```
#
# AIMER: Artificial Intelligence Mark Evaluator & Recognizer
# Unpublished Copyright (C) 2022  Fereydoun Memarzanjany ("AUTHOR"), All Rights Reserved.
#
# NOTICE: All information contained herein is, and remains the property of AUTHOR.  The intellectual and technical concepts contained herein are
# proprietary to AUTHOR and may be covered by U.S. and Foreign Patents, patents in process, and are protected by trade secret or copyright law.
# Dissemination of this information or reproduction of this material is strictly forbidden unless prior written permission is obtained from AUTHOR.
# Access to the source code contained herein is hereby forbidden to anyone except current AUTHOR employees, managers or contractors who have executed
# Confidentiality and Non-disclosure agreements explicitly covering such access.
#
# The copyright notice above does not evidence any actual or intended publication or disclosure of this source code, which includes information
# that is confidential and/or proprietary, and is a trade secret, of AUTHOR.  ANY REPRODUCTION, MODIFICATION, DISTRIBUTION, PUBLIC PERFORMANCE,
# OR PUBLIC DISPLAY OF OR THROUGH USE OF THIS SOURCE CODE WITHOUT THE EXPRESS WRITTEN CONSENT OF AUTHOR IS STRICTLY PROHIBITED, AND IN VIOLATION OF
# APPLICABLE LAWS AND INTERNATIONAL TREATIES.  THE RECEIPT OR POSSESSION OF THIS SOURCE CODE AND/OR RELATED INFORMATION DOES NOT CONVEY OR IMPLY ANY
# RIGHTS TO REPRODUCE, DISCLOSE OR DISTRIBUTE ITS CONTENTS, OR TO MANUFACTURE, USE, OR SELL ANYTHING THAT IT MAY DESCRIBE, IN WHOLE OR IN PART.
#

```

```
numpy==1.22.2
opencv-python==4.5.5.62
opencv-contrib-python==4.5.5.62

```

```
@echo off

echo "Installing dependencies..."

python -m pip install --upgrade -r requirements.txt

```

```python
import logging
import time
from utils import camera_driver, scanner, grader
import cv2 as cv
import numpy as np

# This sets the root logger to write to stdout.  Also, by default the
# logger is set to print (only) WARNINGs, we want INFOs to get printed as
well.
logging.basicConfig(encoding="utf-8", level=logging.NOTSET)

cap = camera_driver.init_camera()


def main():
    cv.namedWindow("webcam")
    cv.namedWindow("graded")

    img = None
    scanned = None

    #image = cv.imread("test.png", cv.IMREAD_GRAYSCALE)

    while True:
        _ret, img = cap.read()

        img, scanned_img = scanner.scan(img)
        cv.imshow("webscam", img)

        score, graded_img = grader.grade(scanned_img)
        cv.imshow("graded", graded_img)

        if (cv.waitKey(33) == ord('s')):
            break

    print("The final grade was: " + str(score) + "%")

cv.destroyAllWindows()

# Main harness function for driving the code.
if (__name__ == "__main__"): # The module is being run directly
    main()
else: # The module is being imported into another one
    pass
```

```
import logging
import time
import cv2 as cv

# TODO: Move these to a centralized configuration section or file.
#
CAMERA_SOURCE = int(0) # 0 is the default camera device.
# Most cameras are at least 720p (1280x720 pixels) and have a frame rate of
30hz.
CAMERA_WIDTH = float(1280)
CAMERA_HEIGHT = float(720)
CAMERA_FPS = float(30)
VIDEO_BACKEND = cv.CAP_DSHOW # TODO: Directshow video backend is not portable
outside of Windows.
# The string below is case insensitive
VIDEO_CODEC = "mjpg" # MJPG is widely supported, even though H.265 is better.


# Initializes a camera video stream and configures it for the best possible
performance.
#
# TODO: Class interfaces, are suited better for this camera driver.  Use them
later.
def init_camera():
    logging.info("Attempting to open the default camera device at index `" +
str(CAMERA_SOURCE) + "`...")
    # Captures a video stream.
    cap = cv.VideoCapture(CAMERA_SOURCE, VIDEO_BACKEND)
    #time.sleep(2.0) # A 2-seconds "grace period" for allowing the camera to
finish its setup.

