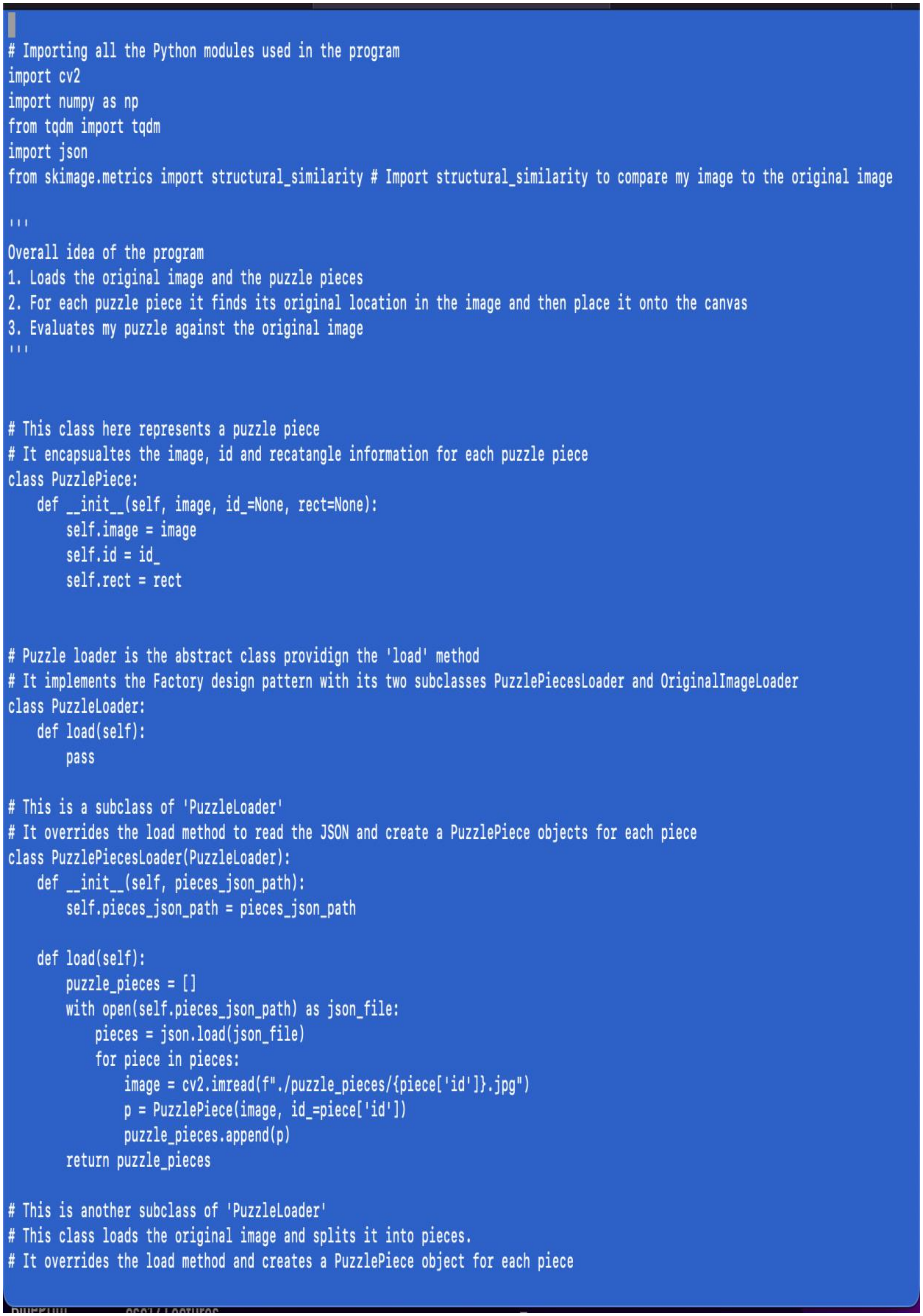
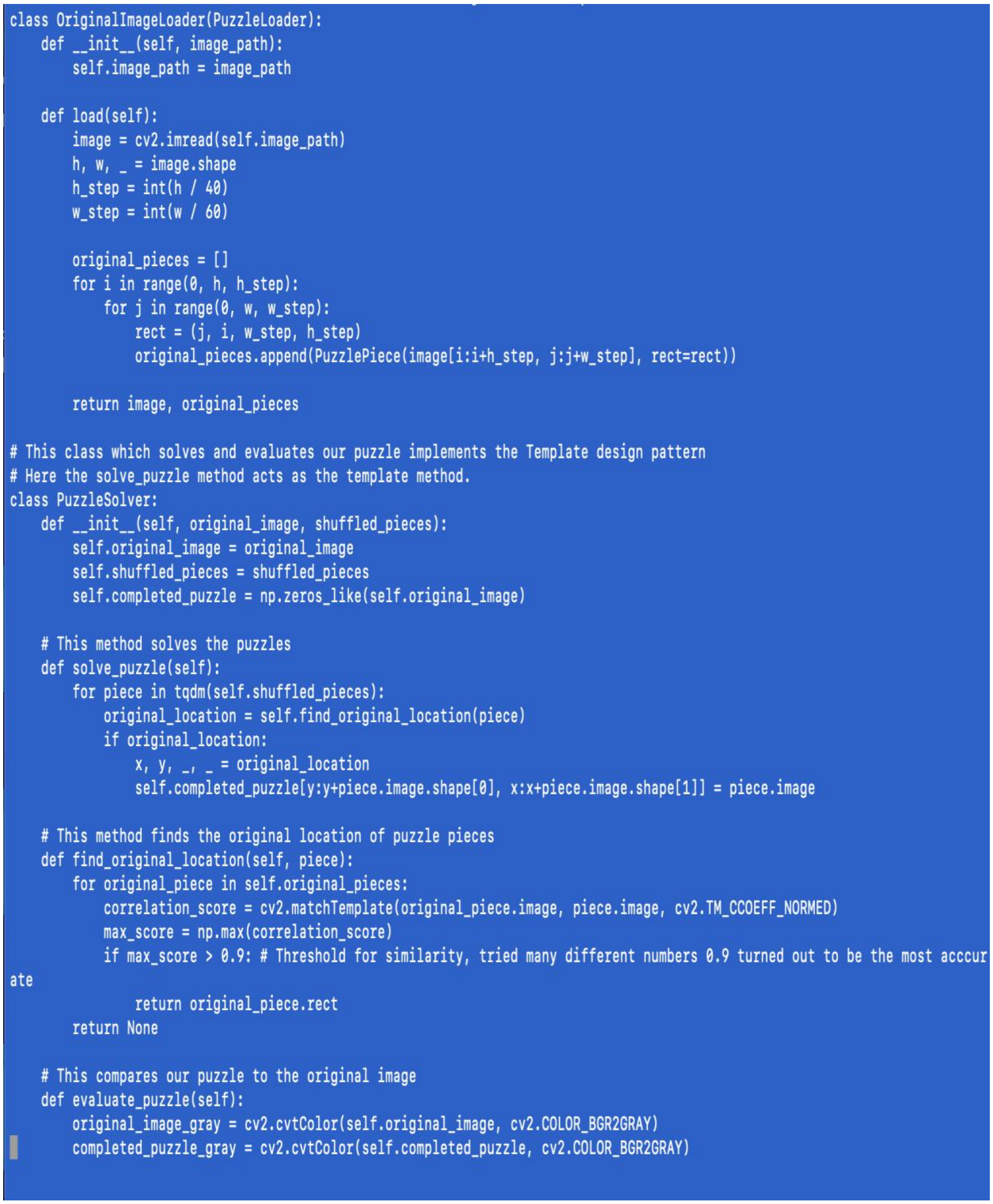
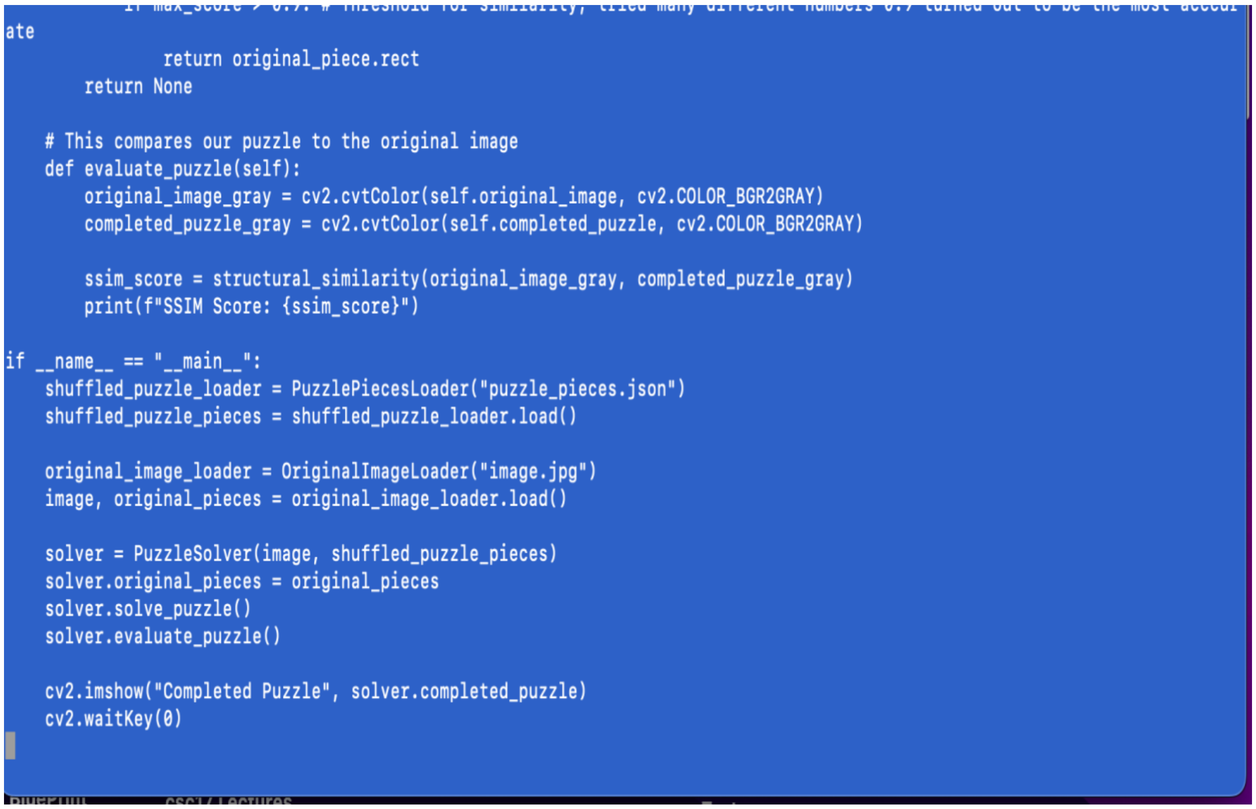


```python
# Importing all the Python modules used in the program
import cv2
import numpy as np
from tqdm import tqdm
import json
from skimage.metrics import structural_similarity # Import structural_similarity to compare my image to the original image

'''
Overall idea of the program
1. Loads the original image and the puzzle pieces
2. For each puzzle piece it finds its original location in the image and then place it onto the canvas
3. Evaluates my puzzle against the original image
'''


# This class here represents a puzzle piece
# It encapsualtes the image, id and recatangle information for each puzzle piece
class PuzzlePiece:
    def __init__(self, image, id_=None, rect=None):
        self.image = image
        self.id = id_
        self.rect = rect


# Puzzle loader is the abstract class providign the 'load' method
# It implements the Factory design pattern with its two subclasses PuzzlePiecesLoader and OriginalImageLoader
class PuzzleLoader:
    def load(self):
        pass

# This is a subclass of 'PuzzleLoader'
# It overrides the load method to read the JSON and create a PuzzlePiece objects for each piece
class PuzzlePiecesLoader(PuzzleLoader):
    def __init__(self, pieces_json_path):
        self.pieces_json_path = pieces_json_path

    def load(self):
        puzzle_pieces = []
        with open(self.pieces_json_path) as json_file:
            pieces = json.load(json_file)
            for piece in pieces:
                image = cv2.imread(f"./puzzle_pieces/{piece['id']}.jpg")
                p = PuzzlePiece(image, id_=piece['id'])
                puzzle_pieces.append(p)
