**Source Codeof Image Editor**

**adjustFrame.py:**

from tkinter import Toplevel, Label, Scale, Button, HORIZONTAL, RIGHT

import cv2

import numpy as np

s = 100

MAX\_VALUE = 100

class AdjustFrame(Toplevel):

    def \_\_init\_\_(self, master=None):

        Toplevel.\_\_init\_\_(self, master=master)

        self.brightness\_value = 255

        self.previous\_brightness\_value = 255

        self.i=0

        self.contrast\_value = 127

        self.previous\_contrast\_value = 127

        self.original\_image = self.master.processed\_image

        self.copy=self.original\_image

        self.copy1=cv2.cvtColor(self.original\_image, cv2.COLOR\_BGR2HLS)

        self.processing\_image = self.master.processed\_image

        self.contrast\_label = Label(self, text="Contrast")

        self.contrast\_scale = Scale(self, from\_=0, to\_=255, length=250, resolution=1,

                                       orient=HORIZONTAL)

        self.brightness\_label = Label(self, text="Brightness")

        self.brightness\_scale = Scale(self, from\_=0, to\_=510, length=250, resolution=1,

                                      orient=HORIZONTAL)

        # self.saturation\_label = Label(self, text="Saturation")

        # self.saturation\_scale = Scale(self, from\_=0, to\_=2, length=250, resolution=0.1,

        #                               orient=HORIZONTAL)

        self.clarity\_label = Label(self, text="Blur")

        self.clarity\_scale = Scale(self, from\_=0, to\_=2, length=250, resolution=0.1,

                                    orient=HORIZONTAL)

        self.warmth\_label = Label(self, text="Warmth")

        self.warmth\_scale = Scale(self, from\_=0, to\_=1, length=250, resolution=0.05,

                                   orient=HORIZONTAL)

        self.cool\_label = Label(self, text="Cool")

        self.cool\_scale = Scale(self, from\_=0, to\_=1, length=250, resolution=0.05,

                                  orient=HORIZONTAL)

        self.r\_label = Label(self, text="R")

        self.r\_scale = Scale(self, from\_=-100, to\_=100, length=250, resolution=1,

                             orient=HORIZONTAL)

        self.g\_label = Label(self, text="G")

        self.g\_scale = Scale(self, from\_=-100, to\_=100, length=250, resolution=1,

                             orient=HORIZONTAL)

        self.b\_label = Label(self, text="B")

        self.b\_scale = Scale(self, from\_=-100, to\_=100, length=250, resolution=1,

                             orient=HORIZONTAL)

        self.apply\_button = Button(self, text="Apply")

        self.preview\_button = Button(self, text="Preview")

        self.cancel\_button = Button(self, text="Cancel")

        self.brightness\_scale.set(255)

        self.contrast\_scale.set(127)

        self.warmth\_scale.set(0)

        self.cool\_scale.set(0)

        # self.saturation\_scale.set(1)

        self.clarity\_scale.set(0)

        self.apply\_button.bind("<ButtonRelease>", self.apply\_button\_released)

        self.preview\_button.bind("<ButtonRelease>", self.show\_button\_release)

        self.cancel\_button.bind("<ButtonRelease>", self.cancel\_button\_released)

        self.brightness\_label.pack()

        self.brightness\_scale.pack()

        self.warmth\_label.pack()

        self.warmth\_scale.pack()

        self.cool\_label.pack()

        self.cool\_scale.pack()

        self.clarity\_label.pack()

        self.clarity\_scale.pack()

        self.contrast\_label.pack()

        self.contrast\_scale.pack()

        self.r\_label.pack()

        self.r\_scale.pack()

        self.g\_label.pack()

        self.g\_scale.pack()

        self.b\_label.pack()

        self.b\_scale.pack()

        self.cancel\_button.pack(side=RIGHT)

        self.preview\_button.pack(side=RIGHT)

        self.apply\_button.pack()

    def apply\_button\_released(self, event):

        self.show\_button\_release(self)

        self.master.processed\_image = self.processing\_image

        self.close()

    def gamma\_function(self,channel, gamma):

        invGamma = 1 / gamma

        table = np.array([((i / 255.0) \*\* invGamma) \* 255

                          for i in np.arange(0, 256)]).astype("uint8")  # creating lookup table

        channel = cv2.LUT(channel, table)

        return channel

    def show\_button\_release(self, event):

