

# Real Time Object Recognition with Voice-Guided Navigation for the Visually Impaired

Our project, Real-Time Object Recognition with Voice-Guided Navigation (RT-ORVGN) using OpenCV, aims to assist visually impaired individuals in navigating their surroundings safely and independently.

By leveraging computer vision and deep learning, the system processes real-time video input, detects objects with high accuracy, and provides voice-based guidance to enhance spatial awareness. Additionally, it estimates the distance between the user and objects, supports multiple languages for accessibility, and includes a personalized settings page for user preferences. With its advanced AI-driven approach, RT-ORVGN offers an efficient, user-friendly, and practical solution to improve mobility and independence for visually impaired individuals.

## Object Detection Voice Guide

**Code overview:** This program detects objects in real-time using a pre-trained SSD (Single Shot MultiBox Detector) MobileNet model, calculates their approximate distance from the camera based on known object dimensions, and provides real-time voice feedback about the object's location and distance. It uses OpenCV for image processing, pyttsx3 for voice synthesis, and threading to ensure smooth audio output.

## Libraries Used

1. **OpenCV (cv2):**
  - Open Source Computer Vision Library.
  - Used for capturing video from the webcam, processing images, and drawing bounding boxes around detected objects.
2. **NumPy:**
  - Fundamental Python library for numerical operations.
  - Used to reshape arrays and handle bounding box data.

3. **pyttsx3:**
  - ☐ Python Text-to-Speech library.
  - ☐ Enables voice feedback for detected objects' names, distances, and positions.
4. **Threading:**
  - ☐ A Python module for multithreading.
  - ☐ Used to run the voice feedback function in a separate thread to ensure the program runs smoothly without lag.
5. **Queue:**
  - ☐ A thread-safe way to share data between threads.
  - ☐ Used to queue objects and their details for voice feedback.
6. **Time:**
  - ☐ Used to add short delays in the voice feedback loop.

## Pre-trained Model

The program uses **SSD MobileNet V3** (Single Shot Detector with MobileNet backbone), which is a lightweight object detection model trained on the COCO dataset.

1. **Model Files:**
  - ☐ `ssd_mobilenet_v3_large_coco_2020_01_14.pbtxt` - Configuration file for the SSD MobileNet model.
  - ☐ `frozen_inference_graph.pb` - Pre-trained weights of the SSD MobileNet model.
2. **COCO Dataset (Common Objects in Context):**
  - ☐ A dataset containing 91 commonly found object classes such as person, car, bicycle, chair, etc.
  - ☐ Class names are stored in the `coco.names` file.

## Supporting File: `average_sizes.txt`

- Contains real-world average sizes (width in meters) of objects (e.g., cars, humans, chairs).
- Used for calculating approximate distances based on the object's bounding box width.

### Example Content:

Copy code

```
person,0.5  
car,1.7  
bottle,0.075
```

## Code Explanation

### 1. Loading Class Names and Average Sizes

- Reads class names from coco.names and real-world average object sizes from average\_sizes.txt into a dictionary.

python

```
with open(classFile, 'rt') as f:  
    classNames = f.read().rstrip('\n').split('\n')  
  
with open(average_sizes_file, 'rt') as f:  
    for line in f:  
        obj, size = line.strip().split(',')  
        average_sizes[obj.strip()] = float(size.strip())
```

### 2. Model Initialization

- Loads the SSD MobileNet V3 model and sets input parameters:
  - Input size: (320x320)
  - Normalization: Rescales input by dividing pixel values by 127.5.

python

Copy code

```
net = cv2.dnn_DetectionModel(weightsPath, configPath)  
net.setInputSize(320, 320)  
net.setInputScale(1.0 / 127.5)  
net.setInputMean((127.5, 127.5, 127.5))
```

```
net.setInputSwapRB(True)
```

### 3. Voice Feedback Using pyttsx3

- A separate thread handles text-to-speech processing to avoid slowing down the main loop.
- The Queue stores object details (label, distance, and position).

python

```
def speak(q):
    engine = pyttsx3.init()
    engine.setProperty('rate', 235)
    engine.setProperty('volume', 1.0)
    while True:
        if not q.empty():
            label, distance, position = q.get()
            rounded_distance = round(distance * 2) / 2
            engine.say(f"{label.upper()} is {rounded_distance}
meters to your {position}")
            engine.runAndWait()
            with queue.mutex:
                queue.queue.clear()
        else:
            time.sleep(0.1)
```

### 4. Distance Estimation

- Uses the formula for calculating distance based on the **focal length**, **real-world width**, and the object's bounding box width:

$$\text{Distance} = \frac{\text{Real Width} \times \text{Focal Length}}{\text{Width in Image}}$$

- Focal length is an approximate constant, and real-world sizes are read from `average_sizes.txt`.

python

Copy code

```
def calculate_distance(object_width, real_width):  
    return (real_width * focal_length) / (object_width + 1e-6)
```

## 5. Object Position

- Determines whether the object is to the **LEFT**, **FORWARD**, or **RIGHT** based on its horizontal position in the frame.