        return puzzle_pieces

# This is another subclass of 'PuzzleLoader'
# This class loads the original image and splits it into pieces.
# It overrides the load method and creates a PuzzlePiece object for each piece
```



```
class OriginalImageLoader(PuzzleLoader):
    def __init__(self, image_path):
        self.image_path = image_path

    def load(self):
        image = cv2.imread(self.image_path)
        h, w, _ = image.shape
        h_step = int(h / 40)
        w_step = int(w / 60)

        original_pieces = []
        for i in range(0, h, h_step):
            for j in range(0, w, w_step):
                rect = (j, i, w_step, h_step)
                original_pieces.append(PuzzlePiece(image[i:i+h_step, j:j+w_step], rect=rect))

        return image, original_pieces

# This class which solves and evaluates our puzzle implements the Template design pattern
# Here the solve_puzzle method acts as the template method.
class PuzzleSolver:
    def __init__(self, original_image, shuffled_pieces):
        self.original_image = original_image
        self.shuffled_pieces = shuffled_pieces
        self.completed_puzzle = np.zeros_like(self.original_image)

    # This method solves the puzzles
    def solve_puzzle(self):
        for piece in tqdm(self.shuffled_pieces):
            original_location = self.find_original_location(piece)
            if original_location:
                x, y, _, _ = original_location
                self.completed_puzzle[y:y+piece.image.shape[0], x:x+piece.image.shape[1]] = piece.image

    # This method finds the original location of puzzle pieces
    def find_original_location(self, piece):
        for original_piece in self.original_pieces:
            correlation_score = cv2.matchTemplate(original_piece.image, piece.image, cv2.TM_CCOEFF_NORMED)
            max_score = np.max(correlation_score)
            if max_score > 0.9: # Threshold for similarity, tried many different numbers 0.9 turned out to be the most accccur
ate
                return original_piece.rect
        return None

    # This compares our puzzle to the original image
    def evaluate_puzzle(self):
        original_image_gray = cv2.cvtColor(self.original_image, cv2.COLOR_BGR2GRAY)
        completed_puzzle_gray = cv2.cvtColor(self.completed_puzzle, cv2.COLOR_BGR2GRAY)
```



```
            if max_score > 0.7: # Threshold for similarity, tried many different numbers 0.7 turned out to be the most accur
ate
                return original_piece.rect
        return None

    # This compares our puzzle to the original image
    def evaluate_puzzle(self):
        original_image_gray = cv2.cvtColor(self.original_image, cv2.COLOR_BGR2GRAY)
        completed_puzzle_gray = cv2.cvtColor(self.completed_puzzle, cv2.COLOR_BGR2GRAY)

        ssim_score = structural_similarity(original_image_gray, completed_puzzle_gray)
        print(f"SSIM Score: {ssim_score}")

if __name__ == "__main__":
    shuffled_puzzle_loader = PuzzlePiecesLoader("puzzle_pieces.json")
    shuffled_puzzle_pieces = shuffled_puzzle_loader.load()

    original_image_loader = OriginalImageLoader("image.jpg")
    image, original_pieces = original_image_loader.load()

    solver = PuzzleSolver(image, shuffled_puzzle_pieces)
    solver.original_pieces = original_pieces
    solver.solve_puzzle()
    solver.evaluate_puzzle()

    cv2.imshow("Completed Puzzle", solver.completed_puzzle)
    cv2.waitKey(0)
```