        temp = self.copy.copy

        self.show\_image(self.copy)

        self.original\_image = self.copy

        b, g, r = cv2.split(self.processing\_image)

        for b\_value in b:

            cv2.add(b\_value, self.b\_scale.get(), b\_value)

        for g\_value in g:

            cv2.add(g\_value, self.g\_scale.get(), g\_value)

        for r\_value in r:

            cv2.add(r\_value, self.r\_scale.get(), r\_value)

        self.rgb = cv2.merge((b, g, r))

        brightness = int((self.brightness\_scale.get() - 0) \* (255 - (-255)) / (510 - 0) + (-255))

        contrast = int((self.contrast\_scale.get() - 0) \* (127 - (-127)) / (254 - 0) + (-127))

        if brightness != 0:

            if brightness > 0:

                shadow = brightness

                max = 255

            else:

                shadow = 0

                max = 255 + brightness

            al\_pha = (max - shadow) / 255

            ga\_mma = shadow

            # The function addWeighted calculates

            # the weighted sum of two arrays

            cal = cv2.addWeighted(self.rgb, al\_pha,

                                  self.rgb, 0, ga\_mma)

        else:

            cal = self.rgb

        if contrast != 0:

            Alpha = float(131 \* (contrast + 127)) / (127 \* (131 - contrast))

            Gamma = 127 \* (1 - Alpha)

            # The function addWeighted calculates

            # the weighted sum of two arrays

            cal = cv2.addWeighted(cal, Alpha,

                                  cal, 0, Gamma)

        clar = self.clarity\_scale.get()

        print(clar)

        if clar!=0:

            clar = (int)(clar \* 10)

            img = cv2.blur(cal, (clar, clar))

            self.processing\_image = img

            # cv2.imshow("test", img)

            # cv2.waitKey(0)

        else:

            self.processing\_image = cal

        warmth=self.warmth\_scale.get()

        warmth/=2

        img = self.processing\_image

        img[:, :, 0] = self.gamma\_function(img[:, :, 0], 1-warmth)  # down scaling blue channel

        img[:, :, 2] = self.gamma\_function(img[:, :, 2], 1+warmth)  # up scaling red channel

        hsv = cv2.cvtColor(img, cv2.COLOR\_BGR2HSV)

        hsv[:, :, 1] = self.gamma\_function(hsv[:, :, 1], 1+warmth-0.01)  # up scaling saturation channel

        img = cv2.cvtColor(hsv, cv2.COLOR\_HSV2BGR)

        self.processing\_image = img

        cool = self.cool\_scale.get()

        cool /= 2

        img1 = self.processing\_image

        img1[:, :, 0] = self.gamma\_function(img1[:, :, 0], 1 + cool)  # down scaling blue channel

        img1[:, :, 2] = self.gamma\_function(img1[:, :, 2], 1 - cool)  # up scaling red channel

        hsv1 = cv2.cvtColor(img1, cv2.COLOR\_BGR2HSV)

        hsv1[:, :, 1] = self.gamma\_function(hsv1[:, :, 1], 1 - cool+0.01)  # up scaling saturation channel

        img1 = cv2.cvtColor(hsv1, cv2.COLOR\_HSV2BGR)

        self.processing\_image = img1

        self.original\_image=temp

        # self.processing\_image=cv2.addWeighted(self.processing\_image,0.3,self.rgb,0.7,0.0)

        # self.processing\_image = cv2.addWeighted(self.processing\_image, 0.3, img, 0.7, 0.0)

        self.show\_image(self.processing\_image)

        # self.destroy()

    def cancel\_button\_released(self, event):

        self.close()

    def show\_image(self, img=None):

        self.master.image\_viewer.show\_image(img=img)

    def close(self):

        self.show\_image()

        self.destroy()

**editBar.py:**

from tkinter import Frame, Button, LEFT

from tkinter import filedialog

from filterFrame import FilterFrame

from adjustFrame import AdjustFrame

import cv2

class EditBar(Frame):

    def \_\_init\_\_(self, master=None):