    if (cap is None) or (not cap.isOpened()): # Check if the camera capture
was successful or not.
        logging.error("Unable to open the specified video source, returning
early.")
        return None
    else:
        logging.info("The specified camera device has successfully been
opened.")


    # NOTE: Automatically determining the maximum camera resolution is not
currently possible with Directshow.
    #
    #max_width = max_height = max_fps = float(0)
    #
    #default_fps =  float(cap.get(cv.CAP_PROP_FPS))
    #if (cap.set(cv.CAP_PROP_FPS, HIGH_VALUE)):
    #    max_fps = float(cap.get(cv.CAP_PROP_FPS))
    #    cap.set(cv.CAP_PROP_FPS, default_fps)
    #
    #default_width = float(cap.get(cv.CAP_PROP_FRAME_WIDTH))
    #if (cap.set(cv.CAP_PROP_FRAME_WIDTH, HIGH_VALUE)):
```

```
    #    max_width = float(cap.get(cv.CAP_PROP_FRAME_WIDTH))
    #    cap.set(cv.CAP_PROP_FRAME_WIDTH, default_width)
    #
    #default_height = float(cap.get(cv.CAP_PROP_FRAME_HEIGHT))
    #if (cap.set(cv.CAP_PROP_FRAME_HEIGHT, HIGH_VALUE)):
    #    max_height = float(cap.get(cv.CAP_PROP_FRAME_HEIGHT))
    #    cap.set(cv.CAP_PROP_FRAME_HEIGHT, default_height)
    #
    ## Restart the camera.
    #cap.release; del cap
    #cap = cv.VideoCapture(CAMERA_SOURCE, VIDEO_BACKEND)


    # TODO: Try-expect error handling.  Use them later.

    cap.set(cv.CAP_PROP_FPS, CAMERA_FPS)
    # TODO: Investigate why this _has_ to be called twice with both lower and upper-case letters.
    cap.set(cv.CAP_PROP_FOURCC, cv.VideoWriter.fourcc( *(VIDEO_CODEC.lower())) )
    cap.set(cv.CAP_PROP_FOURCC, cv.VideoWriter.fourcc( *(VIDEO_CODEC.upper())) )
    cap.set(cv.CAP_PROP_FRAME_WIDTH, CAMERA_WIDTH)
    cap.set(cv.CAP_PROP_FRAME_HEIGHT, CAMERA_HEIGHT)

    # NOTE: Directly capturing a video stream in grayscale mode is currently only possible on the cv.CAP_V4L backend.
    #
    #cap.set(cv.CAP_PROP_MODE, 2) # cv.CAP_MODE_GRAY = 2 = 0b10
    #cap.set(cv.CAP_PROP_CONVERT_RGB, 0)
    #cap.set(cv.CAP_PROP_FORMAT, cv.CV_8UC3)

    # Read a frame to double-check if the camera is still working fine after the configurations.
    _ret, _frame = cap.read()
    if (_ret == False) or (_frame is None): # The camera should not be returning an empty frame.
        logging.warning("Empty frames were read from the specified video source, continuing regardless.")
    else: # Everything went OK.  So we can safely notify the user of this and return afterwards.
        pass


    # TODO: Use another way (such as num_frames/elapsed_time) to programmatically get new_fps.
    new_fps = CAMERA_FPS # Should have been `int(cap.get(cv.CAP_PROP_FPS))` if it wasn't for an OpenCV bug.
    new_width = int(cap.get(cv.CAP_PROP_FRAME_WIDTH))
    new_height = int(cap.get(cv.CAP_PROP_FRAME_HEIGHT))

    # TODO: f-Strings with curly brackets {} for formatting the output string.  Use them later.
```

```
86      logging.info("Camera is set-up and ready with the configurations (Width x
    Height @ FPS): " + str(new_width) + "x" + str(new_height) + "@" +
    str(new_fps))
87
88      # And lastly, return the captured stream object.
89      return cap
```

```python
import cv2 as cv
import numpy as np

# TODO: Move these to a centralized configuration section or file.
#
NUM_CHOICES = int(4)
NUM_QUESTIONS = int(5)
# TODO: Make this more 'natural' and 1-based instead of 0-based.
ANSWERS_KEY = list([0, 2, 3, 1, 0])

# TODO: Merge these two functions.
#
def check_answers(answers_provided, answers_key):
    correct_answers = int(0)

    for answer in range(0, NUM_QUESTIONS):
        if answers_provided[answer] == answers_key[answer]:
            correct_answers += 1 # Award a single credit for the correct answer.
        else:
            correct_answers += 0 # No credits, pass.

