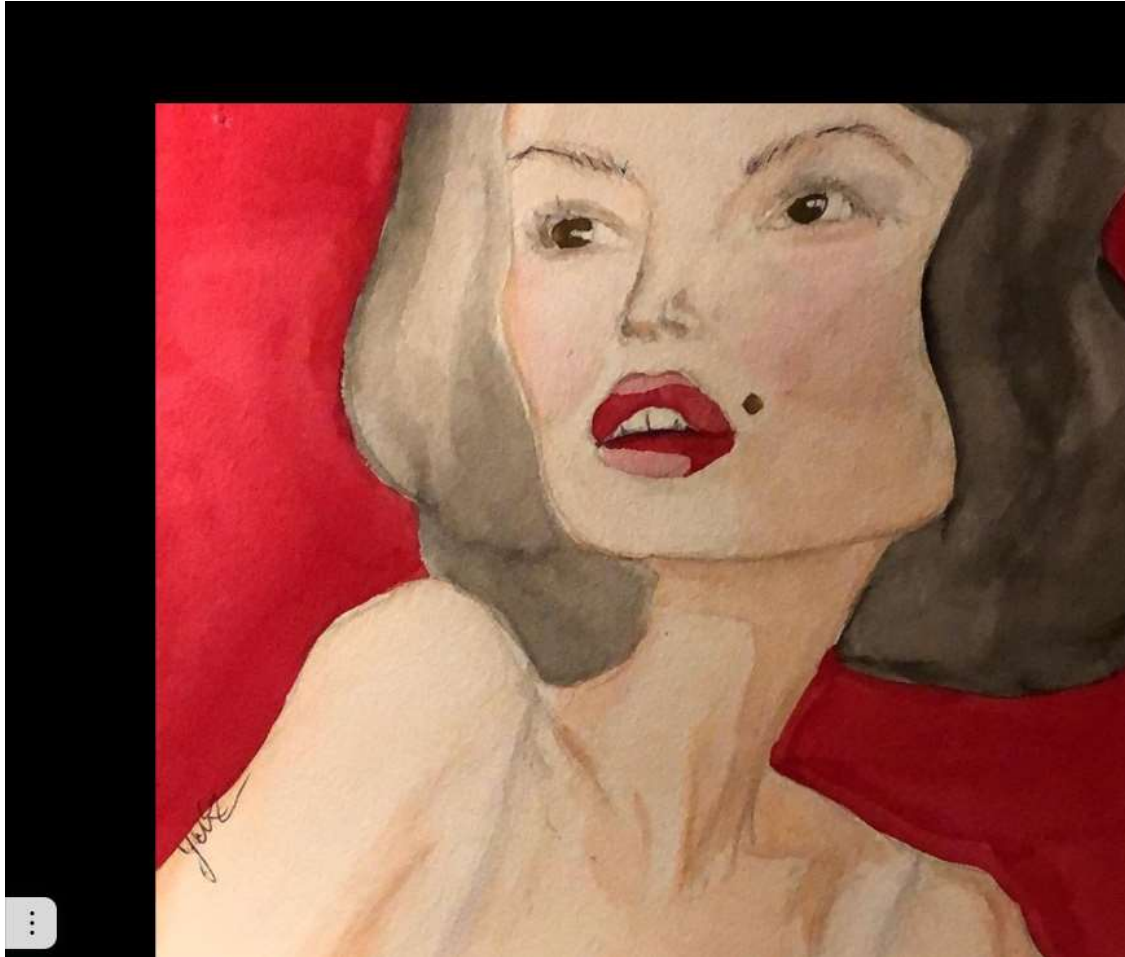
Сначала, в переменной `translation_matrix` мы создаём матрицу преобразований как показано ниже:

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Первая строка матрицы — $[1, 0, t_x]$, где t_x — количество пикселей, на которые мы будем сдвигать изображение влево или вправо. Отрицательное значения t_x будет сдвигать изображение влево, положительное — вправо.

Вторая строка матрицы — $[0, 1, t_y]$, где t_y — количество пикселей, на которые мы будем сдвигать изображение вверх или вниз. Отрицательное значения t_y будет сдвигать изображение вверх, положительное — вниз. Важно помнить, что данная матрица определяется как массив с плавающей точкой.

На следующей строчке и происходит сдвиг изображения вдоль осей, с помощью, как я писал выше, функции `cv2.warpAffine()`, которая первым аргументом принимает изображение, вторым — матрицу, третьим — размеры нашего изображения. Если вы запустите данный код, то увидите следующее:



Вырез фрагмента изображения

Для того, чтобы вырезать интересующий вас фрагмент из изображения, достаточно воспользоваться следующим кодом:

```
def cropping():  
    crop_img = img[10:450, 300:750]
```

В данной строчке мы предоставляем массив `numpy` для извлечения прямоугольной области изображения, начиная с (300, 10) и заканчивая (750, 450), где 10 — это начальная координата по `y`, 300 — начальная координата по `x`, 450 — конечная координата по `y` и 750 — конечная координата по `x`. Выполнив код выше, мы увидим, что обрезали лицо девочке:



Поворот изображения

И, последнее, что мы на сегодня рассмотрим — это как повернуть изображение на некоторый угол:

```
def rotation():  
    (h, w) = img.shape[:2]  
    center = (int(w / 2), int(h / 2))  
    rotation_matrix = cv2.getRotationMatrix2D(center, -45, 0.6)  
    rotated = cv2.warpAffine(img, rotation_matrix, (w, h))
```

