Ain Shams University – Faculty of Engineering

Computer Engineering and Software Systems Program

CSE483 - Computer Vision

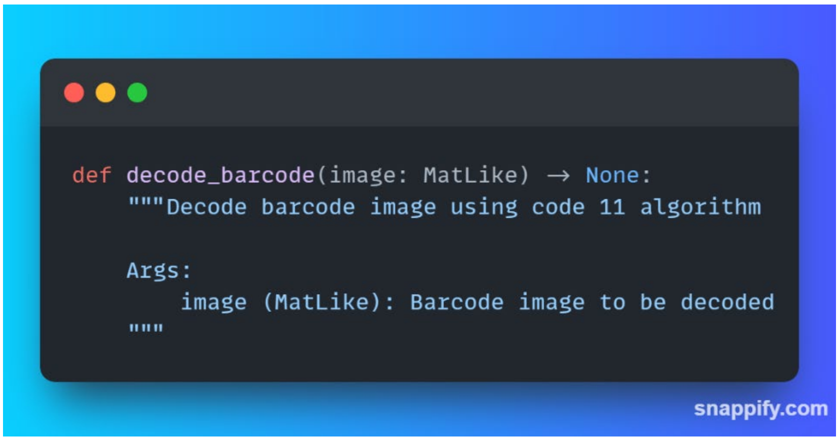
Barcode Extraction

Milestone 2

Name	ID
Youssef Adel Albert	21P0258
Mark Saleh Sobhi	21P0206
Ahmed Tarek Mahmoud	2100561

Table of Contents: phase 2

Decoding the barcode data



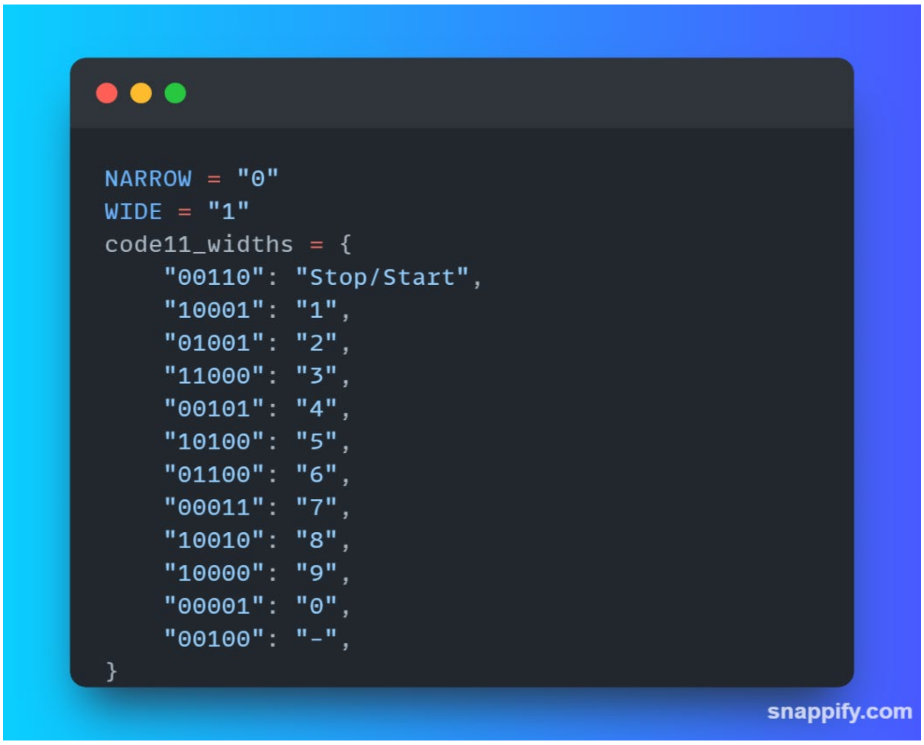
```python
def decode_barcode(image: MatLike) → None:
    """Decode barcode image using code 11 algorithm

    Args:
        image (MatLike): Barcode image to be decoded
    """
```

snappify.com

This function decodes a barcode image using the Code 11 algorithm. It identifies the barcode's individual bars (black and white) and their widths to map the encoding into corresponding symbols.

Constants and Code Definitions

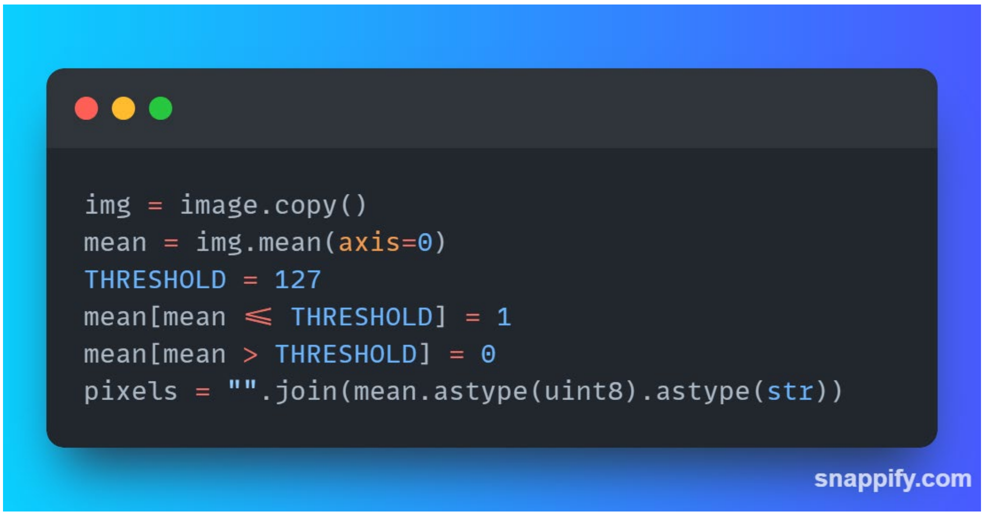


```python
NARROW = "0"
WIDE = "1"
code11_widths = {
    "00110": "Stop/Start",
    "10001": "1",
    "01001": "2",
    "11000": "3",
    "00101": "4",
    "10100": "5",
    "01100": "6",
    "00011": "7",
    "10010": "8",
    "10000": "9",
    "00001": "0",
    "00100": "-",
}
```

snappify.com

Here we start by defining constants for "narrow" (0) and "wide" (1) bar encodings and a dictionary (code11_widths) mapping bar patterns to their respective digits or symbols.