        Frame.\_\_init\_\_(self, master=master)

        self.new\_button = Button(self, text="New")

        self.save\_button = Button(self, text="Save")

        self.save\_as\_button = Button(self, text="Save As")

        self.draw\_button = Button(self, text="Draw")

        self.crop\_button = Button(self, text="Crop")

        self.mirror\_button = Button(self, text="Mirror")

        self.rotate\_button = Button(self, text="Rotate")

        self.filter\_button = Button(self, text="Filter")

        self.adjust\_button = Button(self, text="Adjust")

        self.clear\_button = Button(self, text="Clear")

        self.new\_button.bind("<ButtonRelease>", self.new\_button\_released)

        self.save\_button.bind("<ButtonRelease>", self.save\_button\_released)

        self.save\_as\_button.bind("<ButtonRelease>", self.save\_as\_button\_released)

        self.draw\_button.bind("<ButtonRelease>", self.draw\_button\_released)

        self.crop\_button.bind("<ButtonRelease>", self.crop\_button\_released)

        self.mirror\_button.bind("<ButtonRelease>", self.mirror\_button\_released)

        self.rotate\_button.bind("<ButtonRelease>", self.rotate\_button\_released)

        self.filter\_button.bind("<ButtonRelease>", self.filter\_button\_released)

        self.adjust\_button.bind("<ButtonRelease>", self.adjust\_button\_released)

        self.clear\_button.bind("<ButtonRelease>", self.clear\_button\_released)

        self.new\_button.pack(side=LEFT)

        self.save\_button.pack(side=LEFT)

        self.save\_as\_button.pack(side=LEFT)

        self.draw\_button.pack(side=LEFT)

        self.crop\_button.pack(side=LEFT)

        self.filter\_button.pack(side=LEFT)

        self.adjust\_button.pack(side=LEFT)

        self.mirror\_button.pack(side=LEFT)

        self.rotate\_button.pack(side=LEFT)

        self.clear\_button.pack()

    def rotate\_button\_released(self, event):

        if self.winfo\_containing(event.x\_root, event.y\_root) == self.rotate\_button:

            if self.master.is\_image\_selected:

                if self.master.is\_draw\_state:

                    self.master.image\_viewer.deactivate\_draw()

                if self.master.is\_crop\_state:

                    self.master.image\_viewer.deactivate\_crop()

                self.master.processed\_image = cv2.rotate(self.master.original\_image, cv2.ROTATE\_90\_CLOCKWISE)

                self.master.image\_viewer.show\_image()

                self.master.original\_image = self.master.processed\_image

    def mirror\_button\_released(self, event):

        if self.winfo\_containing(event.x\_root, event.y\_root) == self.mirror\_button:

            if self.master.is\_image\_selected:

                if self.master.is\_draw\_state:

                    self.master.image\_viewer.deactivate\_draw()

                if self.master.is\_crop\_state:

                    self.master.image\_viewer.deactivate\_crop()

                self.master.processed\_image = cv2.flip(self.master.processed\_image, 1)

                self.master.image\_viewer.show\_image()

                self.master.original\_image = self.master.processed\_image

    def new\_button\_released(self, event):

        if self.winfo\_containing(event.x\_root, event.y\_root) == self.new\_button:

            if self.master.is\_draw\_state:

                self.master.image\_viewer.deactivate\_draw()

            if self.master.is\_crop\_state:

                self.master.image\_viewer.deactivate\_crop()

            filename = filedialog.askopenfilename()

            image = cv2.imread(filename)

            if image is not None:

                self.master.filename = filename

                self.master.original\_image = image.copy()

                self.master.processed\_image = image.copy()

                self.master.image\_viewer.show\_image()

                self.master.is\_image\_selected = True

    def save\_button\_released(self, event):

        if self.winfo\_containing(event.x\_root, event.y\_root) == self.save\_button:

            if self.master.is\_image\_selected:

                if self.master.is\_crop\_state:

                    self.master.image\_viewer.deactivate\_crop()

                if self.master.is\_draw\_state:

                    self.master.image\_viewer.deactivate\_draw()

                save\_image = self.master.processed\_image

                image\_filename = self.master.filename

                cv2.imwrite(image\_filename, save\_image)

    def save\_as\_button\_released(self, event):

        if self.winfo\_containing(event.x\_root, event.y\_root) == self.save\_as\_button:

            if self.master.is\_image\_selected:

                if self.master.is\_draw\_state:

                    self.master.image\_viewer.deactivate\_draw()

                if self.master.is\_crop\_state:

                    self.master.image\_viewer.deactivate\_crop()

                original\_file\_type = self.master.filename.split('.')[-1]

                filename = filedialog.asksaveasfilename()

                filename = filename + "." + original\_file\_type

                save\_image = self.master.processed\_image

                cv2.imwrite(filename, save\_image)