Когда мы поворачиваем изображение, нам нужно указать, вокруг какой точки мы будем вращаться, именно это принимает первым аргументом функция `cv2.getRotationMatrix2D()`. В данном случае я указал центр изображения, однако `opencv` позволяет указать любую произвольную точку, вокруг которой вы захотите вращаться. Следующим аргументом данная функция принимает угол, на который мы хотим повернуть наше изображение, а последним аргументом — коэффициент масштабирования. Мы используем `0.6`, то есть уменьшаем изображение на 40%, для того, чтобы оно поместилось в кадр. Данная функция возвращает массив `numpy`, который мы передаём вторым аргументом в функцию `cv2.warpAffine()`. В итоге, у вас на экране должно отобразиться следующее изображение:



Часть 3



Арифметика изображений

Надеюсь, что все знают такие арифметические операции как сложение и вычитание, но при работе с изображениями мы не должны забывать о типе данных.

К примеру, у нас есть RGB изображение, пиксели которого попадают в диапазон $[0, 255]$. Итак, что же произойдёт, если мы попытаемся к пикселю с интенсивностью 250 прибавить 30 или от 70 отнять 100? Если бы мы пользовались стандартными арифметическими правилами, то получили бы 280 и -30 соответственно. Однако, если мы работаем с RGB изображениями, где значения пикселей представлены в виде 8-битного целого беззнакового числа, то 280 и -30 не являются допустимыми значениями. Для того, чтобы разобраться, что же произойдёт, давайте посмотрим на строчки кода ниже:

```
print("opencv addition: {}".format(cv2.add(np.uint8([250]),  
                                           np.uint8([30]))))  
print("opencv subtract: {}".format(cv2.subtract(np.uint8([70]),  
                                                np.uint8([100]))))  
print("numpy addition: {}".format(np.uint8([250]) + np.uint8([30])))  
print("numpy subtract: {}".format(np.uint8([70]) - np.uint8([71])))
```

Как мы видим, сложение и вычитание можно осуществить с помощью функций `opencv add` и `subtract` соответственно, а также с помощью `numpy`. И результаты будут отличаться:

OpenCV выполняет обрезку и гарантирует, что значения пикселей никогда не выйдут за пределы диапазона $[0, 255]$. В numpy же всё происходит немного иначе. Представьте себе обычные настенные часы, где вместо 60 находится 255. Получается, что после достижения 255 следующим числом будет идти 0, а когда мы отнимаем от меньшего числа большее, то после 0 (против часовой стрелки) будет идти 255.

```
opencv addition: 255
```

```
opencv subtract: 0
```

```
numpy addition: 24
```

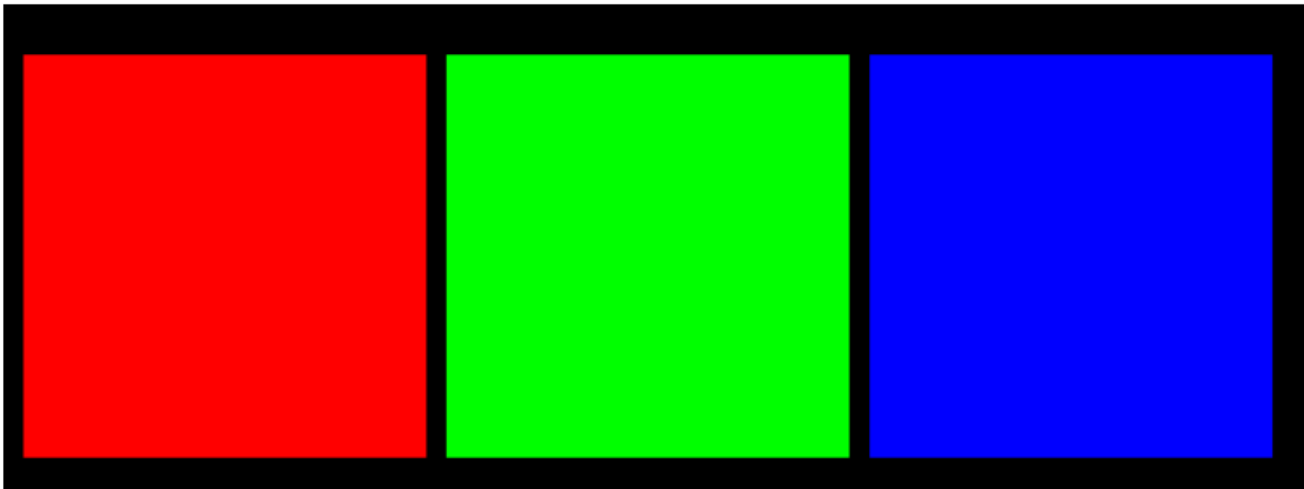
```
numpy subtract: 255
```

Разбиение и слияние каналов

Как мы знаем, RGB изображение состоит из красной, зелёной и синих компонент. И что, если мы захотим разделить изображение на соответствующие компоненты? Для этого в opencv есть специальная функция `split()`.

```
image = cv2.imread('rectangles.png')  
b, g, r = cv2.split(image)  
cv2.imshow('blue', b)  
cv2.imshow('green', g)  
cv2.imshow('red', r)
```