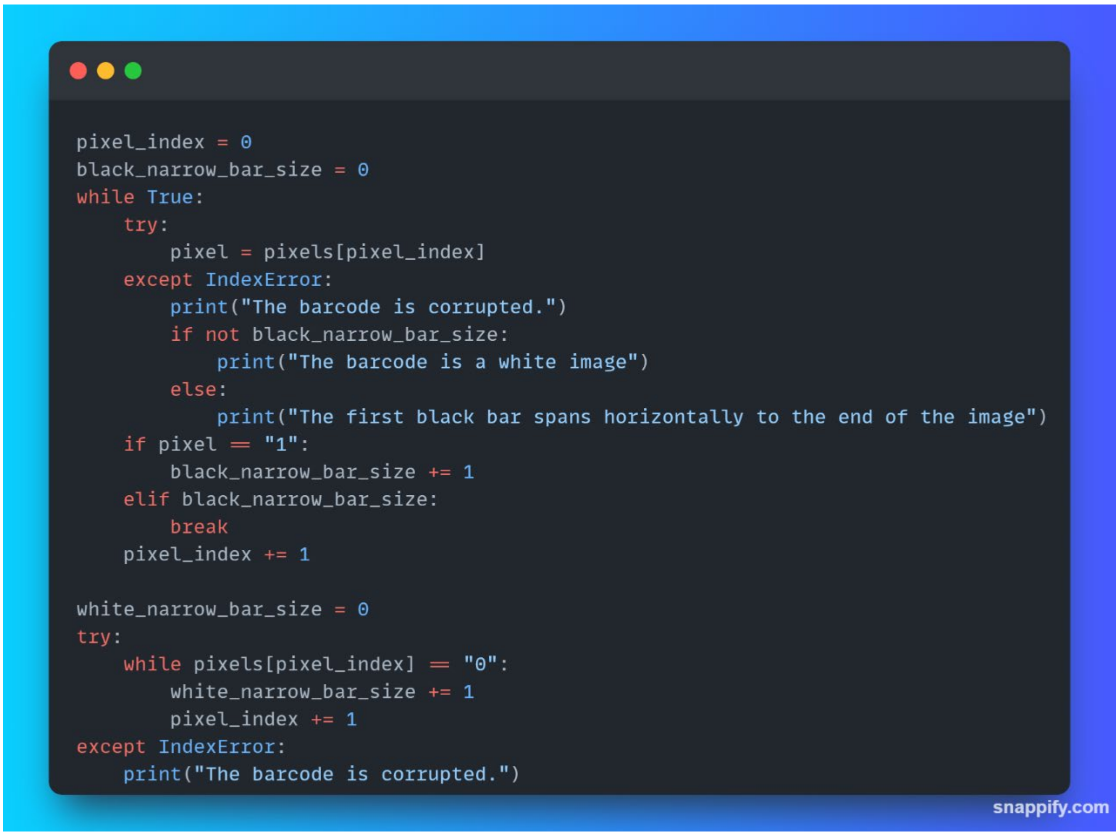
Image Preprocessing



```
img = image.copy()
mean = img.mean(axis=0)
THRESHOLD = 127
mean[mean <= THRESHOLD] = 1
mean[mean > THRESHOLD] = 0
pixels = "".join(mean.astype(uint8).astype(str))
```

The barcode image is copied, and the mean intensity of each column is calculated to binarize the image into black (1) and white (0) pixels based on a threshold. This results in a string of 0 and 1 values representing the barcode.

Finding Narrow Bar Sizes

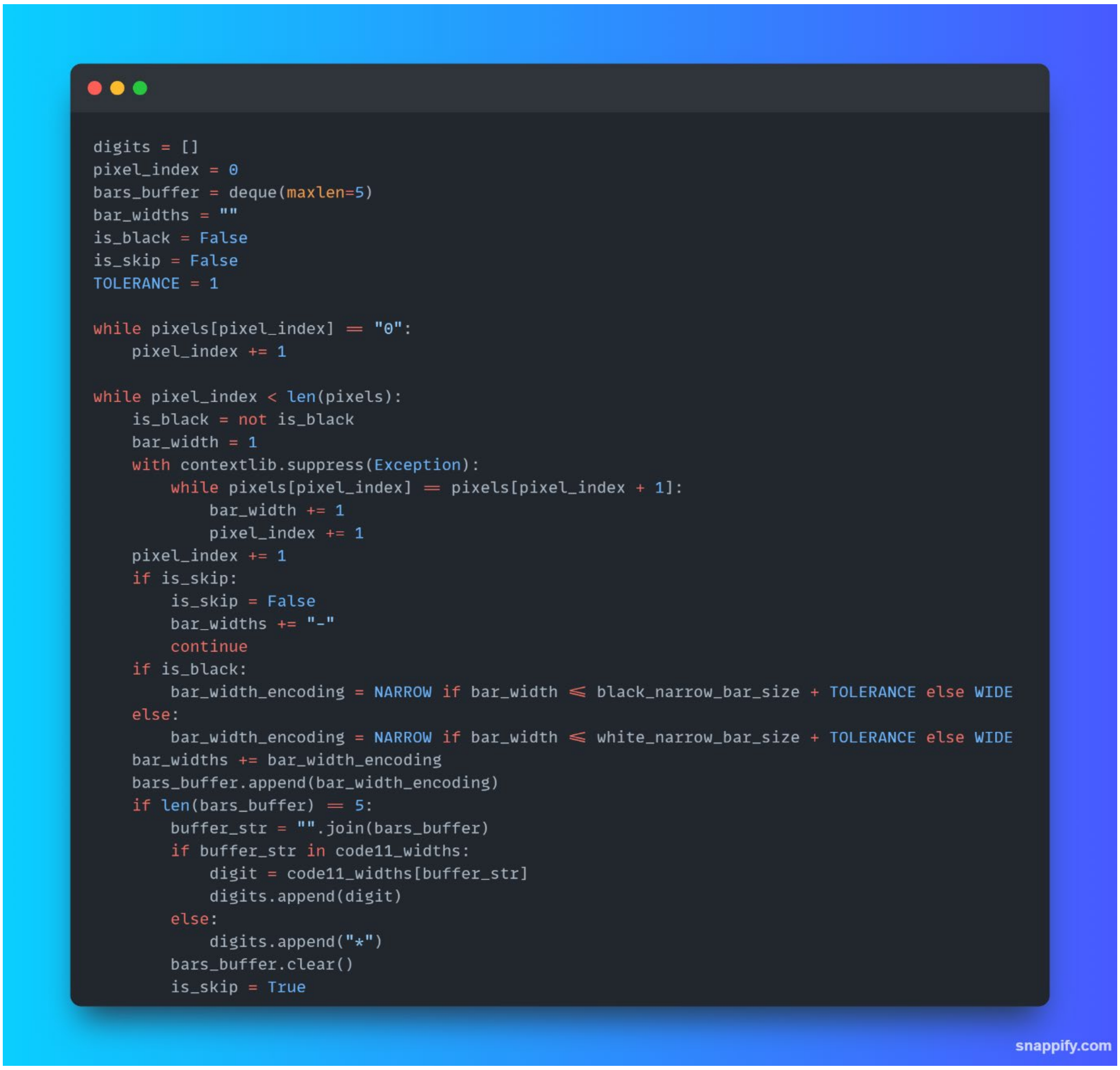


```
pixel_index = 0
black_narrow_bar_size = 0
while True:
    try:
        pixel = pixels[pixel_index]
    except IndexError:
        print("The barcode is corrupted.")
        if not black_narrow_bar_size:
            print("The barcode is a white image")
        else:
            print("The first black bar spans horizontally to the end of the image")
    if pixel == "1":
        black_narrow_bar_size += 1
    elif black_narrow_bar_size:
        break
    pixel_index += 1

white_narrow_bar_size = 0
try:
    while pixels[pixel_index] == "0":
        white_narrow_bar_size += 1
        pixel_index += 1
except IndexError:
    print("The barcode is corrupted.")
```

Then we measure the widths of the first narrow black and white bars. These sizes are used to categorize subsequent bars into "narrow" or "wide" based on their widths.