                self.master.filename = filename

    def draw\_button\_released(self, event):

        if self.winfo\_containing(event.x\_root, event.y\_root) == self.draw\_button:

            if self.master.is\_image\_selected:

                if self.master.is\_crop\_state:

                    self.master.image\_viewer.deactivate\_crop()

                if self.master.is\_draw\_state:

                    self.master.image\_viewer.deactivate\_draw()

                else:

                    self.master.image\_viewer.activate\_draw()

    def crop\_button\_released(self, event):

        if self.winfo\_containing(event.x\_root, event.y\_root) == self.crop\_button:

            if self.master.is\_image\_selected:

                if self.master.is\_draw\_state:

                    self.master.image\_viewer.deactivate\_draw()

                if self.master.is\_crop\_state:

                    self.master.image\_viewer.deactivate\_crop()

                else:

                    self.master.image\_viewer.activate\_crop()

    def filter\_button\_released(self, event):

        if self.winfo\_containing(event.x\_root, event.y\_root) == self.filter\_button:

            if self.master.is\_image\_selected:

                if self.master.is\_draw\_state:

                    self.master.image\_viewer.deactivate\_draw()

                if self.master.is\_crop\_state:

                    self.master.image\_viewer.deactivate\_crop()

                self.master.filter\_frame = FilterFrame(master=self.master)

                self.master.filter\_frame.grab\_set()

    def adjust\_button\_released(self, event):

        self.master.processed\_image = self.master.original\_image.copy()

        self.master.image\_viewer.show\_image()

        if self.winfo\_containing(event.x\_root, event.y\_root) == self.adjust\_button:

            if self.master.is\_image\_selected:

                if self.master.is\_draw\_state:

                    self.master.image\_viewer.deactivate\_draw()

                if self.master.is\_crop\_state:

                    self.master.image\_viewer.deactivate\_crop()

                self.master.adjust\_frame = AdjustFrame(master=self.master)

                self.master.adjust\_frame.grab\_set()

    def clear\_button\_released(self, event):

        if self.winfo\_containing(event.x\_root, event.y\_root) == self.clear\_button:

            if self.master.is\_image\_selected:

                if self.master.is\_draw\_state:

                    self.master.image\_viewer.deactivate\_draw()

                if self.master.is\_crop\_state:

                    self.master.image\_viewer.deactivate\_crop()

                self.master.processed\_image = self.master.original\_image.copy()

                self.master.image\_viewer.show\_image()

**filterFrame.py:**

from tkinter import Toplevel, Button, RIGHT

import numpy as np

import cv2

class FilterFrame(Toplevel):

    def \_\_init\_\_(self, master=None):

        Toplevel.\_\_init\_\_(self, master=master)

        self.original\_image = self.master.processed\_image

        self.filtered\_image = None

        self.negative\_button = Button(master=self, text="Negative")

        self.black\_white\_button = Button(master=self, text="Black White")

        self.sepia\_button = Button(master=self, text="Sepia")

        self.emboss\_button = Button(master=self, text="Emboss")

        self.gaussian\_blur\_button = Button(master=self, text="Gaussian Blur")

        self.median\_blur\_button = Button(master=self, text="Median Blur")

        self.details\_button = Button(master=self, text="Details")

        self.summer\_button = Button(master=self, text="Summer")

        self.winter\_button = Button(master=self, text="Winter")

        self.daylight\_button = Button(master=self, text="DayLight")

        self.grainy\_button = Button(master=self, text="Grainy")

        self.highcontrast\_button=Button(master=self,text="High Contrast")

        self.smoothen\_button = Button(master=self, text="Smoothen")

        self.vignette\_button = Button(master=self, text="Vignette")

        self.bonus\_button = Button(master=self, text="Bonus")

        self.distorted\_button = Button(master=self, text="Distorted")

        self.cancel\_button = Button(master=self, text="Cancel")

        self.apply\_button = Button(master=self, text="Apply")

        self.details\_button=Button(master=self,text="Details")

        self.negative\_button.bind("<ButtonRelease>", self.negative\_button\_released)

        self.black\_white\_button.bind("<ButtonRelease>", self.black\_white\_released)

        self.sepia\_button.bind("<ButtonRelease>", self.sepia\_button\_released)

        self.emboss\_button.bind("<ButtonRelease>", self.emboss\_button\_released)