Сначала мы загружаем изображение. Для наглядности работы данной функции взято следующее изображение



Затем разделяем изображение на три канала и показываем каждый канал по отдельности. В результате выполнения данной функции отобразится три изображения в оттенках серого:

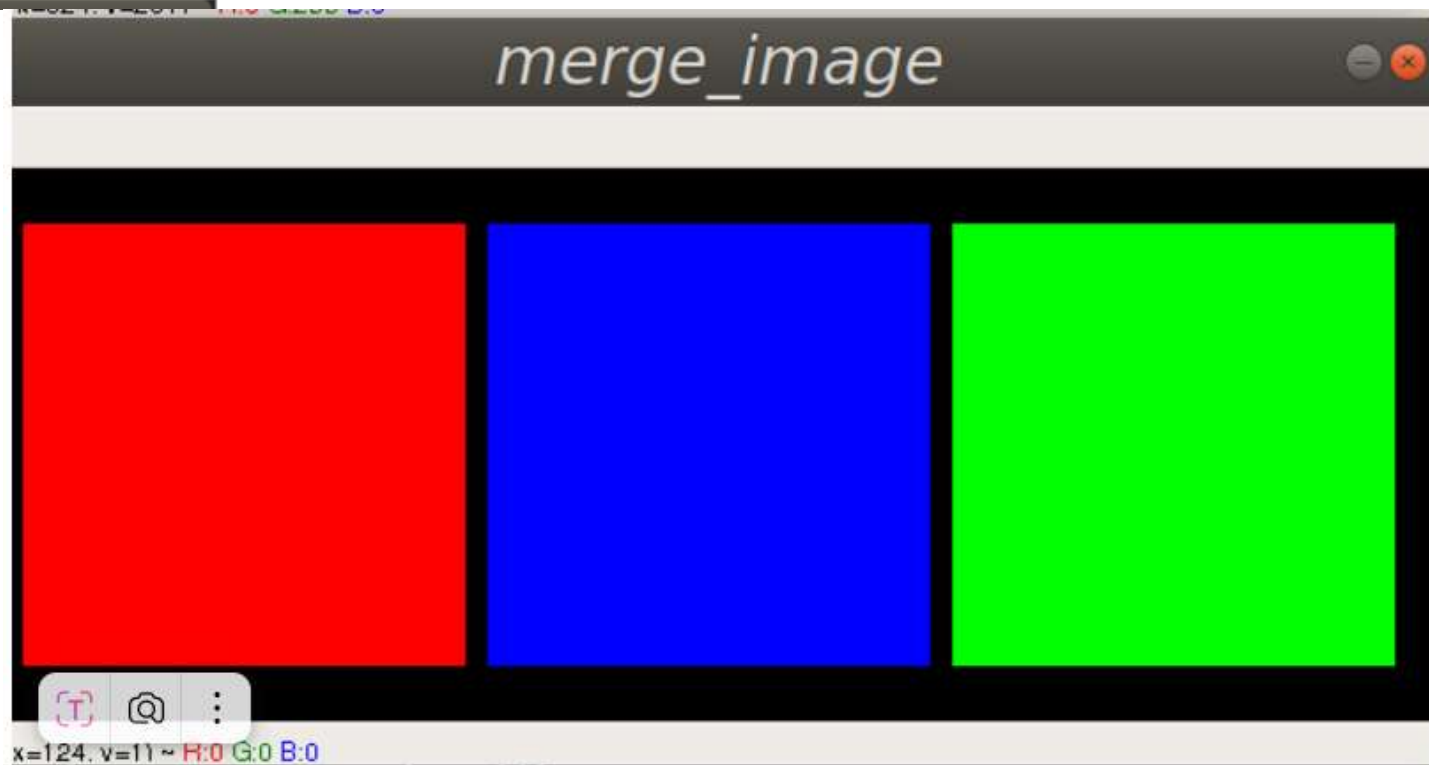
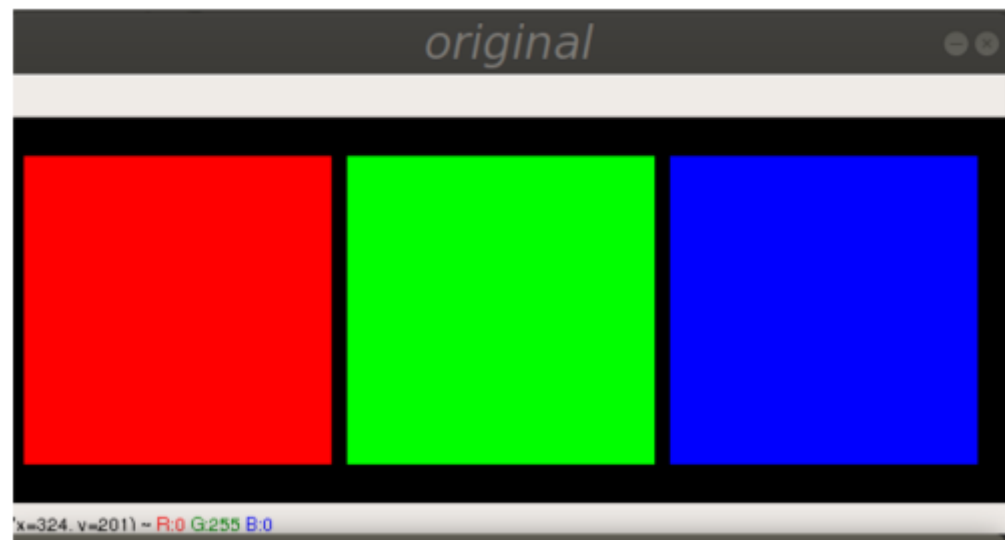