Decoding Bars



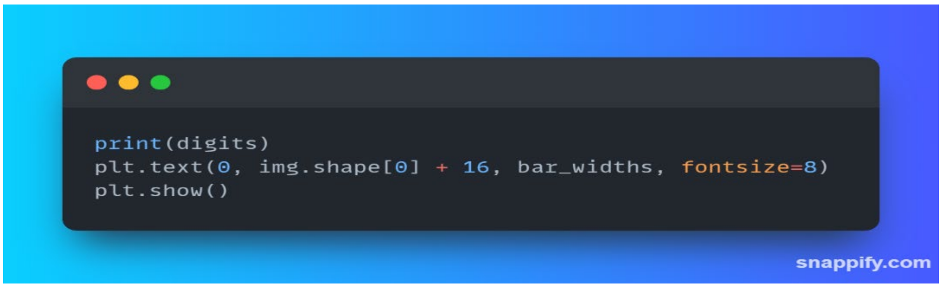
```
digits = []
pixel_index = 0
bars_buffer = deque(maxlen=5)
bar_widths = ""
is_black = False
is_skip = False
TOLERANCE = 1

while pixels[pixel_index] == "0":
    pixel_index += 1

while pixel_index < len(pixels):
    is_black = not is_black
    bar_width = 1
    with contextlib.suppress(Exception):
        while pixels[pixel_index] == pixels[pixel_index + 1]:
            bar_width += 1
            pixel_index += 1
    pixel_index += 1
    if is_skip:
        is_skip = False
        bar_widths += "-"
        continue
    if is_black:
        bar_width_encoding = NARROW if bar_width <= black_narrow_bar_size + TOLERANCE else WIDE
    else:
        bar_width_encoding = NARROW if bar_width <= white_narrow_bar_size + TOLERANCE else WIDE
    bar_widths += bar_width_encoding
    bars_buffer.append(bar_width_encoding)
    if len(bars_buffer) == 5:
        buffer_str = "".join(bars_buffer)
        if buffer_str in code11_widths:
            digit = code11_widths[buffer_str]
            digits.append(digit)
        else:
            digits.append("*")
        bars_buffer.clear()
        is_skip = True
```

A deque buffer is used to read bar patterns in chunks of five (Code 11 uses five bars to encode each symbol). Each bar's width is measured, classified as "narrow" or "wide," and appended to the buffer. If a valid code is recognized, it's decoded into a symbol; otherwise, an asterisk (*) is used to indicate an error. A spacer (white narrow bar) is skipped after decoding each digit.

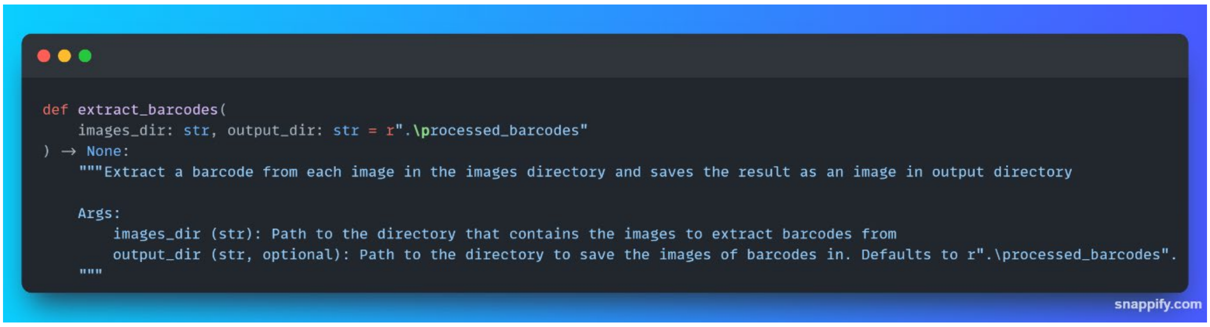
Output and Debugging



```
print(digits)
plt.text(0, img.shape[0] + 16, bar_widths, fontsize=8)
plt.show()
```

Finally, the decoded digits are printed, and the bar widths are visualized for debugging purposes.

Apply our pipeline to images

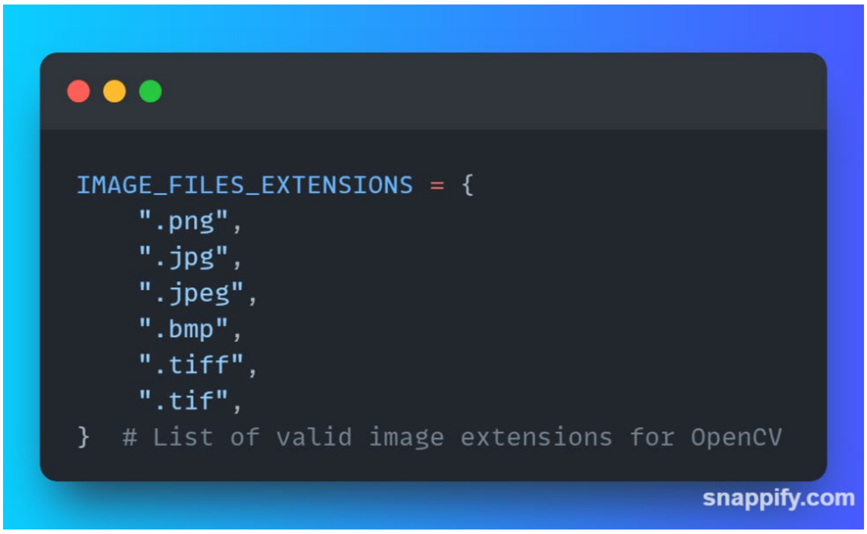


```python
def extract_barcodes(
    images_dir: str, output_dir: str = r".\processed_barcodes"
) → None:
    """Extract a barcode from each image in the images directory and saves the result as an image in output directory

    Args:
        images_dir (str): Path to the directory that contains the images to extract barcodes from
        output_dir (str, optional): Path to the directory to save the images of barcodes in. Defaults to r".\processed_barcodes".
    """
```

This function processes images in a specified directory to extract barcodes, decode them, and save the results. It applies the barcode extraction pipeline on each valid image file.

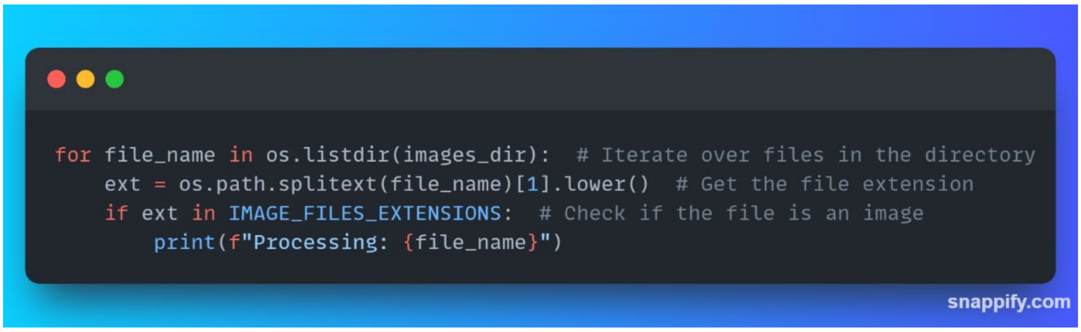
Define Valid Image Extensions



```python
IMAGE_FILES_EXTENSIONS = {
    ".png",
    ".jpg",
    ".jpeg",
    ".bmp",
    ".tiff",
    ".tif",
}  # List of valid image extensions for OpenCV
```

Here we starts by defining a set of valid image file extensions that can be processed by OpenCV. This ensures only supported file types are processed.