        self.gaussian\_blur\_button.bind("<ButtonRelease>", self.gaussian\_blur\_button\_released)

        self.median\_blur\_button.bind("<ButtonRelease>", self.median\_blur\_button\_released)

        self.details\_button.bind("<ButtonRelease>", self.details\_button\_released)

        self.summer\_button.bind("<ButtonRelease>", self.summer\_button\_released)

        self.winter\_button.bind("<ButtonRelease>", self.winter\_button\_released)

        self.daylight\_button.bind("<ButtonRelease>", self.daylight\_button\_released)

        self.grainy\_button.bind("<ButtonRelease>", self.grainy\_button\_released)

        self.smoothen\_button.bind("<ButtonRelease>", self.smoothen\_button\_released)

        self.highcontrast\_button.bind("<ButtonRelease>", self.highcontrast\_button\_released)

        self.distorted\_button.bind("<ButtonRelease>", self.distorted\_button\_released)

        self.vignette\_button.bind("<ButtonRelease>", self.vignette\_button\_released)

        self.bonus\_button.bind("<ButtonRelease>", self.bonus\_button\_released)

        self.apply\_button.bind("<ButtonRelease>", self.apply\_button\_released)

        self.cancel\_button.bind("<ButtonRelease>", self.cancel\_button\_released)

        self.negative\_button.pack()

        self.black\_white\_button.pack()

        self.sepia\_button.pack()

        self.emboss\_button.pack()

        self.gaussian\_blur\_button.pack()

        self.median\_blur\_button.pack()

        self.details\_button.pack()

        self.summer\_button.pack()

        self.winter\_button.pack()

        self.distorted\_button.pack()

        self.daylight\_button.pack()

        self.grainy\_button.pack()

        self.smoothen\_button.pack()

        self.highcontrast\_button.pack()

        self.vignette\_button.pack()

        self.bonus\_button.pack()

        self.cancel\_button.pack(side=RIGHT)

        self.apply\_button.pack()

    def details\_button\_released(self, event):

        self.details()

        self.show\_image()

    def bonus\_button\_released(self, event):

        self.bonus()

        self.show\_image()

    def distorted\_button\_released(self, event):

        self.distorted()

        self.show\_image()

    def highcontrast\_button\_released(self, event):

        self.highcontrast()

        self.show\_image()

    def summer\_button\_released(self, event):

        self.summer()

        self.show\_image()

    def winter\_button\_released(self, event):

        self.winter()

        self.show\_image()

    def daylight\_button\_released(self, event):

        self.daylight()

        self.show\_image()

    def grainy\_button\_released(self, event):

        self.grainy()

        self.show\_image()

    def smoothen\_button\_released(self, event):

        self.smoothen()

        self.show\_image()

    def vignette\_button\_released(self, event):

        self.vignette()

        self.show\_image()

    # dst = cv2.stylization(self.original\_image, sigma\_s=60, sigma\_r=0.07)

    # dst\_gray, dst\_color = cv2.pencilSketch(self.original\_image, sigma\_s=60, sigma\_r=0.07, shade\_factor=0.05)

    def details(self):

        # sigma\_s controls how much the image is smoothed - the larger its value,

        # the more smoothed the image gets, but it's also slower to compute.

        # sigma\_r is important if you want to preserve edges while smoothing the image.

        # Small sigma\_r results in only very similar colors to be averaged (i.e. smoothed), while colors that differ much will stay intact.

        kernel\_sharpening = np.array([[-1, -1, -1],

                                      [-1, 9, -1],

                                      [-1, -1, -1]])

        dst2 = cv2.filter2D(self.original\_image, -1, kernel\_sharpening)

        self.filtered\_image=dst2

    def gamma\_function(self,channel, gamma):

        invGamma = 1 / gamma

        table = np.array([((i / 255.0) \*\* invGamma) \* 255

                          for i in np.arange(0, 256)]).astype("uint8")  # creating lookup table

        channel = cv2.LUT(channel, table)

        return channel

    def summer(self):

        img = self.original\_image

        img[:, :, 0] = self.gamma\_function(img[:, :, 0], 0.75)  # down scaling blue channel

        img[:, :, 2] = self.gamma\_function(img[:, :, 2], 1.25)  # up scaling red channel

        hsv = cv2.cvtColor(img, cv2.COLOR\_BGR2HSV)

        hsv[:, :, 1] = self.gamma\_function(hsv[:, :, 1], 1.2)  # up scaling saturation channel