    # Converts the student's score to a percentage.
    score = float( (correct_answers / NUM_QUESTIONS) * 100 )

    return score

def display_gradings_on_paper(score, answers_provided, answers_key, paper):
    # The following lines will draw a grid over the paper image.
    #
    # Properties of the supplied paper.
    paper_width = paper.shape[1]
    paper_height = paper.shape[0]

    # Properties of each cell in the overall grid.
    cell_width = (paper_width // NUM_CHOICES) # Height // Questions
    cell_height = (paper_height // NUM_QUESTIONS) # Width // Choices


    line_color = (255, 0, 0) # Blue
    line_thickness = 2

    # TODO: Maybe merge the two below for loops?
    # TODO: Also, this would overflow beyond the image.
    #
    for step in range(0, NUM_QUESTIONS*NUM_CHOICES):
        # Vertical lines
        start = (0, cell_height*step)
        end = (paper_width, cell_height*step)
        cv.line(paper, start, end, line_color, line_thickness)

        # Horizontal lines
        start = (cell_width*step, 0)
```

```
        end = (cell_width*step, paper_height)
        cv.line(paper, start, end, line_color, line_thickness)


    # The following lines will draw a circle over the correct answers provided by the key.
    #
    for step in range(0, NUM_QUESTIONS):
        current_cell_center_x = (ANSWERS_KEY[step] * cell_width) + (cell_width // 2)
        current_cell_center_y = (step * cell_height) + (cell_height // 2)
        current_cell_center = (current_cell_center_x, current_cell_center_y)

        bubble_size = 35
        correct_bubble_color = (0, 255, 0) # Green
        wrong_bubble_color = (0, 0, 255) # Red

        bubble_color = (0, 0, 0)
        if answers_provided[step] == ANSWERS_KEY[step]:
            bubble_color = correct_bubble_color
        else:
            bubble_color = wrong_bubble_color
        cv.circle(paper, current_cell_center, bubble_size, bubble_color, cv.FILLED)

    # The following lines will display the grade/score right on the paper's center.
    #
    # Text setup
    text = str(float(score)) + "%" # E.g., "82.5%"
    text_color = (255, 255, 0) # Cyan
    text_thickness = 5
    text_font = cv.FONT_HERSHEY_SIMPLEX
    text_font_scale = 2

    # Get the image's center coordinates, accounting for the text's size too.
    text_size = cv.getTextSize(text, text_font, text_font_scale, text_thickness)[0]
    center_x = (paper_width - text_size[0]) // 2
    center_y = (paper_height + text_size[1]) // 2
    text_coordinates = (center_x, center_y)

    # Lastly, add the text.
    cv.putText(paper, text, text_coordinates, text_font, text_font_scale, text_color, text_thickness)

    return paper

def extract_answers_from_paper(paper):
    paper = cv.cvtColor(paper, cv.COLOR_BGR2GRAY)
    paper = cv.threshold(paper, 127, 255, cv.THRESH_BINARY_INV)[1]


```

```
    # The answer grid is a 2-D array filled with zeros at first that will later hold the 'weight' of choices.
    answer_grid = np.zeros( (NUM_QUESTIONS, NUM_CHOICES) )
    # Evenly splits the paper into seperate rows of questions, each having a number of choices.
    questions = list(np.array_split(paper, NUM_QUESTIONS, axis=0)) #np.vsplit(paper, NUM_QUESTIONS)


    for count_question, question in enumerate(questions):
        # Evenly splits every column of choices in each row into multiple 'cells' or choice.
        choices = list(np.array_split(question, NUM_CHOICES, axis=1)) #np.hsplit(question, NUM_CHOICES)
        for count_choice, choice in enumerate(choices):
            # Stores the 'weight' of the choice/cell (i.e., how many filled pixels it had.)
            answer_grid[count_question][count_choice] = cv.countNonZero(choice)