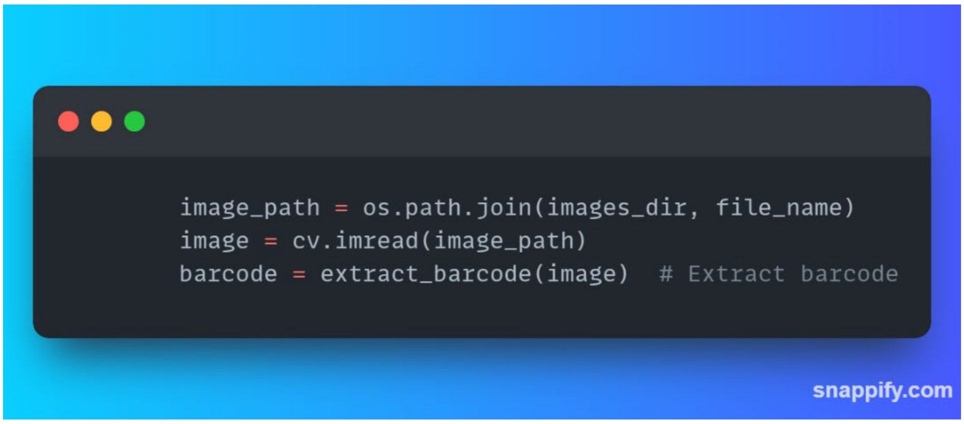
Iterate Over Files in Directory



```python
for file_name in os.listdir(images_dir):  # Iterate over files in the directory
    ext = os.path.splitext(file_name)[1].lower()  # Get the file extension
    if ext in IMAGE_FILES_EXTENSIONS:  # Check if the file is an image
        print(f"Processing: {file_name}")
```

The function iterates through all files in the specified images_dir. For each file, it checks if the extension matches a valid image format.

Load and Process Images

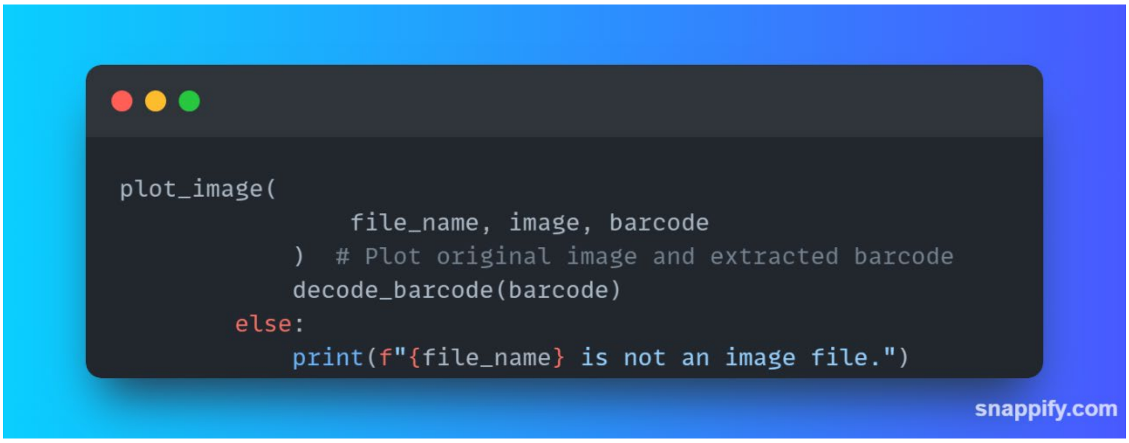


```
image_path = os.path.join(images_dir, file_name)
image = cv.imread(image_path)
barcode = extract_barcode(image)  # Extract barcode
```

snappify.com

If the file is a valid image, its full path is constructed, and the image is read using OpenCV. The barcode is then extracted using the extract_barcode function, which presumably isolates the barcode from the image.

Visualize Results, Decode the Barcode, and Handle Non-Image Files

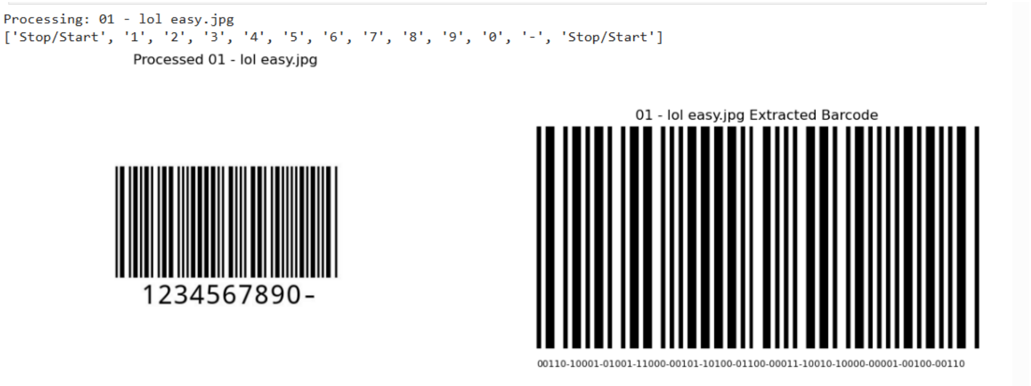


```
plot_image(
                file_name, image, barcode
            )  # Plot original image and extracted barcode
            decode_barcode(barcode)
        else:
            print(f"{file_name} is not an image file.")
```

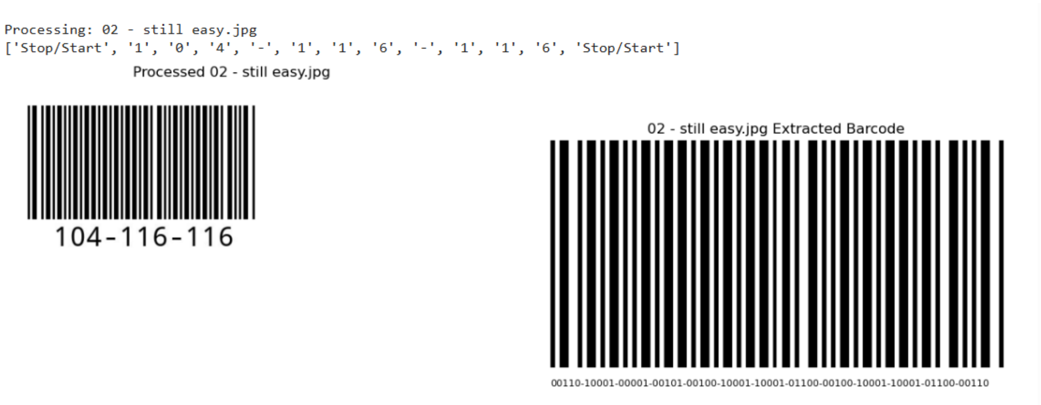
snappify.com

The plot_image function is called to display the original image alongside the extracted barcode. Then we extract the barcode then If a file in the directory is not an image (based on its extension), the function prints a message indicating the file is skipped.