python

Copy code

```
def get_position(frame_width, box):  
    if box[0] < frame_width // 3:  
        return "LEFT"  
    elif box[0] < 2 * (frame_width // 3):  
        return "FORWARD"  
    else:  
        return "RIGHT"
```

## 6. Object Detection Loop

- Captures video from the webcam.
- Performs object detection using `net.detect()`.
- Filters results using **Non-Maximum Suppression (NMS)** to remove redundant bounding boxes.
- Calculates the object's distance and determines its position.
- Adds this information to the voice feedback queue.

python

```
classIds, confs, bbox = net.detect(img, confThreshold=thres)  
indices = cv2.dnn.NMSBoxes(bbox, confs, thres, nms_threshold)
```

```

if len(indices) > 0:
    for i in indices.flatten():
        # Draw bounding boxes and calculate distance/position
        label = classNames[class_id - 1].lower() distance =
        calculate_distance(w, average_sizes[label]) position =
        get_position(frame_width, (x, y, x + w, y +
h))
        queue.put((label, distance, position))

```

## 7. Displaying Results

- Draws bounding boxes and distance information on the video feed.
- Press q to exit the program.

Python

```

cv2.rectangle(img, (x, y), (x + w, y + h), color=(0, 255, 0),
thickness=2)
cv2.putText(img, text, (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX,
0.6, (0, 255, 0), 2)
cv2.imshow("Object Detection with Distance and Voice Guide",
img)

```

## Approach

1. **Object Detection:**
  - Uses SSD MobileNet for detecting objects in the video feed.
  - Applies NMS to improve detection accuracy.
2. **Distance Estimation:**
  - Calculates the distance to the detected object using the focal length and real-world dimensions.
3. **Voice Feedback:**

- ☐ Provides real-time voice alerts for object names, distances, and positions using pyttsx3.
- 4. **Real-Time Visualization:**
  - ☐ Draws bounding boxes and overlays distance data on the video feed.

## How to Run the Code

1. Ensure the following files are present in the working directory:
  - ☐ coco.names
  - ☐ ssd\_mobilenet\_v3\_large\_coco\_2020\_01\_14.pbtxt
  - ☐ frozen\_inference\_graph.pb
  - ☐ average\_sizes.txt

Install required libraries:

bash

Copy code

```
pip install opencv-python numpy pyttsx3
```

2.

Run the script:

```
Python obj_det_voice_guide.py
```

3.

4. Use the **webcam** to detect objects and listen to voice feedback.

## Conclusion

This program integrates object detection, distance estimation, and voice feedback into a seamless pipeline. It is useful for applications like assistive technologies for visually impaired individuals, smart surveillance systems, and real-time navigation aids.

# Text to Multiple Language Converter

## Code Overview

This application provides two core functionalities:

1. **Text-to-Speech (TTS) Conversion** – Converts text into speech using **Google Text-to-Speech (gTTS)** and serves the audio as an MP3 file.
2. **Machine Translation** – Translates text from one language to another using the **MBart-50 multilingual model from Hugging Face's Transformers library**.

It consists of two separate modules:

- **Flask API (speech.py)** – Handles text-to-speech conversion.
- **Translation Service (translation\_service.py)** – Loads a pre-trained translation model and translates text between languages.

## 1. Libraries and Models Used

### Flask

- Used to create a REST API to handle HTTP requests for text-to-speech conversion.
- **Blueprint** is used to modularize the API.

### gTTS (Google Text-to-Speech)

- Converts text into speech in multiple languages.
- Outputs an **MP3 audio file**.

### io

- Handles byte-stream operations for in-memory audio file storage.

### transformers (Hugging Face Library)

- Provides access to the **MBart-50** multilingual translation model.
- **Model Used:** facebook/mbart-large-50-many-to-many-mmt.



## **MBart50TokenizerFast**

- Tokenizes input text and converts it into a format that the translation model can process.

## **os**

- Manages file paths for saving and loading models locally.

## **2. How the App Works**

### **Text-to-Speech (TTS) API (speech.py)**

1. Accepts **POST** requests with a JSON payload containing:
  - text: The input text to be converted into speech.
  - language: The language code (e.g., "en" for English, "te" for Telugu).
2. Uses **Google Text-to-Speech (gTTS)** to generate speech from the text.
3. Streams the generated **MP3 audio file** back to the client.

### **Translation Service (translation\_service.py)**

1. Loads the **MBart-50** multilingual translation model from Hugging Face.
2. Accepts a text input along with source and target language codes.
3. Tokenizes the text and generates a translated output.
4. Returns the translated text to the user.