Как мы видим, для каждого изображения каждого канала только прямоугольник с тем же цветом отображается белым. Вот как будет выглядеть каждая компонента для девушки из предыдущих частей:



Как можно увидеть, красный канал очень светлый. Это происходит потому, что оттенки красного очень сильно представлены в нашем изображении. Синий и зелёный каналы, наоборот, очень тёмные. Это случается потому, что на данном изображении очень мало данных цветов. Для того, чтобы объединить каналы воедино, достаточно воспользоваться функцией `merge()`, которая принимает значения каналов:

```
merge_image = cv2.merge([g,b,r])
cv2.imshow('merge_image', merge_image)
cv2.imshow('original', image)
cv2.waitKey(0)
```



Таким образом, мы получаем такое же изображение как оригинальное, за исключением того, что я поменял местами синий с зелёным каналом.

Размытие

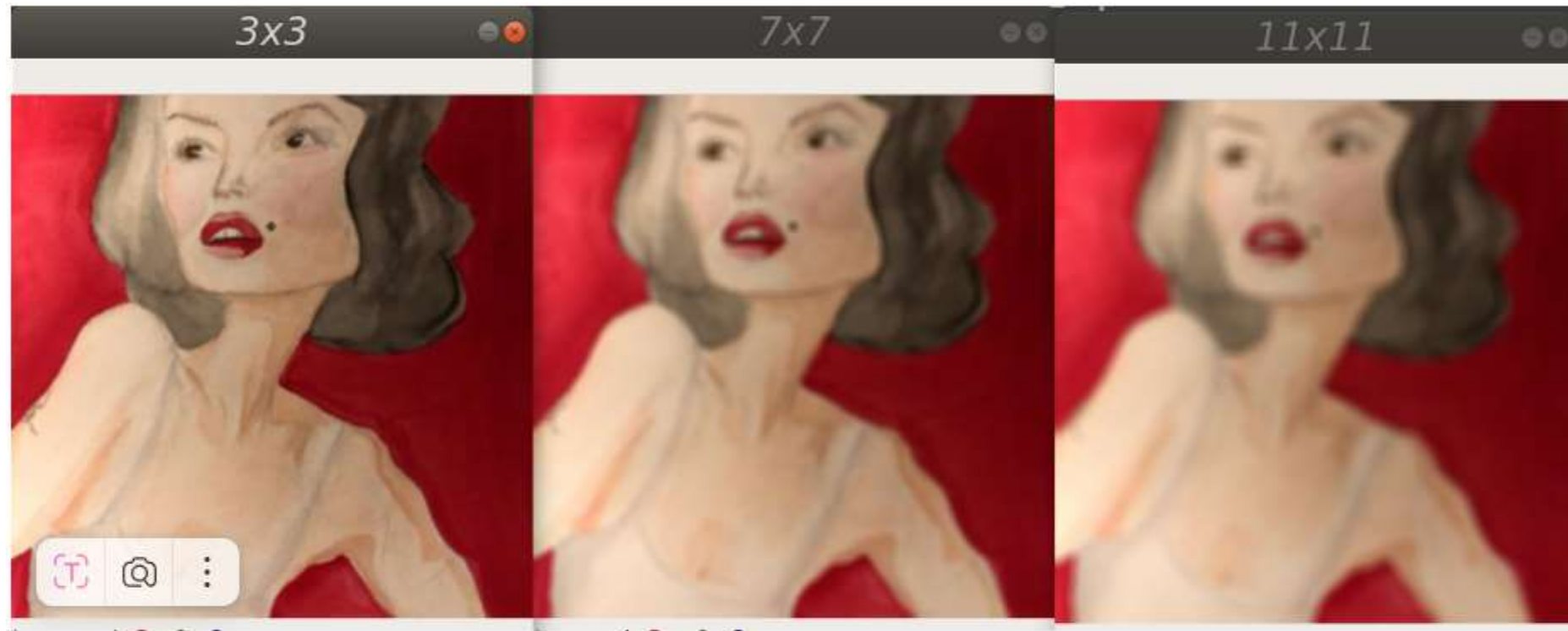
Размытие — это когда более резкие области на изображении теряют свою детализацию, в результате чего изображение становится менее чётким. В `opencv` имеются следующие основные методы размытия: `averaging`(усреднённое), `gaussian`(гауссово) и `median`(медианное).