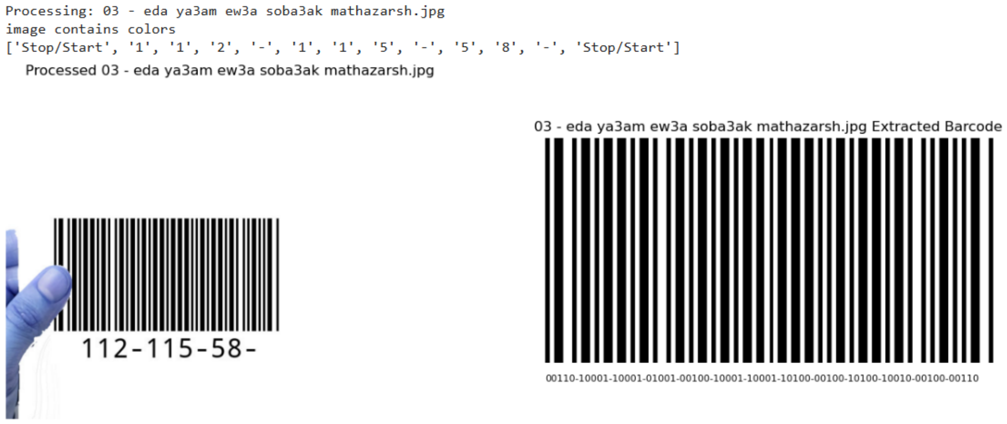
Results



```
Processing: 01 - lol easy.jpg
['Stop/Start', '1', '2', '3', '4', '5', '6', '7', '8', '9', '0', '-', 'Stop/Start']
```

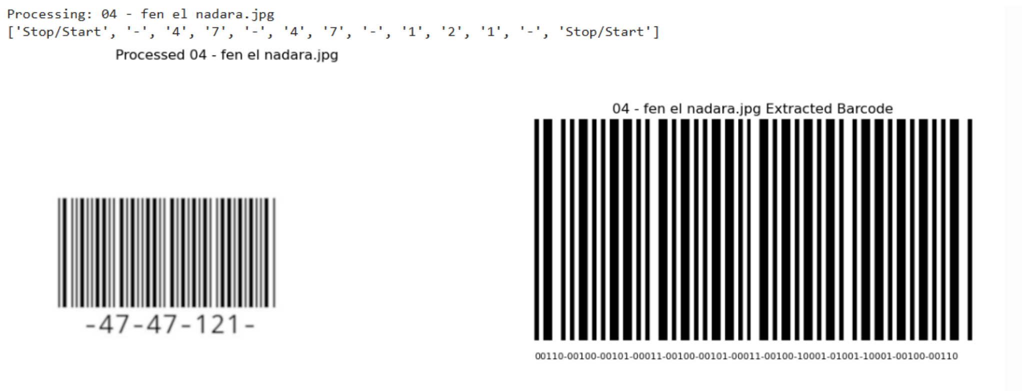


```
Processing: 02 - still easy.jpg
['Stop/Start', '1', '0', '4', '-', '1', '1', '6', '-', '1', '1', '6', 'Stop/Start']
```



```
Processing: 03 - eda ya3am ew3a soba3ak mathazarsh.jpg
image contains colors
['Stop/Start', '1', '1', '2', '-', '1', '1', '5', '-', '5', '8', '-', 'Stop/Start']
```

Processed 03 - eda ya3am ew3a soba3ak mathazarsh.jpg
112-115-58-
03 - eda ya3am ew3a soba3ak mathazarsh.jpg Extracted Barcode
00110-10001-10001-01001-00100-10001-10001-10100-00100-10100-10010-00100-00110

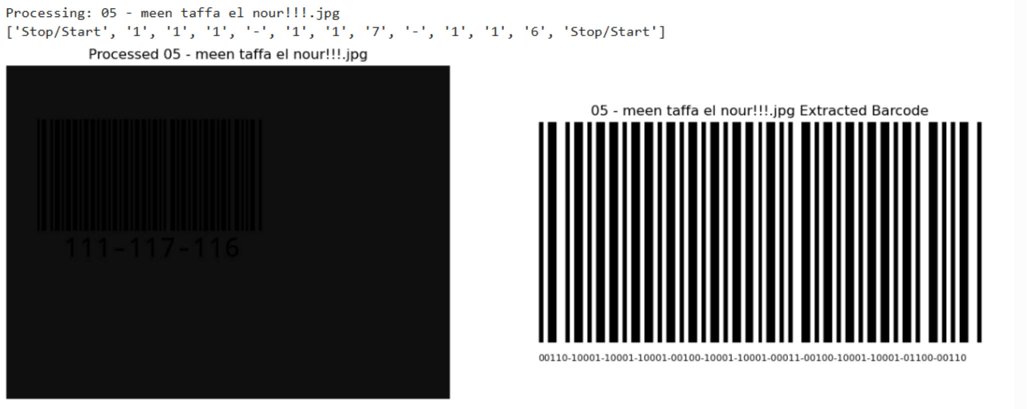


```
Processing: 04 - fen el nadara.jpg
['Stop/Start', '-', '4', '7', '-', '4', '7', '-', '1', '2', '1', '-', 'Stop/Start']
```

Processed 04 - fen el nadara.jpg

04 - fen el nadara.jpg Extracted Barcode

00110-00100-00101-00011-00100-00101-00011-00100-10001-01001-10001-00100-00110

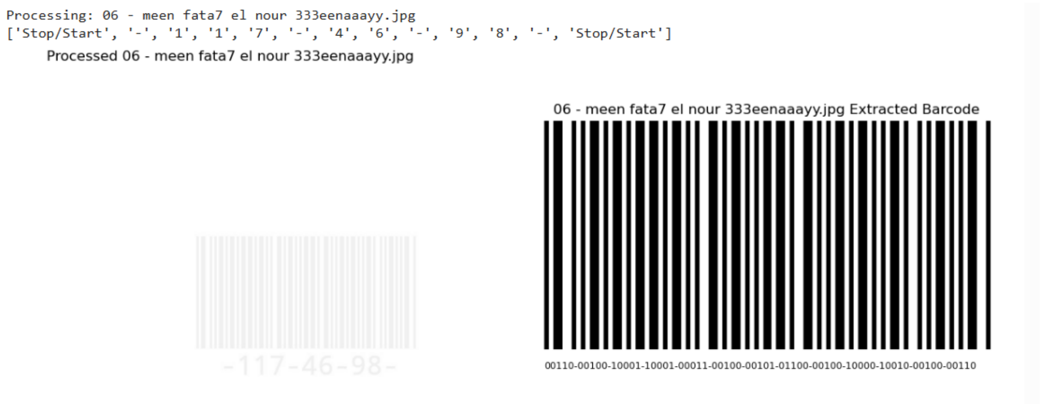


```
Processing: 05 - meen taffa el nour!!!.jpg
['Stop/Start', '1', '1', '1', '-', '1', '1', '7', '-', '1', '1', '6', 'Stop/Start']
```