<u>**Code Explanation**</u>

1: Import cv2 → CV2 is the main module in OpenCV which is a module that helps process images.
2: Import Numpy as np → Here we are importing the Numpy module as np, so that we can refer to it as np in the program. Numpy provides many different mathematical operations on arrays.
3. from tqdm import tqdm → A module to add progress bars
4. import json → A python module to help handle JSON data
5. from skimage. metrics import structural_similarity → Here implementing a function from the scikit library. I am using this here to calculate the Structural Similarity Index (SSI) between two images.

```
6. class PuzzlePiece:
7.      def __init__(self, image, id =None, rect=None):
8.              self. image = image
9.              self.id = id
10.             self.rect = rect
```

The first class of the program represents a puzzle piece.
It encapsulates the image, id, and rectangle information for each puzzle piece.

Image: The image of the puzzle piece

Id: The identifier for the puzzle piece. The ones provided in the json file. This helps us load a specific puzzle piece from the puzzle_pieces folder.

Rect: In our OriginalImageLoader class when we create PuzzlePiece objects from the original image we are setting rect. Rect stores the rectangle information. First you have the position of the piece and then you have the size which have been determined by h_step and w_step.
Since the height of the image is 1080 and the widht is 1920
h_step = int (h / 40) → 1080/40 = 27
W_step = int (w / 60) → 1920/60 = 32

The puzzle is split into 60 pieces wide by 40 pieces tall. Each piece is 27 pixels in height and 32 in width.

What are pixels?
The basic unit of programmable color in a computer image. The smallest unit in digital display.

