YOLO MCQ Grader Documentation

This document explains your Python script section-by-section in the same style you wrote in chat: imports, each function's purpose, how data flows through the pipeline, and important notes for production use.

1) Imports and what each one is used for

script starts by importing libraries needed for the OCR/MCQ grading pipeline. Below is every import grouped by purpose, with a clear reason for why it exists.

Backend / API framework

```
from quart import current_app, jsonify
```

- current_app gives access to the Quart app configuration (upload folder, allowed extensions, max file size, etc.).
- jsonify is used to return JSON responses in Quart/Flask style APIs (in this script it is imported but not directly used).

Database id handling

```
from bson import ObjectId
```

- Converts exam_id (string) into a MongoDB ObjectId so the value is stored/queried correctly in MongoDB.

Standard utilities

```
import base64

import re

import json
import os

import io

from datetime import datetime

from typing import List, Dict, Any, Tuple
```

- base64: encode images to base64 for sending to OpenAI Vision (data:image/png;base64,...)
- re: regular expressions used to extract JSON blocks from model output
- json: parse and generate JSON structures
- os: filesystem paths, creating folders, joining directories safely
- io / BytesIO: in-memory byte buffers for converting PIL image to bytes (no need to write temp files)
- datetime: timestamps for naming uploaded files uniquely
- typing: type hints like List, Dict, Any (better readability + tooling support)

Upload helper

```
from werkzeug.utils import secure_filename
```

- secure_filename sanitizes user-uploaded filenames to prevent path traversal and unsafe characters.
- Example: '../../../evil.png' becomes a safe filename.

OpenAI client

```
from openai import AsyncOpenAI
```

- AsyncOpenAI is used to call OpenAI Vision/Chat asynchronously to extract the student’s index number from the student answer sheet image.

Image processing

```
from PIL import Image, ImageOps, ExifTags
```

- Image: open and manipulate images
- ImageOps: padding and simple image operations
- ExifTags: read EXIF metadata (especially camera orientation) to rotate images correctly

ML / detection / math

```
import numpy as np

from ultralytics import YOLO

from sklearn.cluster import KMeans
```

- numpy: array operations for computing centers of bounding boxes and clustering inputs
- YOLO (Ultralytics): loads your trained model (best5.pt) and runs object detection
- KMeans: auto-detect whether the sheet has 2 or 3 columns and group questions in reading order

Unused imports (cleanup note)

These are imported but not used (based on the script you shared). Removing them is recommended for cleaner production code:

- jsonify (not used directly)
- Tuple (imported but not used if you don’t return Tuple types)
- matplotlib.pyplot as plt (imported but not used)
- Note: BytesIO is used (good).

2) OpenAI client initialization (important security note)

```
client = AsyncOpenAI(api_key="sk-proj-...")
```

This creates an async OpenAI client so later you can call:

```
await client.chat.completions.create(...)
```

🚨 Security issue (Must fix before pushing to GitHub)

- API key is hard-coded inside the Python file. This is dangerous because it can leak in GitHub, logs, screenshots, or shared ZIP files.
- Best practice: store the key in an environment variable (OPENAI_API_KEY).
- Then read it using: os.getenv('OPENAI_API_KEY').

3) File validation helper

allowed_file(filename)

This function checks whether the uploaded file extension is allowed. Typical logic:

- Verify there is a dot in the filename (so it has an extension).
- Extract the file extension after the last dot.
- Check if the extension exists in current_app.config['ALLOWED_EXTENSIONS'].

This is used later inside grade_answers() before saving uploaded files, so invalid files are rejected early.

4) Convert PIL image → base64 for OpenAI Vision

image_to_base64(img)

What it does: converts a PIL image into a base64 string.

- Creates an in-memory buffer (BytesIO()).
- Saves the PIL image into that buffer as PNG.
- Encodes buffer bytes → base64 string and returns it.

This is required because the OpenAI Vision request uses a data URL format like:

```
"url": "data:image/png;base64,...."
```

5) Extract JSON from OpenAI’s response

extract_json_block(text)

OpenAI sometimes wraps JSON in a markdown code block like:

~~~~
```json
~~~~

~~~
{ ... }

```
~~~

This function:

- Uses regex to find JSON inside triple backticks (optionally tagged as json).
- Returns only the JSON object part: { ... }
- If no code block is found, it returns the whole text stripped.

Later you do:

```
student_json = extract_json_block(student_output_text)

json.loads(student_json)
```

6) Image preprocessing before YOLO detection

preprocess_image(image: Image.Image)

Goal: normalize input images so YOLO sees consistent size/layout even if photos come from different phones/scanners.

Step 1: Target size

Sets: target_size = 1280. The pipeline standardizes images to 1280×1280.

Step 2: EXIF orientation fix

Phones/cameras often store “rotation” in EXIF metadata instead of rotating pixels. This block:

- Reads EXIF data.
- Finds the Orientation tag.
- Rotates image if orientation is 3 / 6 / 8.
- If EXIF read fails, logs a warning and continues (no crash).

Step 3: Grayscale

image.convert("L") converts to grayscale (1 channel). This reduces noise and can improve detection consistency.

Step 4: Resize

thumbnail((1280, 1280)) scales down the image to fit inside the target while preserving aspect ratio.

Step 5: Pad

ImageOps.pad(..., (1280,1280), color=0) pads the resized image so the final size is exactly 1280×1280.

Return values

- Returns the processed PIL image.
- Also returns a NumPy array version (even if not used later, it can be helpful for debugging or future improvements).

7) Sorting questions in “reading order” using column detection

group_questions_by_columns(question_boxes, image_width, num_columns=0)

Problem it solves: MCQ sheets often have multiple columns. YOLO detects question blocks but the detections are not naturally ordered. You need human reading order: top-to-bottom in left column, then top-to-bottom in next column.