Processed 05 - meen taffa el nour!!!.jpg

05 - meen taffa el nour!!!.jpg Extracted Barcode

00110-10001-10001-10001-00100-10001-10001-00011-00100-10001-10001-01100-00110

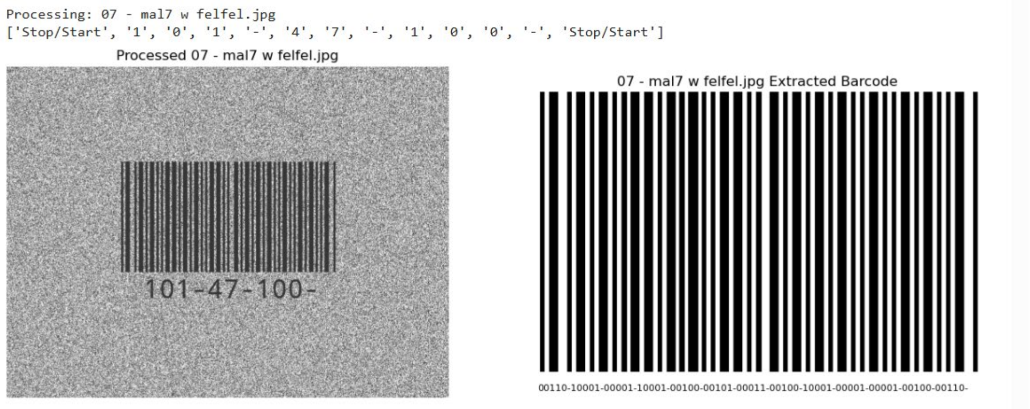


```
Processing: 06 - meen fata7 el nour 333eenaaayy.jpg
['Stop/Start', '-', '1', '1', '7', '-', '4', '6', '-', '9', '8', '-', 'Stop/Start']
```

Processed 06 - meen fata7 el nour 333eenaaayy.jpg

06 - meen fata7 el nour 333eenaaayy.jpg Extracted Barcode

00110-00100-10001-10001-00011-00100-00101-01100-00100-10000-10010-00100-00110

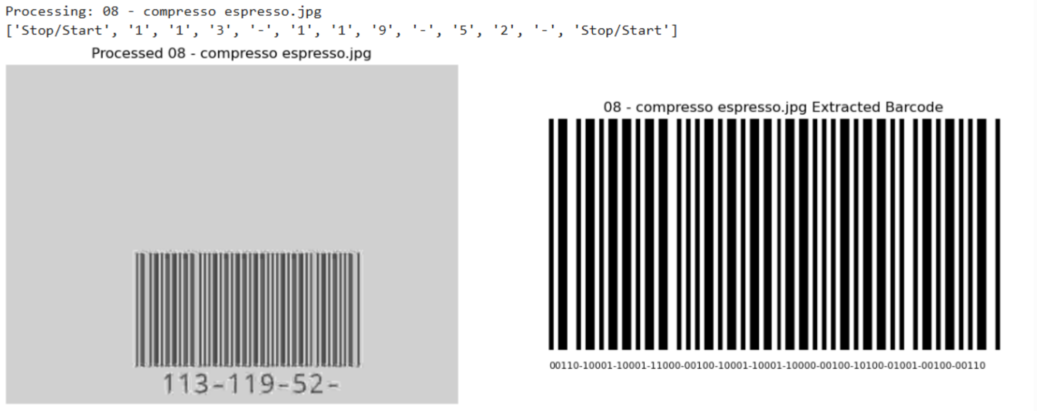


```
Processing: 07 - mal7 w felfel.jpg
['Stop/Start', '1', '0', '1', '-', '4', '7', '-', '1', '0', '0', '-', 'Stop/Start']
```

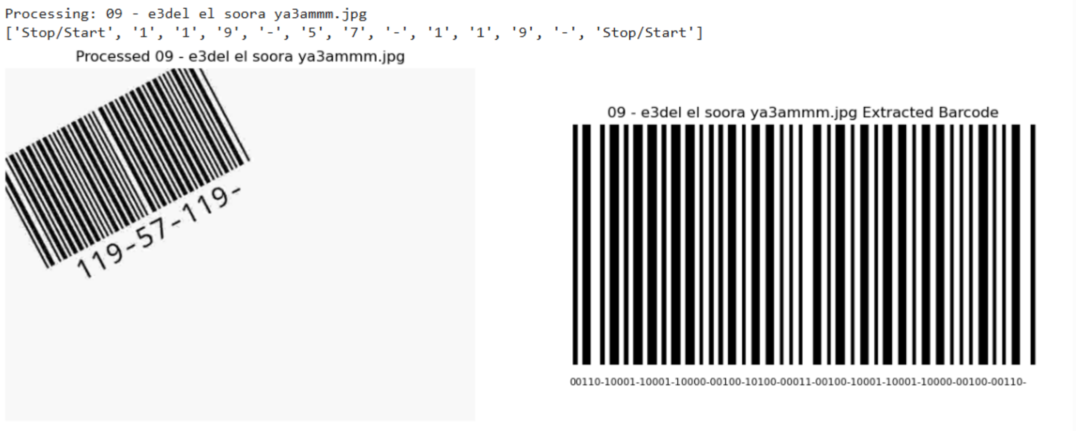
Processed 07 - mal7 w felfel.jpg

07 - mal7 w felfel.jpg Extracted Barcode

00110-10001-00001-10001-00100-00101-00011-00100-10001-00001-00001-00100-00110-

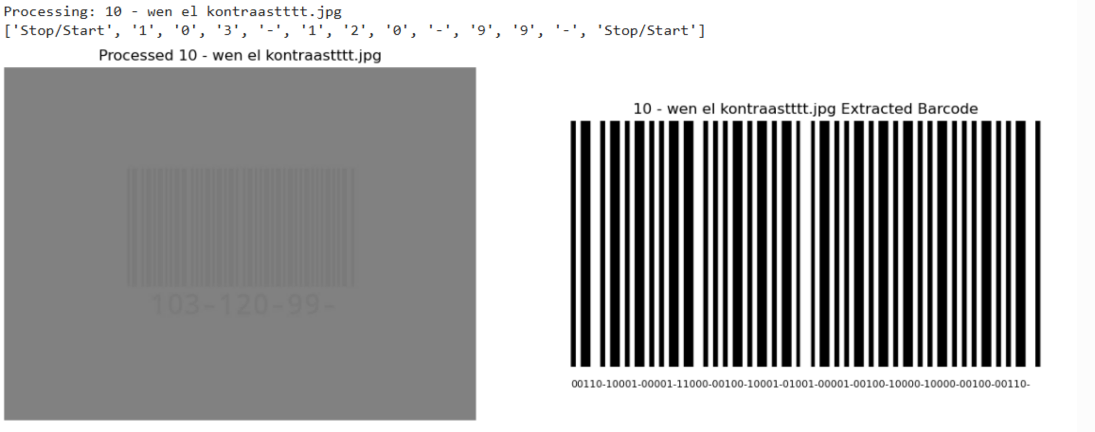


```
Processing: 08 - compresso espresso.jpg
['Stop/Start', '1', '1', '3', '-', '1', '1', '9', '-', '5', '2', '-', 'Stop/Start']
```