We are not doing that in the PuzzlePiece Loader class.

```
class PuzzleLoader:
    def load(self):
        pass
```

Here I am creating an abstract class with a method called load that is going to be implemented by two subclasses below.

```
class OriginallmageLoader(PuzzleLoader):
    def __init__ (self, image_path):
        self. image_path = image_path
```

Here the __init__ the constructor method initializes an object that holds the path to the folder of all the individual images.

```
def load (self):
    puzzle_pieces = []
    with open (self. pieces_json_path) as json_file:
        pieces = json.load(json_file)
        for piece in pieces:
            image = cv2. imread (f"./puzzle _pieces/(piecel 'id' I}. jpg")
             PuzzlePiece(image, id_=piece[' id' ])
            puzzle_pieces. append(p)
    return puzzle_pieces
```

First, I initialize a puzzle_pieces list that is to contain our PuzzlePiece object. Then I open and load the json file that contains all the json. For each json the “id” refers to an image in the folder.

Open() → Opens the json file in read mode.

Json.load() → Loads the json data into a python dictionary.

After we iterate over each piece

Line 22 reads the image file. --- f"./puzzle_pieces/{piece['id']}.jpg" ---. This constructs the filepath for the image using the id piece from the Jason data.

Line 23 creates a PuzzlePiece object passing the image and the ID.

Line 24 appends the Puzzlepiece object to the puzzle pieces list .

Line 25 returns this list.

```
26. class OriginallmageLoader(PuzzleLoader):
27.     def __init__ (self, image_path):
28.         self. image_path = image_path
29.         def load(self):
30.             image = cv2.imread(self. image_path)
31.             h, W, = image. shape
32.             h_step = int (h 40)
33.             W_step E int (w 60)
34.             original_pieces = []
35.             for i in range (0, h, h_step):
36.                 for j in range (0, W, w_step):
37.                     rect = (j, I, w_step, h_step)
38.                     original_pieces. append (PuzzlePiece(image[i:i+h_step,
j:j+w_step], rect=rect))
39.         return image, original_pieces
```

Line 26 is the start of the OriginalImageLoader class which implements the Puzzle loader class and overrides the load function.

The class loads the original image and splits it into pieces, creating PuzzlePiece objects out of it.

Line 27 is the constructor of the class which just like the previous class keeps the path of the image in the directory.

Line 29 is the start of the load function.
Line 30 reads the original image using OpenCV’s imread function.
Line 31 gets the dimensions of the image.
Line 32 and Line 33 h_step and w_step define the measurements into which the image must be split in.

Since the height of the image is 1080 and the widht is 1920
h_step = int (h / 40) → 1080/40 = 27

W_step = int (w / 60) → 1920/60 = 32

The puzzle is split into 60 pieces wide by 40 pieces tall. Each piece is 27 pixels in height and 32 in width.

Line 34 initializes a list that is going to contain the puzzle piece objects.

Line 35, 36, 37, 38 is the nested for loop that is responsible for splitting the original image into pieces, iterating over its height and width in steps defined by ‘h_step’, ‘w_step’.

Line 35 the outer loop, iterates over the height of the original image in steps h_step. Starting at 0, going to 1080 and the step is 27. I represent the starting row index of the current piece.

Line 36 the inner loop, iterates over the width of the original image in steps w_step. J represents the starting column index of the current piece.

Line 37 calculates the rectangle information for the current piece. (J, I) represents the top left corner coordinates of the piece. w_step and h_step represent the width and height of the piece.

Line 38 creates a Puzzle Piece and appends it to the list. image [i:i+h_step, j:j+w_step] slices the original image to extract the current piece based on its coordinates.

Rect = rect passed the calculated rectangle information. Its width and height as well as its location in the image.

Line 39 returns the original image and the list of puzzle pieces.

```
Line 40 class PuzzleSolver:
Line 41         def _init__ (self, original image, shuffled _pieces):
Line 42                 self.original_image = original_image
Line 43                 self. shuffled_pieces = shuffled_pieces
Line 44                 self.completed_puzzle = np.zeros_like(self.original image)
```

Line 41 –44 is the constructor that initializes PuzzleSolver objects with the original image, shuffled pieces and a blank canvas for the completed puzzle.

self.completed_puzzle = np.zeros_like(self.original_image)

This line above initializes self.completed_puzzle as a numpy array filled with zeros and with the same shape as self.original_image.

```
        # This method solves the puzzles
Line 45         def solve_puzzle (self):
Line 46                 for piece in tqdm(self. shuffled pieces):
```

```
Line 47                         original_location = self. find_original_location(piece)
Line 48                         if original location:
Line 49                                 x, y, : original location
Line 50                         self. completed_puzzle[y: y+piece. image. shapelol,
X:X+piece.image.shape[1]] = piece. image
```

This method here solves the puzzle.
Line 46 iterates over each shuffled piece in the list. The tqdm creates a progress bar that tracks the iteration.