    # Since each cell is more or less going to have a number of filled-in pixels, we will
    # find the maximumly-filled area along the x-axis or the choices of each question.
    answer_list = np.argmax(answer_grid, axis=1)

    #cv.namedWindow("test")
    #while True:
    #    cv.imshow("test", question)
    #    if (cv.waitKey(33) == ord('z')):
    #        break

    return answer_list


def grade(paper):
    answers = extract_answers_from_paper(paper)

    score = check_answers(answers, ANSWERS_KEY)

    paper = display_gradings_on_paper(score, answers, ANSWERS_KEY, paper)

    return score, paper
```

```python
import cv2 as cv
import numpy as np


# Sorts a given list of points in a clock-wise manner.
def sort_points(points):
    points = np.reshape(points, (4, 2)) # FOUR pairs of (x, y) coordinates
    sorted_points = np.zeros((4, 2), dtype=np.int32)

    summation = np.sum(points, axis=1)
    sorted_points[0] = points[np.argmin(summation)]
    sorted_points[3] = points[np.argmax(summation)]

    difference = np.diff(points, axis=1)
    sorted_points[1] = points[np.argmin(difference)]
    sorted_points[2] = points[np.argmax(difference)]

    return sorted_points

# Usually, the paper or document in an image, when compared with other objects (e.g., pens, erasers, etc), would be the largest shape.
def find_largest_contours(contours):
    largest = np.array([])

    max_area = 0 # A variable to hold the highest area of each contour and check it against the next contour.

    for contour in contours:
        area = cv.contourArea(contour)

        if area > 5000:
            # True in these two lines indicates that the "curve" (shape) is a closed one (rectangle in this case).
            perimeter = cv.arcLength(contour, True)
            curve = cv.approxPolyDP(contour, (0.02*perimeter), True)

            if (area > max_area) and (len(curve) == 4): # Rectangles have 4 points.
                max_area = area
                largest = curve

    return largest

# TODO: Move these to a centralized configuration section or file.
#
# 480p for a higher processing speed
WIDTH = 640
HEIGHT = 480
# The margin or outline/border of the document
MARGIN = 25 # TODO: Obtain this automatically from the image dimensions and ratios.

def scan(img):
```

```
    img = cv.resize(img, (WIDTH, HEIGHT))
    img_original = img

    img_grayscale = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
    img_grayscale = cv.GaussianBlur(img_grayscale, (5, 5), 1)

    img_grayscale = img

    # TODO: lower the brightness and preserve edges.
    #img = cv.bilateralFilter(img, 11, 17, 17)
    img = cv.Canny(img, 200, 200)

    kernel = np.ones((5, 5))
    img = cv.dilate(img, kernel, iterations=2)
    img = cv.erode(img, kernel, iterations=1)

    #cv.namedWindow("debug")
    #cv.imshow("debug", img)


    # Extracts all the shapes ("contours") from the image.
    contours, _hierarchy = cv.findContours(img, cv.RETR_EXTERNAL,
cv.CHAIN_APPROX_SIMPLE)

    largest_countors = find_largest_contours(contours) # Finds the largest
contour.
    if largest_countors.size == 0: # No shapes were detected.
        return img_original, np.zeros((HEIGHT, WIDTH, 3), np.uint8) # So, we
return a blank image.

    img_bordered = img_original
    cv.drawContours(img_bordered, [largest_countors.astype(int)], -1, (0, 255,
0), 2)

    largest_countors = sort_points(largest_countors) # Sorts its points.

    # De-skew the image
    src = np.float32(largest_countors) # Coordinates of the largest shape
(i.e., the paper or document).
    dst = np.float32([[0, 0], [WIDTH, 0], [0, HEIGHT], [WIDTH, HEIGHT]]) #
Same size as the resized img.
    matrix = cv.getPerspectiveTransform(src, dst)
    img_scanned = cv.warpPerspective(img_original, matrix, (WIDTH, HEIGHT))


    # Crops the blurry outline/margin.
    img_scanned = img_scanned[MARGIN:img_scanned.shape[0]-MARGIN,
MARGIN:img_scanned.shape[1]-MARGIN]
    img_scanned = cv.resize(img_scanned, (WIDTH, HEIGHT))

    # Anti-aliasing
    img_scanned = cv.medianBlur(img_scanned, 9)
```

```
95      return img_bordered, img_scanned
```