        img = cv2.cvtColor(hsv, cv2.COLOR\_HSV2BGR)

        self.filtered\_image=img

    def winter(self):

        img = self.original\_image

        img[:, :, 0] = self.gamma\_function(img[:, :, 0], 1.25)  # down scaling blue channel

        img[:, :, 2] = self.gamma\_function(img[:, :, 2], 0.75)  # up scaling red channel

        hsv = cv2.cvtColor(img, cv2.COLOR\_BGR2HSV)

        hsv[:, :, 1] = self.gamma\_function(hsv[:, :, 1], 0.8)  # up scaling saturation channel

        img = cv2.cvtColor(hsv, cv2.COLOR\_HSV2BGR)

        self.filtered\_image=img

    def vignette(self):

        rows, cols = self.original\_image.shape[:2]

        # generating vignette mask using Gaussian

        # resultant\_kernels

        X\_resultant\_kernel = cv2.getGaussianKernel(cols, 200)

        Y\_resultant\_kernel = cv2.getGaussianKernel(rows, 200)

        # generating resultant\_kernel matrix

        resultant\_kernel = Y\_resultant\_kernel \* X\_resultant\_kernel.T

        # creating mask and normalising by using np.linalg

        # function

        mask = 255 \* resultant\_kernel / np.linalg.norm(resultant\_kernel)

        output = np.copy(self.original\_image)

        # applying the mask to each channel in the input image

        for i in range(3):

            output[:, :, i] = output[:, :, i] \* mask

        self.filtered\_image=output

    def smoothen(self):

        dst2 = cv2.edgePreservingFilter(self.original\_image, flags=1, sigma\_s=60, sigma\_r=0.4)

        self.filtered\_image = dst2

    def bonus(self):

        dst = cv2.stylization(self.original\_image, sigma\_s=60, sigma\_r=0.07)

        self.filtered\_image = dst

    def distorted(self):

        dst\_gray, dst\_color = cv2.pencilSketch(self.original\_image, sigma\_s=60, sigma\_r=0.07, shade\_factor=0.05)

        self.filtered\_image = dst\_color

    def daylight(self):

        img = self.original\_image

        image\_HLS = cv2.cvtColor(img, cv2.COLOR\_BGR2HLS)  # Conversion to HLS

        image\_HLS = np.array(image\_HLS, dtype=np.float64)

        daylight = 1.15

        image\_HLS[:, :, 1] = image\_HLS[:, :, 1] \* daylight  # scale pixel values up for channel 1(Lightness)

        image\_HLS[:, :, 1][image\_HLS[:, :, 1] > 255] = 255  # Sets all values above 255 to 255

        image\_HLS = np.array(image\_HLS, dtype=np.uint8)

        image\_RGB = cv2.cvtColor(image\_HLS, cv2.COLOR\_HLS2BGR)

        self.filtered\_image=image\_RGB

    def grainy(self):

        img = self.original\_image

        height, width = img.shape[:2]

        gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

        thresh = 0.8  # creating threshold. This means noise will be added to 80% pixels

        for i in range(height):

            for j in range(width):

                if np.random.rand() <= thresh:

                    if np.random.randint(2) == 0:

                        gray[i, j] = min(gray[i, j] + np.random.randint(0, 64),

                                         255)  # adding random value between 0 to 64. Anything above 255 is set to 255.

                    else:

                        gray[i, j] = max(gray[i, j] - np.random.randint(0, 64),

                                         0)  # subtracting random values between 0 to 64. Anything below 0 is set to 0.

        self.filtered\_image=gray

    def highcontrast(self):

        img = self.original\_image

        gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

        xp = [0, 64, 112, 128, 144, 192, 255]  # setting reference values

        fp = [0, 16, 64, 128, 192, 240, 255]  # setting values to be taken for reference values

        x = np.arange(256)

        table = np.interp(x, xp, fp).astype('uint8')  # creating lookup table

        img = cv2.LUT(gray, table)  # changing values based on lookup table

        self.filtered\_image = img

    def negative\_button\_released(self, event):

        self.negative()

        self.show\_image()

    def black\_white\_released(self, event):

        self.black\_white()

        self.show\_image()

    def sepia\_button\_released(self, event):

        self.sepia()

        self.show\_image()

    def emboss\_button\_released(self, event):

        self.emboss()

        self.show\_image()

    def gaussian\_blur\_button\_released(self, event):