Averaging

Данный фильтр делает операцию свёртки на изображении с неким ядром, где свёртка — это вычисление нового значения пикселя, при котором учитываются значения соседних пикселей. Ядро свёртки — это квадратная матрица, где пиксель в центре этой матрицы затем устанавливается как среднее значение всех других пикселей, окружающих его. Для того, чтобы воспользоваться данным размытием достаточно вызвать метод `blur()`, который принимает изображение и кортеж, с указанием размера ядра:

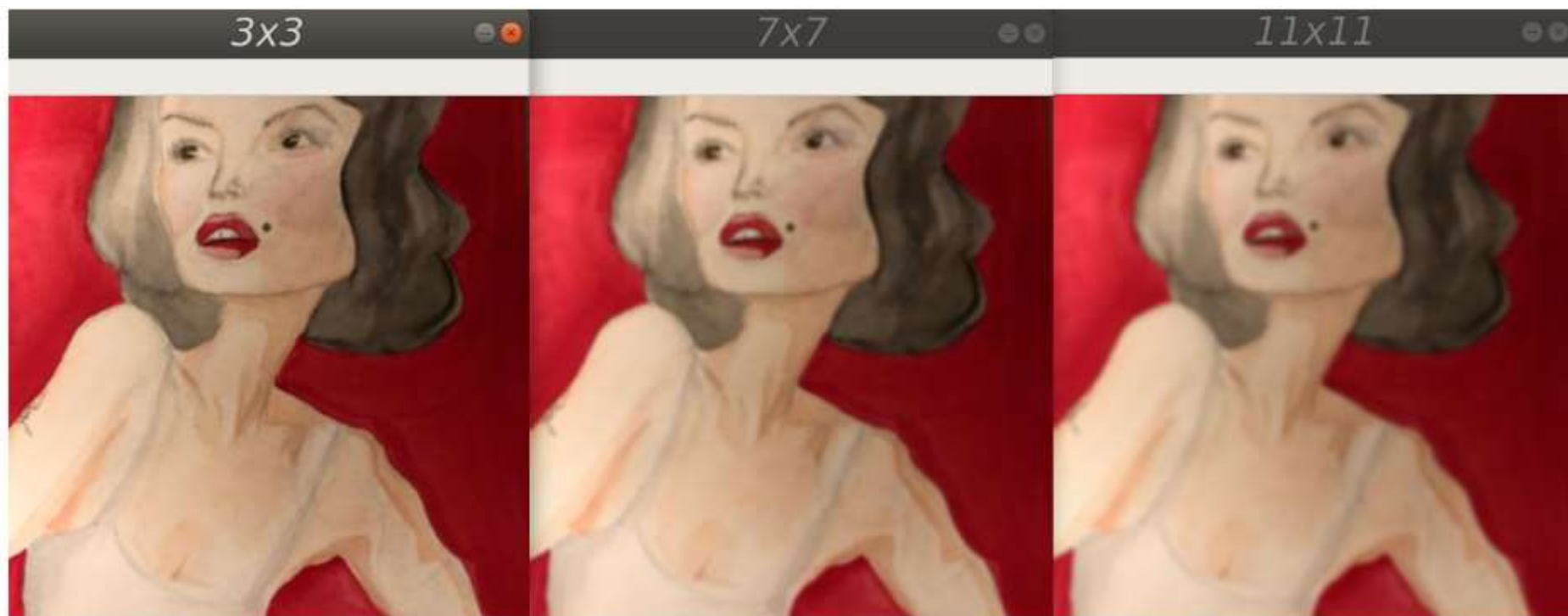
```
def averaging_blurring():  
    image = cv2.imread('girl.jpg')  
    img_blur_3 = cv2.blur(image, (3, 3))  
    img_blur_7 = cv2.blur(image, (7, 7))  
    img_blur_11 = cv2.blur(image, (11, 11))
```

Чем больше размер ядра, тем более размытым будет становиться изображение:



Gaussian

Гауссово размытие похоже на предыдущее размытие, за исключением того, что вместо простого среднего мы теперь используем взвешенное среднее, где соседние пиксели, которые ближе к центральному пикселю, вносят больший «вклад» в среднее. Конечным результатом является то, что наше изображение размыто более естественно:



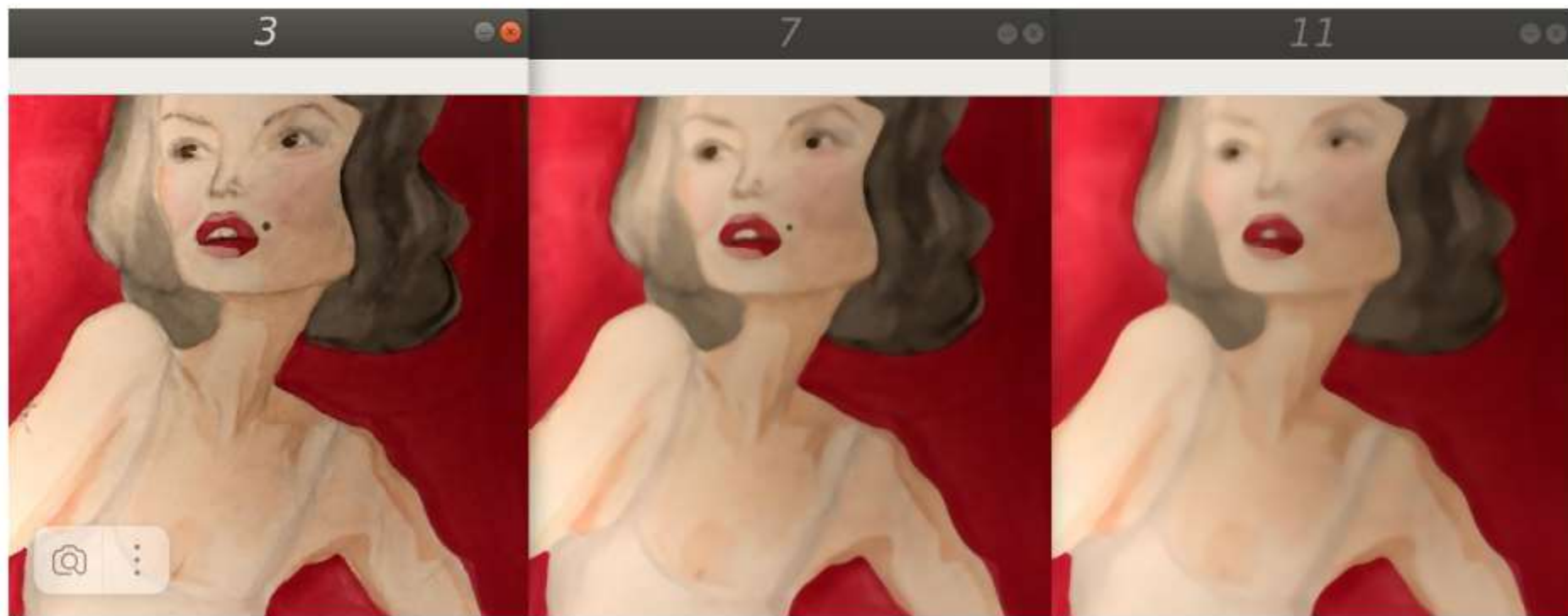
Это размытие реализуется в opencv с помощью функции GaussianBlur(), которая принимает первые два аргумента такие же как и предыдущая функция, а третьим аргументом указываем стандартное отклонение ядра Гаусса. Установив это значение в 0, мы тем самым говорим opencv автоматически вычислять его, в зависимости от размера нашего ядра:

```
def gaussian_blurring():  
    image = cv2.imread('girl.jpg')  
    img_blur_3 = cv2.GaussianBlur(image, (3, 3), 0)  
    img_blur_7 = cv2.GaussianBlur(image, (7, 7), 0)  
    img_blur_11 = cv2.GaussianBlur(image, (11, 11), 0)
```

Median

В медианном размытии центральный пиксель изображения заменяется медианой всех пикселей в области ядра, в результате чего это размытие наиболее эффективно при удалении шума в стиле «соли». Для того, чтобы применить данный вид размытия, необходимо вызвать функцию `medianBlur()` и передать туда два параметра: изображение и размер ядра:

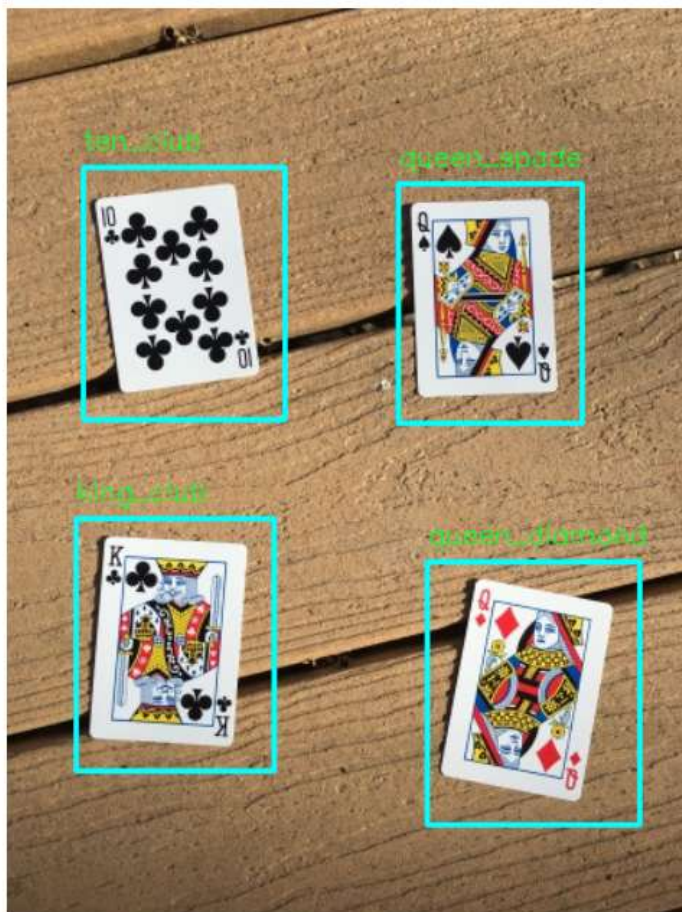
```
def median_blurring():  
    image = cv2.imread('girl.jpg')  
    img_blur_3 = cv2.medianBlur(image, 3)  
    img_blur_7 = cv2.medianBlur(image, 7)  
    img_blur_11 = cv2.medianBlur(image, 11)
```



Часть 4



Распознаем объекты помощью OpenCV на примере игральных карт:



у нас имеется следующее изображение с картами:



А также у нас имеются эталонные изображения каждой карты:



И теперь для того, чтобы детектировать каждую карту, нам необходимо написать три ключевые функции, которые:

1. находит контуры всех карт;
2. находит координаты каждой отдельной карты;
3. распознаёт карту с помощью ключевых точек.