Line 47 calls the function original_location just underneath.

Line 48 – 50. If the original location is found, then places it in into its original location on the new canvas

```
# This method finds the original location of puzzle pieces
Line 51         def find_original_location(self, piece):
Line 52                 for original _piece in self. original pieces:
Line 53                         correlation_score = cv2.matchTemplate(original_piece. image,
piece. image, Cv2. TM_CCOEFF_NORMED)
Line 54                         max_score = np. max (correlation_score)
Line 55                         if max_score > 0.9: # Threshold for similarity, tried many different
numbers 0.9 turned out to be the most acccur          ate
Line 56                                 return original_piece. rect
Line 57                 return None
```

This method here compares the shuffled piece with each original piece using template matching. If a match with a high correlation is found it returns the rectangle of the original piece.

Line 53 below
correlation_score = cv2.matchTemplate(original_piece.image, piece.image, cv2.TM_CCOEFF_NORMED)

This line compares ‘piece.image’ with one of the original pieces.

Original_piece.image → the image of the original puzzle piece

Piece.image → The image of the shuffled piece

cv2.TM_CCOEFF_NORMED → This parameter specifies the method for template matching. It’s one of the methods available in OpenCV for template matching. The correlation coefficient lies between -1 and 1.

Line 54 max_score = np. max (correlation_score) finds the maximum score in the correlation_score array. The highest score will be the puzzle piece that matches the original piece the most.

Line 55 if max_score > 0.9
This checks if the maximum correlation score is greater than a predefined threshold (0.9 in this case). This threshold is used to determine whether the puzzle piece matches closely enough with the original piece.

Line 56. If the correlation score is above the threshold, it means the puzzle piece matches the original piece closely enough. Then we return the rectangle (**rect**) of the original piece, which represents its location in the original image.

# This compares our puzzle to the original image
Line 57 def evaluate_puzzle(self):
Line 58 original_image_gray = cv2.cvtColor (self. original_image, cv2.COLOR_BGR2GRAY)
Line 59 completed_puzzle_gray = cv2. cvtColor(self.completed_puzzle, cv2. COLOR BGR2GRAY)
Line 60 sim_score = structural_ similarity (original_image_gray, completed_puzzle _gray)
Line 61 print (f"SSIM Score: {ssim_score}")

Line 58 and 59 convert the original image and the completed canvas image to grayscale to simplify the image to a single channel, where each pixel represents the intensity of light.

Line 60 calculates the Structural Similarity Index between the two images.

Line 61 prints out the index, the closer the index is to 1 the more similar the two images are.

Line 62 if __name_ == "__main__":
Line 63 shuffled_puzzle_loader = PuzzlePiecesLoader("puzzle_pieces.json")
Line 64 shuffled_puzzle_pieces = shuffled_puzzle_loader.load()

Line 65 original_image_loader = OriginalImageLoader("image.jpg")
Line 66 image, original_pieces = original_image_loader. load()

Line 67 solver = PuzzleSolver (image, shuffled_puzzle_pieces)
Line 68 solver. original_pieces = original _pieces

Line 69 solver.solve_puzzle()
Line 70 solver. evaluate_puzzle()

Line 71 cv2. imshow ("Completed Puzzle", solver. completed_puzzle)
Line 72 cv2.waitKey(0)

Line 63 creates a ‘PuzzlePiecesLoader” object to load the shuffle pieces from the Jason file.
Line 64 calls the load method to load them.

Line 65 creates a ‘OriginalImageLoader’ object to load the original image.
Line 66 call the load method to load the image and split it into pieces. It returns the original method and a list of original puzzle pieces.

Line 67 creates a ‘PuzzleSolver’ object with the original image and shuffled pieces.
Line 68 Sets the original_pieces list as an attribute to solver.

Line 69 calls the solve_puzzle method to solve the puzzle.

Line 70 calls the evaluate_puzzle to evaluate our puzzle against the original

Line 71 displays the completed puzzle with the imshow() function

Line 72 waits indefinitely until a key is pressed to close the window.

Design Pattern Explanation

In my program I use the factory design pattern and the template design pattern. Below is a brief description of how they work.

The factory method is a creational design pattern that provides and interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.

The template method is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but let’s subclasses override specific steps of the algorithm without changing its structure.