Steps

- Compute x-center of each question box: (x1 + x2) / 2.
- If num_columns == 0, auto-detect 2 vs 3 columns using KMeans.
- Run KMeans for k=2 and k=3, compare inertia (distortion).
- Choose 2 columns if it is good enough; otherwise choose 3.
- Fit KMeans with the chosen k; assign each question to a column label (0..k-1).
- Group questions by label into columns.

- Sort columns left→right using mean x-center.
- Sort each column top→bottom using y1.
- Concatenate columns to get final reading order.

8) Match bubbles (filled/unfilled) to each question

match_bubbles_to_questions(questions, bubbles)

For each detected question box, the function collects bubbles that belong to it based on spatial rules.

Matching conditions

- Horizontal: bx1 >= qx1 and bx2 <= qx2 (bubble lies inside question region).
- Vertical: bubble is within the question's y-range with a small tolerance margin (±10 pixels).

After filtering

- Sort matched bubbles left-to-right by their x-center.
- Store them as answers for that question.
- Each answer includes the bubble box coordinates and whether it is filled (True/False).

Output shape per question:

```
{
  "question_box": { ... },
  "answers": [
    {"box": [x1, y1, x2, y2], "filled": true/false},
    ...
  ]
}
```

9) Run YOLO detection and produce grouped structure

detect_and_group(image, model, image_width=1280)

This is the core YOLO + layout + grouping pipeline.

Pipeline steps

- Preprocess image to 1280×1280 grayscale.
- Run YOLO: results = model(processed_img, conf=0.5).
- Extract YOLO outputs: bounding boxes, class IDs, class name mapping.
- Build a filtered list of boxes containing only: question, filled, unfilled.
- Separate question_boxes and bubble_boxes.
- Sort question_boxes by columns (reading order).
- Match bubble_boxes to each question.
- Return the final grouped question list (ready for grading).

10) Debug printing helper

print_grouped_questions(grouped_data)

Prints each question and its bubbles for debugging:

- Question bounding box
- Each bubble box
- Whether each bubble is filled/unfilled

This helps you verify if YOLO + grouping is working before grading logic is applied.

11) Converting bubble index → option letter

index_to_letter(index)

Converts bubble position into letter labels:

- 0 → A
- 1 → B
- 2 → C
- …

This is used to generate human-readable reports (letters instead of numeric indices).

12) Compare student answers vs answer key

compare_answers_by_index(answer_group, mark_group)

Inputs

- answer_group: grouped questions from the student sheet
- mark_group: grouped questions from the answer key sheet

Process (for each question i)

- Find which bubbles are filled in student answers.
- Find which bubbles are filled in answer key.
- Apply rules: multiple marked → incorrect; exact match → correct; mismatch → incorrect.
- Convert indices to letters for reporting.

Return value

Returns a final structure like:

```
{

  "score": 18,

  "total": 20,
  "percentage": 90.0,

  "report": {

    "1": {"student_answer": ["B"], "correct_answer": ["B"], "status":
"correct"},

    "2": {"student_answer": ["A"], "correct_answer": ["C"], "status":
"incorrect"}

  }

}
```

13) The main async function: grade_answers(...)

Signature

```
async def grade_answers(files, student_uuid: str, index_number: str,
exam_id: str):
```

This is the controller function that ties everything together: uploads, model loading, index extraction, detection, grading, and final response building.

A) Configure upload rules

- upload folder: "uploads"
- allowed extensions: {'png', 'jpg', 'jpeg'}
- max content length: 5MB
- Create upload directory if missing

B) Validate incoming files

- Requires files to contain: 'user' (student sheet) and 'key' (answer key).
- Checks filename is not empty.
- Checks extension allowed using allowed_file().
- If invalid, raises error messages.

C) Load the trained YOLO model

- Build model path relative to the script directory: ./trained_models/best5.pt
- Load using: model = YOLO(model_path)

D) Save uploaded files with timestamp

- Create timestamp like 20251216073045 to avoid overwriting.
- Sanitize names using secure_filename().
- Save to uploads/.

E) Open images

- student_image = Image.open(student_path)
- answer_key_image = Image.open(answer_key_path)

F) Extract student index number using OpenAI Vision

- Convert student image to base64.
- Call OpenAI chat completion with a prompt that requests JSON: {"student_index": "..."}.
- Temperature set to 0 for consistent results.
- Parse response using extract_json_block() + json.loads().
- Validate that student_index exists; otherwise raise an error.
- Normalize: if not starting with "indx", prefix "indx".

Important documentation note

In the final return object, your code stores index_number from the function parameter, not the student_index detected by OpenAI. If your intention is to store the detected index, you should return student_index instead.

G) Detect questions and bubbles

- answer_group = detect_and_group(student_image, model)
- mark_group = detect_and_group(answer_key_image, model)

H) Compare answers and compute score

- result = compare_answers_by_index(answer_group, mark_group)

I) Build final response object

- student_uuid
- exam_id as ObjectId(exam_id)

- index_number
- score, total, percentage
- per-question report
- uploaded filenames
- timestamp

J) Error handling

Any exception is caught and re-raised as a ValueError with a general processing failed message.

14) End-to-end: What the script does

- User uploads 2 images: student sheet + answer key.
- Server saves them safely.
- OpenAI extracts student index number from student sheet.
- YOLO detects question regions and bubbles (filled/unfilled).
- Questions are sorted in reading order using column detection (2 or 3 columns).
- Bubbles are assigned to each question and sorted left-to-right (A, B, C, ...).
- Student answers are compared to the key to compute score and detailed report.
- A final JSON-like result structure is returned/stored.