Нахождение контуров карт

```
def find_contours_of_cards(image):  
    blurred = cv2.GaussianBlur(image, (3, 3), 0)  
    T, thresh_img = cv2.threshold(blurred, 215, 255,  
                                  cv2.THRESH_BINARY)  
    (_, cnts, _) = cv2.findContours(thresh_img,  
                                    cv2.RETR_EXTERNAL,  
                                    cv2.CHAIN_APPROX_SIMPLE)  
    return cnts
```

Первым аргументом в данную функцию мы передаём изображение в оттенках серого, к которому применяем гауссово размытие, для того, чтобы было легче найти контуры карт. После этого с помощью функции `threshold()` мы преобразуем наше серое изображение в бинарное. Данная функция принимает первым параметром изображение, вторым — пороговое значение, третьим — это максимальное значение, которое присваивается значениям пикселей, превышающим пороговое значение. Из нашего примера следует, что любое значение пикселя, превышающее 215, устанавливается равным 255, а любое значение, которое меньше 215, устанавливается равным нулю. И последним параметром передаём метод порогового значения. Мы используем `THRESH_BINARY()`, который указывает на то, что значения пикселей, которые больше 215 устанавливаются в максимальное значение, которое мы передали третьим параметром. Данная функция возвращает два значения, где первое — это значение, которое мы передали в данную функцию вторым аргументом, а второе — чёрно-белое изображение, которое выглядит подобным образом:



Теперь мы можем найти контуры наших карт, где контур — это кривая, соединяющая все непрерывные точки, которые имеют одинаковый цвет. Поиск контуров осуществляется с помощью метода `findContours()`, где в качестве первого аргумента эта функция принимает изображение, вторым — это тип контуров, который мы хотим извлечь. Я использую `cv2.RETR_EXTERNAL` для извлечения только внешних контуров. К примеру, для того, чтобы извлечь все контуры используют `cv2.RETR_LIST`, а последний параметром мы указываем метод аппроксимации контура. Мы используем `cv2.CHAIN_APPROX_SIMPLE`, указывая на то, что все лишние точки будут удалены, тем самым экономя память. Например, если вы нашли контур прямой линии, то разве вам нужны все точки этой линии, чтобы представить эту линию? Нет, нам нужны только две конечные точки этой линии. Это как раз то, что и делает `cv2.CHAIN_APPROX_SIMPLE`.

Нахождение координат карт

```
def find_coordinates_of_cards(cnts, image):  
    cards_coordinates = {}  
    for i in range(0, len(cnts)):  
        x, y, w, h = cv2.boundingRect(cnts[i])  
        if w > 20 and h > 30:  
            img_crop = image[y - 15:y + h + 15,  
                             x - 15:x + w + 15]  
            cards_name = find_features(img_crop)  
            cards_coordinates[cards_name] = (x - 15,  
                                             y - 15, x + w + 15, y + h + 15)  
    return cards_coordinates
```

Данная функция принимает контуры, которые мы нашли в предыдущей функции, а также основное изображение в оттенках серого. Первым делом мы создаём словарь, где в роли ключа будет выступать название карты, а в роли значения координаты каждой карты. Далее мы проходимся в цикле по нашим контурам, где с помощью функции `boundingRect()` находим ограничительные рамки каждого контура: начальные `x` и `y` координаты, за которыми следуют ширина и высота рамки. Так получилось, что функция, которая искала контура, нашла аж 31 контур, хотя карт всего 4. Это могут быть незначительные контуры, которые мы дальше сортируем в условии, исходя из размера контура.

Теперь, зная координаты контура, мы можем вырезать по этим координатам каждую карту, что собственно мы и делаем в следующей строчке кода. Далее вырезанное изображение мы передаём в функцию `find_features()` которая опираясь на ключевые точки, возвращает нам название вырезанной карты. После этого добавляем всё в словарь.

Распознавание карт

Распознавание карт будет осуществляться на основе ключевых точек. Ключевые точки — это интересные области изображения. Интересными участками называются такие участки, которые неоднородны. Например, это могут быть углы, так как там происходит резкое изменение интенсивности в двух разных направлениях. Если мы посмотрим на изображение ниже, а потом закроем глаза и попытаемся визуализировать этот образ, то мы вряд ли сможем увидеть что-то конкретное и особенное на этом изображении. Причина этого в том, что изображение не содержит никакой интересной информации

Посмотрите на картинку. Закройте глаза и попытайтесь представить это изображение:



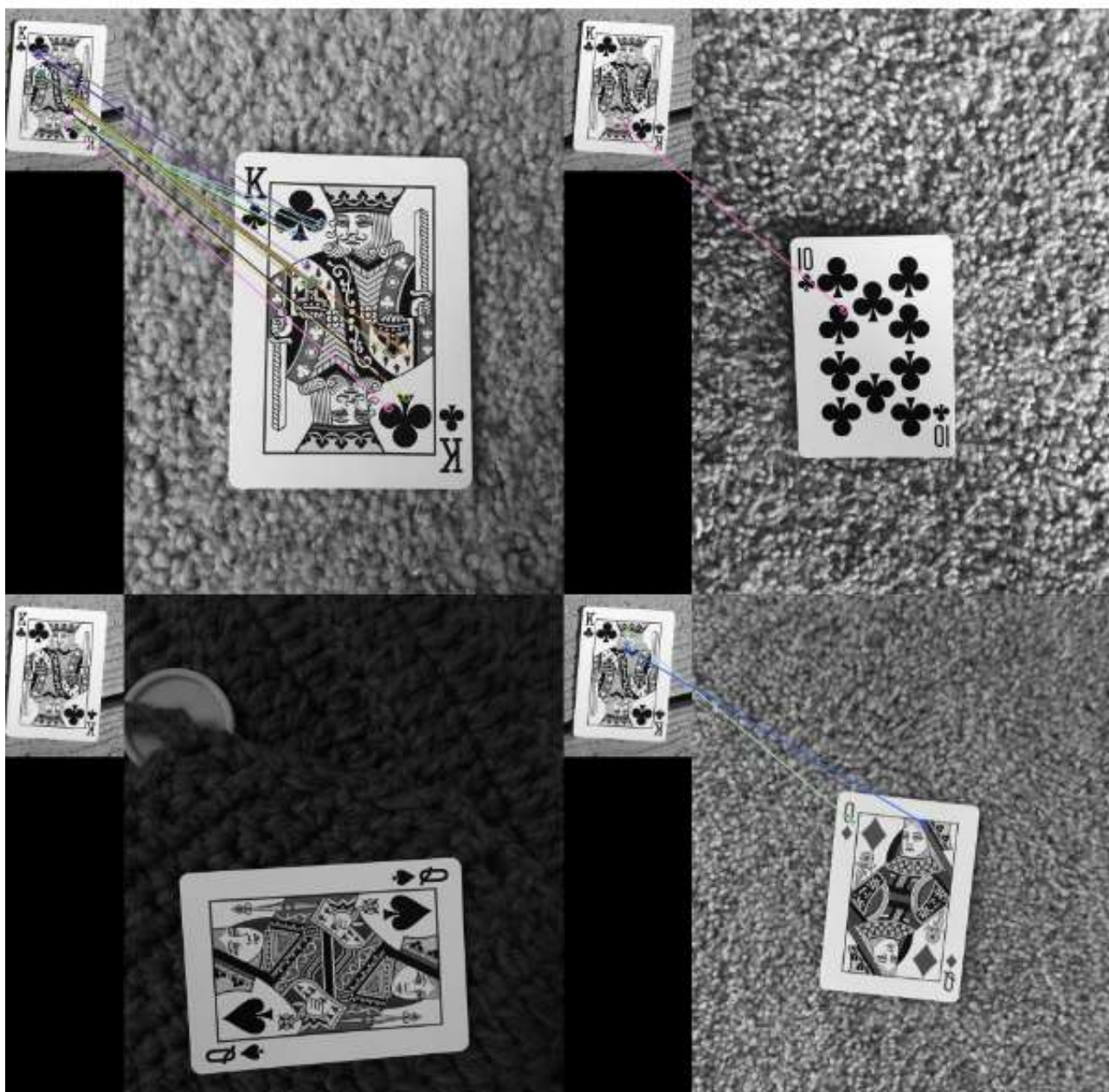
И вы увидите, что помните много деталей об этом образе. Причина этого в том, что на изображении много интересных областей.

```
def find_features(img1):
    correct_matches_dct = {}
    directory = 'images/cards/sample/'
    for image in os.listdir(directory):
        img2 = cv2.imread(directory+image, 0)
        orb = cv2.ORB_create()
        kp1, des1 = orb.detectAndCompute(img1, None)
        kp2, des2 = orb.detectAndCompute(img2, None)
        bf = cv2.BFMatcher()
        matches = bf.knnMatch(des1, des2, k=2)
        correct_matches = []
        for m, n in matches:
            if m.distance < 0.75*n.distance:
                correct_matches.append([m])
                correct_matches_dct[image.split('.')[0]]
                    = len(correct_matches)
    correct_matches_dct =
        dict(sorted(correct_matches_dct.items(),
            key=lambda item: item[1], reverse=True))
    return list(correct_matches_dct.keys())[0]
```

Для обнаружения ключевых точек я использую ORB, который мы инициализируем с помощью вызова функции `ORB_create()`, после чего мы находим ключевые точки и дескрипторы (которые кодируют интересную информацию в ряд чисел) для эталонной карты и для карты, которую мы вырезали из главного изображения. Вот, к примеру, как выглядят ключевые точки для короля:



Затем нам необходимо сопоставить(вычислить расстояние) дескрипторы первого изображения с дескрипторами второго и взять ближайший. Для этого мы создаём BFMatcher объект с помощью вызова метода BFMatcher(). Теперь мы можем с помощью функции knnMatch() найти k лучших совпадений для каждого дескриптора, где k в нашем случае равно 2. Далее нам необходимо выбрать только хорошие совпадения, основываясь на расстоянии. Поэтому мы проходимся в цикле по нашим совпадениям и если оно удовлетворяет условию $m.distance < 0.75 * n.distance$, то мы засчитываем это совпадение как хорошее и добавляем в список. Потом считаем количество хороших совпадений(чем больше, тем лучше) и основываясь на этом делаем вывод, что за карта. Вот какие совпадения были найдены для каждой карты с королём:



И вслед за этим рисуем прямоугольник вокруг карты с помощью функции `draw_rectangle_around_cards()`:

```
def draw_rectangle_around_cards(cards_coordinates, image):  
    for key, value in cards_coordinates.items():  
        rec = cv2.rectangle(image, (value[0], value[1]),  
                             (value[2], value[3]),  
                             (255, 255, 0), 2)  
        cv2.putText(rec, key, (value[0], value[1] - 10),  
                    cv2.FONT_HERSHEY_SIMPLEX,  
                    0.5, (36, 255, 12), 1)  
    cv2.imshow('Image', image)  
    cv2.waitKey(0)
```