        self.gaussian\_blur()

        self.show\_image()

    def median\_blur\_button\_released(self, event):

        self.gaussian\_blur()

        self.show\_image()

    def apply\_button\_released(self, event):

        self.master.processed\_image = self.filtered\_image

        self.show\_image()

        self.close()

    def cancel\_button\_released(self, event):

        self.master.image\_viewer.show\_image()

        self.close()

    def show\_image(self):

        self.master.image\_viewer.show\_image(img=self.filtered\_image)

    def negative(self):

        self.filtered\_image = cv2.bitwise\_not(self.original\_image)

    def black\_white(self):

        self.filtered\_image = cv2.cvtColor(self.original\_image, cv2.COLOR\_BGR2GRAY)

        self.filtered\_image = cv2.cvtColor(self.filtered\_image, cv2.COLOR\_GRAY2BGR)

    def sepia(self):

        kernel = np.array([[0.272, 0.534, 0.131],

                           [0.349, 0.686, 0.168],

                           [0.393, 0.769, 0.189]])

        self.filtered\_image = cv2.filter2D(self.original\_image, -1, kernel)

    def emboss(self):

        kernel = np.array([[0, -1, -1],

                           [1, 0, -1],

                           [1, 1, 0]])

        self.filtered\_image = cv2.filter2D(self.original\_image, -1, kernel)

    def gaussian\_blur(self):

        self.filtered\_image = cv2.GaussianBlur(self.original\_image, (41, 41), 0)

    def median\_blur(self):

        self.filtered\_image = cv2.medianBlur(self.original\_image, 41)

    def close(self):

        self.destroy()

**imageViewer.py:**

from tkinter import Frame, Canvas, CENTER, ROUND

from pil import Image, ImageTk

import cv2

class ImageViewer(Frame):

    def \_\_init\_\_(self, master=None):

        Frame.\_\_init\_\_(self, master=master, bg="gray", width=600, height=400)

        self.shown\_image = None

        self.x = 0

        self.y = 0

        self.crop\_start\_x = 0

        self.crop\_start\_y = 0

        self.crop\_end\_x = 0

        self.crop\_end\_y = 0

        self.draw\_ids = list()

        self.rectangle\_id = 0

        self.ratio = 0

        self.canvas = Canvas(self, bg="gray", width=600, height=400)

        self.canvas.place(relx=0.5, rely=0.5, anchor=CENTER)

    def show\_image(self, img=None):

        self.clear\_canvas()

        if img is None:

            image = self.master.processed\_image.copy()

        else:

            image = img

        image = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

        height, width, channels = image.shape

        ratio = height / width

        new\_width = width

        new\_height = height

        if height > self.winfo\_height() or width > self.winfo\_width():

            if ratio < 1:

                new\_width = self.winfo\_width()

                new\_height = int(new\_width \* ratio)

            else:

                new\_height = self.winfo\_height()

                new\_width = int(new\_height \* (width / height))

        self.shown\_image = cv2.resize(image, (new\_width, new\_height))

        self.shown\_image = ImageTk.PhotoImage(Image.fromarray(self.shown\_image))

        self.ratio = height / new\_height

        self.canvas.config(width=new\_width, height=new\_height)

        self.canvas.create\_image(new\_width / 2, new\_height / 2, anchor=CENTER, image=self.shown\_image)

    def activate\_draw(self):

        self.canvas.bind("<ButtonPress>", self.start\_draw)

        self.canvas.bind("<B1-Motion>", self.draw)

        self.master.is\_draw\_state = True

    def activate\_crop(self):

        self.canvas.bind("<ButtonPress>", self.start\_crop)

        self.canvas.bind("<B1-Motion>", self.crop)

        self.canvas.bind("<ButtonRelease>", self.end\_crop)

        self.master.is\_crop\_state = True

    def activate\_mirror(self):

        self.canvas.bind("<ButtonPress>", self.start\_mirror())

        self.canvas.bind("<B1-Motion>", self.mirror)

        self.canvas.bind("<ButtonRelease>", self.end\_mirror)

        self.master.is\_crop\_state = True

    def deactivate\_draw(self):

        self.canvas.unbind("<ButtonPress>")

        self.canvas.unbind("<B1-Motion>")

        self.master.is\_draw\_state = False

    def deactivate\_crop(self):

        self.canvas.unbind("<ButtonPress>")

        self.canvas.unbind("<B1-Motion>")

        self.canvas.unbind("<ButtonRelease>")

        self.master.is\_crop\_state = False

    def start\_draw(self, event):

        self.x = event.x

        self.y = event.y

    def draw(self, event):

        self.draw\_ids.append(self.canvas.create\_line(self.x, self.y, event.x, event.y, width=2,