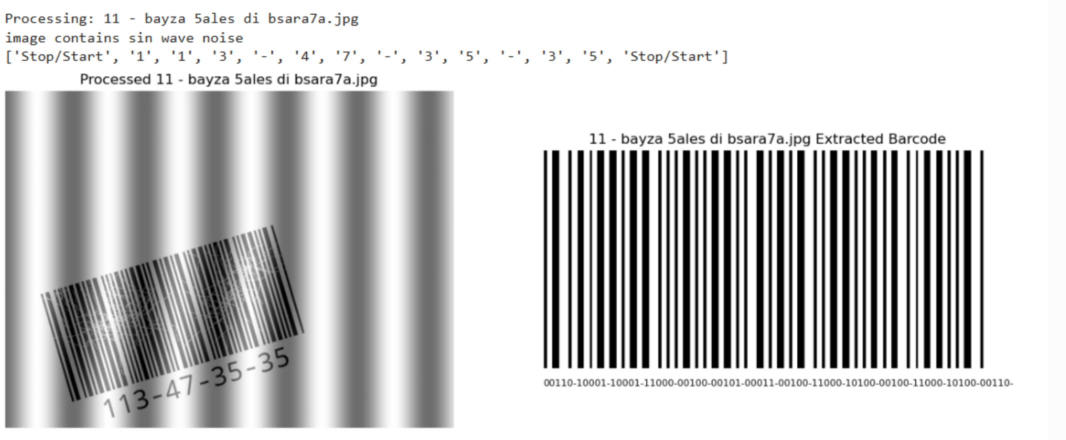


```
Processing: 09 - e3del el soora ya3ammm.jpg
['Stop/Start', '1', '1', '9', '-', '5', '7', '-', '1', '1', '9', '-', 'Stop/Start']
```

Processed 09 - e3del el soora ya3ammm.jpg
119-57-119-
09 - e3del el soora ya3ammm.jpg Extracted Barcode
00110-10001-10001-10000-00100-10100-00011-00100-10001-10001-10000-00100-00110-



```
Processing: 10 - wen el kontraastttt.jpg
['Stop/Start', '1', '0', '3', '-', '1', '2', '0', '-', '9', '9', '-', 'Stop/Start']
```



```
Processing: 11 - bayza 5ales di bsara7a.jpg
image contains sin wave noise
['Stop/Start', '1', '1', '3', '-', '4', '7', '-', '3', '5', '-', '3', '5', 'Stop/Start']
```

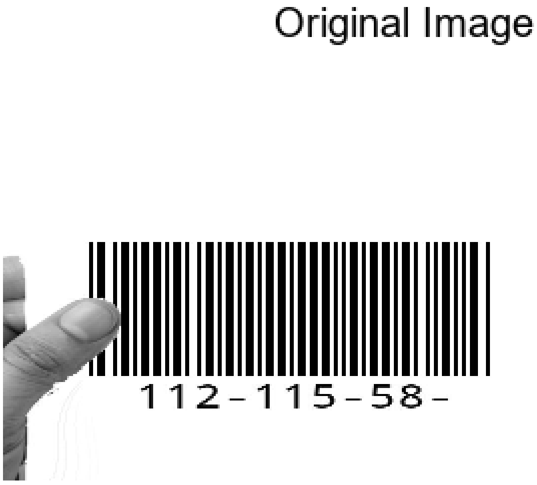
GIF (on provided test cases) of our pipeline 😊