                                                     fill="red", capstyle=ROUND, smooth=True))

        cv2.line(self.master.processed\_image, (int(self.x \* self.ratio), int(self.y \* self.ratio)),

                 (int(event.x \* self.ratio), int(event.y \* self.ratio)),

                 (0, 0, 255), thickness=int(self.ratio \* 2),

                 lineType=8)

        self.x = event.x

        self.y = event.y

    def start\_rotate(self, event):

        self.x = event.x

        self.y = event.y

    def start\_mirror(self, event):

        self.x = event.x

        self.y = event.y

    def start\_crop(self, event):

        self.crop\_start\_x = event.x

        self.crop\_start\_y = event.y

    def crop(self, event):

        if self.rectangle\_id:

            self.canvas.delete(self.rectangle\_id)

        self.crop\_end\_x = event.x

        self.crop\_end\_y = event.y

        self.rectangle\_id = self.canvas.create\_rectangle(self.crop\_start\_x, self.crop\_start\_y,

                                                         self.crop\_end\_x, self.crop\_end\_y, width=1)

    def end\_crop(self, event):

        if self.crop\_start\_x <= self.crop\_end\_x and self.crop\_start\_y <= self.crop\_end\_y:

            start\_x = int(self.crop\_start\_x \* self.ratio)

            start\_y = int(self.crop\_start\_y \* self.ratio)

            end\_x = int(self.crop\_end\_x \* self.ratio)

            end\_y = int(self.crop\_end\_y \* self.ratio)

        elif self.crop\_start\_x > self.crop\_end\_x and self.crop\_start\_y <= self.crop\_end\_y:

            start\_x = int(self.crop\_end\_x \* self.ratio)

            start\_y = int(self.crop\_start\_y \* self.ratio)

            end\_x = int(self.crop\_start\_x \* self.ratio)

            end\_y = int(self.crop\_end\_y \* self.ratio)

        elif self.crop\_start\_x <= self.crop\_end\_x and self.crop\_start\_y > self.crop\_end\_y:

            start\_x = int(self.crop\_start\_x \* self.ratio)

            start\_y = int(self.crop\_end\_y \* self.ratio)

            end\_x = int(self.crop\_end\_x \* self.ratio)

            end\_y = int(self.crop\_start\_y \* self.ratio)

        else:

            start\_x = int(self.crop\_end\_x \* self.ratio)

            start\_y = int(self.crop\_end\_y \* self.ratio)

            end\_x = int(self.crop\_start\_x \* self.ratio)

            end\_y = int(self.crop\_start\_y \* self.ratio)

        x = slice(start\_x, end\_x, 1)

        y = slice(start\_y, end\_y, 1)

        self.master.processed\_image = self.master.processed\_image[y, x]

        self.show\_image()

    def clear\_canvas(self):

        self.canvas.delete("all")

    def clear\_draw(self):

        self.canvas.delete(self.draw\_ids)

**init.py:**

from main import Main

root = Main()

root.mainloop()

**main.py:**

import tkinter as tk

from tkinter import ttk

from editBar import EditBar

from imageViewer import ImageViewer

class Main(tk.Tk):

    def \_\_init\_\_(self):

        tk.Tk.\_\_init\_\_(self)

        self.filename = ""

        self.original\_image = None

        self.processed\_image = None

        self.is\_image\_selected = False

        self.is\_draw\_state = False

        self.is\_crop\_state = False

        self.is\_mirror\_state = False

        self.is\_rotate\_state = False

        self.filter\_frame = None

        self.adjust\_frame = None

        self.title("Image Editor")

        self.editbar = EditBar(master=self)

        separator1 = ttk.Separator(master=self, orient=tk.HORIZONTAL)

        self.image\_viewer = ImageViewer(master=self)

        self.editbar.pack(pady=10)

        separator1.pack(fill=tk.X, padx=20, pady=5)

        self.image\_viewer.pack(fill=tk.BOTH, padx=20, pady=10, expand=1)

**setup.py:**

from setuptools import setup

setup(

    app=["init.py"],

    setup\_requires=["py2app"]

)