GIFs are static here in the PDF. Go to the GitHub repo to view all of the GIFs

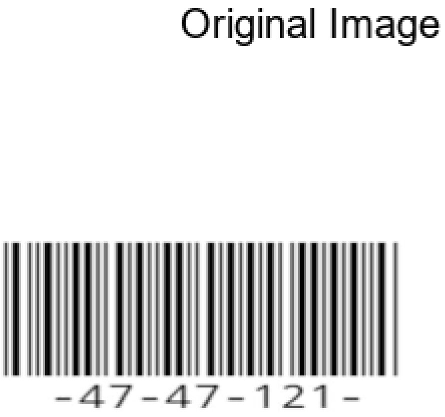
T7¿

T8¿

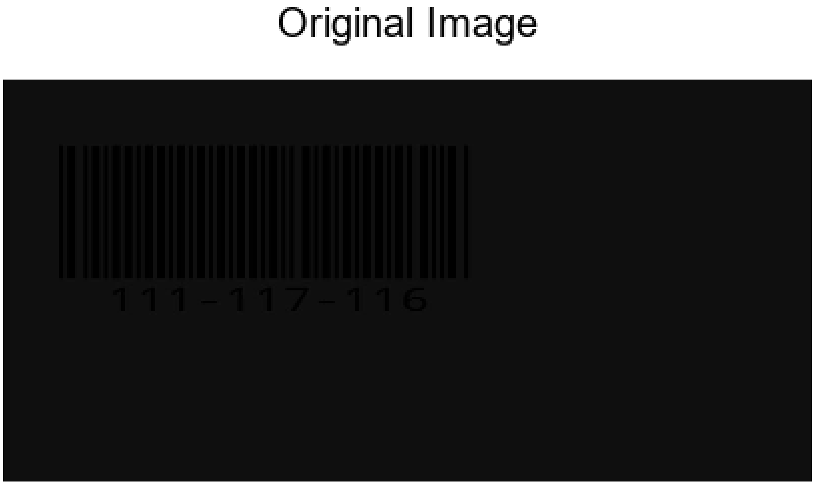
T9¿


Original Image
112-115-58-

T0¿


Original Image
-47-47-121-

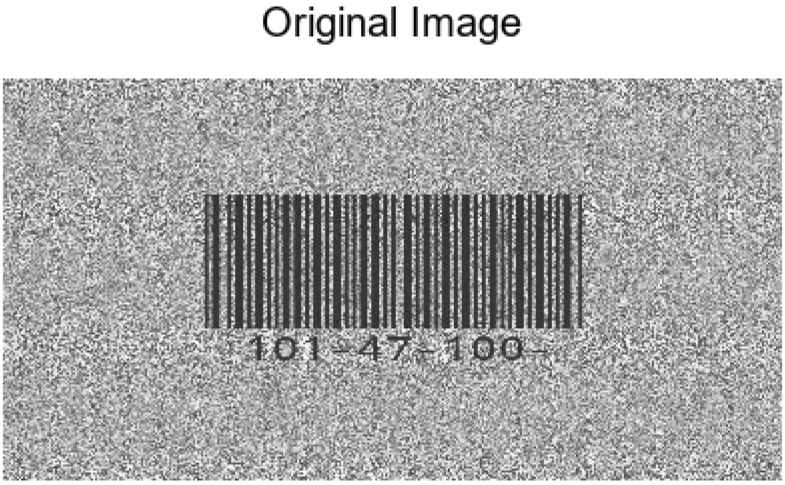
T❶¿


Original Image
111-117-116

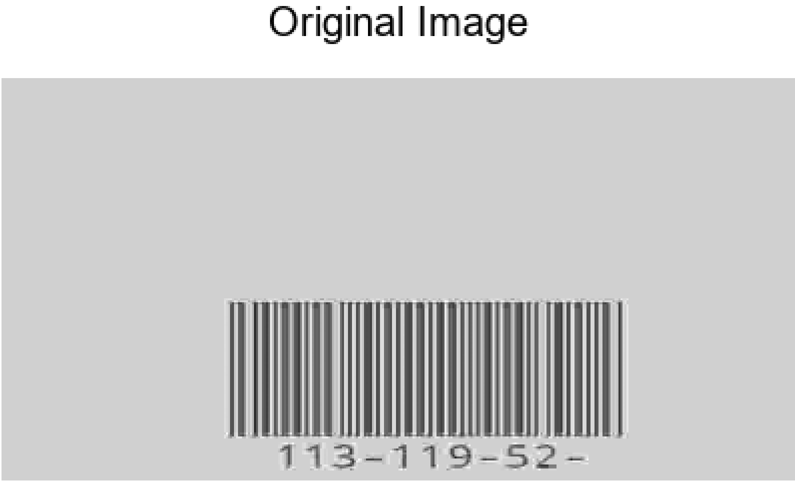
T❷¿

Original Image
-117-46-98-

T❸¿

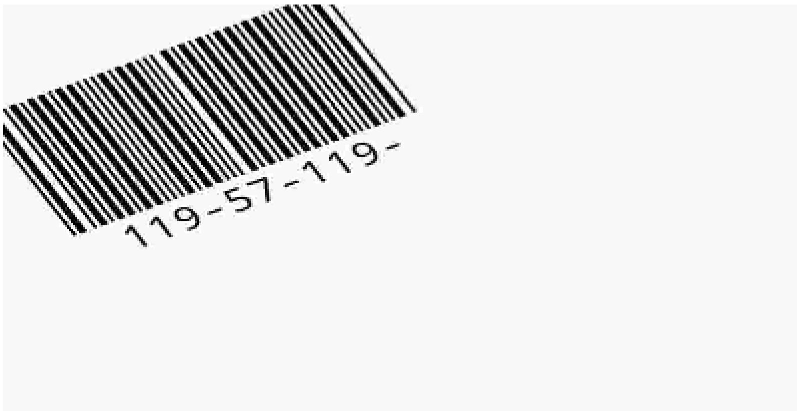

Original Image
101-47-100-

T❹¿


Original Image
113-119-52-

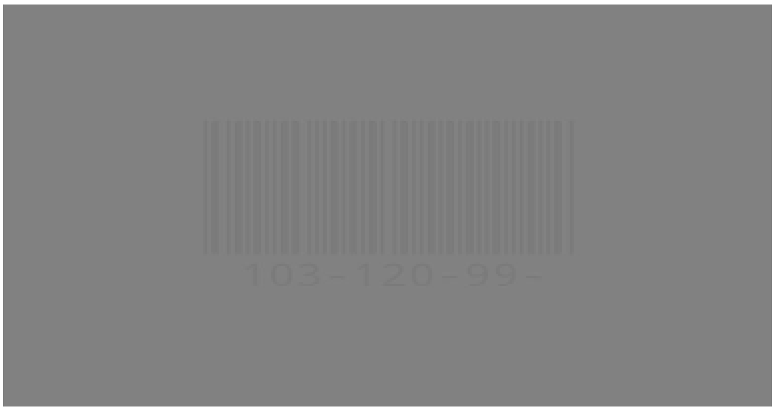
T❺¿

Original Image



T76¿

Original Image



T77¿

Why is our code generic?

Real time test to proof pipeline Robustness

First, let’s prove that our pipeline is generic with a quick GIF of our barcode scanner on a random photo we took of a random barcode in real-time 😊

Generic Features of the Pipeline

Preprocessing for Adaptability

- The preprocessing function handles both grayscale and colored images by detecting and removing non-gray colors.
- Thresholding and blurring reduce noise and enhance the barcode's features for subsequent contour detection.
- Sin wave detection ensures that patterns that look like lighting or interference (alternating lines or brightness issues) are addressed.

How.it's.generic:
It works for images with varying brightness, noise, and color schemes, ensuring consistent output for diverse inputs.

Contour Detection

- By using the largest contour, the pipeline assumes the barcode is the most prominent feature after preprocessing.
- The use of cv.morphologyEx and edge-detection techniques ensures robust handling of irregular or noisy barcodes.

How.it's.generic:
It dynamically adapts to the barcode’s size, orientation, and shape, making it suitable for barcodes on different surfaces or at different angles.

Vertical Erosion and Dilation

- Vertical morphological operations (closing, erosion, dilation) clean up vertical bars, ensuring even distorted or broken lines are reconstructed.

How.it's.generic:
This makes the pipeline effective for barcodes with missing lines, irregular spacing, or distortions due to compression.

Decoding with Tolerance

- The decoding algorithm uses a buffer and tolerance (TOLERANCE) to classify bar widths, allowing it to handle slight variations in bar sizes caused by printing or scanning inconsistencies.

How.it's.generic:
It accounts for minor imperfections in the barcode, ensuring accurate decoding even when the input is not perfectly aligned.

Adaptive Thresholding

- The preprocessing step employs Otsu's method and adaptive thresholding to dynamically adjust to contrast and brightness variations in the image.

How.it's.generic:
This allows the pipeline to work in varying lighting conditions without requiring manual threshold adjustment.

Handling Non-Image Files

- The file type check ensures only valid image files are processed, making the pipeline robust when applied to directories containing mixed file types.

How.it's.generic:
It avoids crashing or errors due to unsupported file types, ensuring smooth operation across different datasets.

Hyperparameters and Their Roles

Threshold Values (THRESH_BINARY, Otsu's Method, Adaptive Thresholding)

These values dynamically adjust to the image's properties, allowing for generic handling of different brightness levels.

Kernel Sizes

- Vertical kernels for morphological operations (e.g., (1, 16) and (1, 256)) ensure robust cleanup for vertical barcode lines.
- A (9, 9) kernel for contour detection closes gaps between edges to create a clean contour.

Why.they're.generic:
These kernel sizes are carefully chosen to work across most barcode sizes and resolutions while remaining flexible.

Bounding Box Logic

- The use of cv.minAreaRect ensures that even rotated barcodes are properly detected and transformed for further processing.

Why.it's.generic:
This allows the pipeline to work with barcodes scanned at arbitrary angles.

Contour Area Selection (max(contours, key=cv.contourArea))

- Selecting the largest contour ensures robustness when other artifacts are present in the image.

Perspective Transformation and Rotation

- The perspective transformation matrix (cv.getPerspectiveTransform) ensures warped barcodes are corrected.
- Automatic rotation to landscape orientation ensures consistent output regardless of the barcode’s original orientation.

Resilience in Processing Various Test Cases (for example these test cases)

Images with Colors

- The color detection and inpainting step remove distracting colors, ensuring the barcode is isolated.
  Test.case.handled: 03 - eda ya3am ew3a soba3ak mathazarsh.jpg.

Sin Wave Noise

- Adaptive thresholding handles lighting problems caused by periodic noise (e.g., sin wave patterns).
  Test.case.handled: 11 - bayza 5ales di bsara7a.jpg.

Low Contrast and Noisy Images

- Blurring, dilation, and erosion clean up noisy lines, ensuring robust barcode extraction.
  Test.case.handled: 06 - meen fata7 el nour 333eenaaayy.jpg.

Rotated or Tilted Barcodes

- The bounding box and perspective transformation correct skewed barcodes.
  Test.case.handled: 09 - e3del el soora ya3ammm.jpg.