## **3. Step-by-Step Explanation of the Code**

### **Flask-based Text-to-Speech API (speech.py)**

from flask import Blueprint, request, jsonify

- **Flask Blueprint:** Helps modularize the application by grouping related routes.
- **request:** Used to get input data from the client.
- **jsonify:** Converts Python dictionaries into JSON responses.

```
from gtts import gTTS
import io
```

- **gTTS (Google Text-to-Speech)**: Converts text into speech.
- **io**: Used to handle in-memory file operations (storing the generated audio file as a byte stream).

```
bp = Blueprint('speech', __name__)
```

- **Creates a new Flask Blueprint** named 'speech' to handle TTS-related routes.

```
@bp.route('/speak', methods=['POST'])
def speak():
```

- Defines an **API endpoint** (/speak) that only accepts **POST** requests.

```
data = request.json
text = data.get('text', '')
language = data.get('language', 'en')
```

- Extracts text and language from the incoming JSON request.
- If text is missing, it defaults to an **empty string**.
- If language is missing, it defaults to **English (en)**.

```
language_map = {
    'en': 'en',
    'te': 'te',
    'hi': 'hi',
    'ja': 'ja',
    'zh': 'zh-cn',
    'es': 'es'
}
```

- Maps frontend language codes to **gTTS-supported language codes**.

```
tts = gTTS(text=text, lang=language_map.get(language, 'en'))
```

- Generates speech audio using gTTS in the selected language.
- If the requested language is not supported, it defaults to **English (en)**.

```
audio_io = io.BytesIO()
tts.write_to_fp(audio_io)
audio_io.seek(0)
```

- Creates an **in-memory file (BytesIO)** to store the generated MP3 audio.
- `write_to_fp()` writes the audio to the file, and `seek(0)` moves the cursor back to the beginning.

```
return audio_io.getvalue(), 200, {
    'Content-Type': 'audio/mpeg',
    'Content-Disposition': 'attachment; filename=speech.mp3'
}
```

- Returns the **MP3 file content** as a response.
- `Content-Type: audio/mpeg` ensures the browser interprets it as an audio file.
- `Content-Disposition: attachment; filename=speech.mp3` forces a download.

```
except Exception as e:
    print(f"Error in speak: {str(e)}")
    return jsonify({"error": str(e)}), 500
```

- Handles errors and returns a **500 Internal Server Error** if anything goes wrong.

## Translation Service (`translation_service.py`)

```
from transformers import MBartForConditionalGeneration, MBart50TokenizerFast
import os
```

- **MBartForConditionalGeneration**: Loads the **pre-trained MBart-50** model for translation.
- **MBart50TokenizerFast**: Tokenizer for preparing text input for the model.

- **os**: Used for file path handling.

```
class TranslationService:
```

- Defines a class to handle translation logic.

```
self.model_dir = os.path.join(os.path.dirname(os.path.dirname(__file__)), 'models',  
'mbart_model')
```

- **Defines a directory (models/mbart\_model)** to save the translation model locally.

```
if not os.path.exists(self.model_dir):  
    os.makedirs(self.model_dir, exist_ok=True)
```

- **Creates the directory if it does not exist.**

```
self.model =  
MBartForConditionalGeneration.from_pretrained("facebook/mbart-large-50-many-to-ma  
ny-mmt")  
self.tokenizer =  
MBart50TokenizerFast.from_pretrained("facebook/mbart-large-50-many-to-many-mmt")
```

- **Loads the MBart-50 model and tokenizer** from Hugging Face.

```
self.model.save_pretrained(self.model_dir)  
self.tokenizer.save_pretrained(self.model_dir)
```

- **Saves the model and tokenizer** locally for future use.

```
def translate(self, text, src_lang="en_XX", tgt_lang="te_IN"):
```

- Defines a function to **translate text** from a source language (src\_lang) to a target language (tgt\_lang).

```
if not text or not isinstance(text, str):  
    return {"error": "Invalid input text"}
```

- Ensures the input is a valid string.

```
encoded_text = self.tokenizer(text, return_tensors="pt", padding=True, truncation=True)
```

- **Tokenizes the input text** into a format the model understands.
- **return\_tensors="pt"** returns PyTorch tensors.

```
self.tokenizer.src_lang = src_lang  
forced_bos_token_id = self.tokenizer.lang_code_to_id[tgt_lang]
```

- **Specifies the source language** and forces the model to start the output in the target language.

```
generated_tokens = self.model.generate(  
    **encoded_text,  
    forced_bos_token_id=forced_bos_token_id,  
    max_length=128  
)
```

- **Generates translated text** using the MBart model.

```
translation = self.tokenizer.batch_decode(generated_tokens, skip_special_tokens=True)  
return {"translation": translation[0]}
```

- **Decodes the output tokens** back into readable text.
- Returns the translated text.

## Conclusion

This application provides **real-time translation** and **text-to-speech conversion** using state-of-the-art AI models. The **Flask API** serves the TTS functionality, while the **MBart-50 model** performs translation tasks efficiently.

# Currency Note Detector - Documentation Overview

## Code Overview

This Flask API processes uploaded images of currency notes and identifies them using a deep learning model. The API uses the **Pillow** library to handle image files, and a **pre-trained ResNet34 model** to classify currency notes. The detection logic is implemented in a separate service (CurrencyService), which loads the trained model and performs inference.

## Libraries Used

### 1. Flask (Web Framework)

- `Blueprint`: Organizes API routes in a modular way.
- `request`: Handles HTTP requests, allowing image uploads.
- `jsonify`: Converts Python dictionaries into JSON responses.

### 2. Pillow (PIL - Python Imaging Library)

- `Image.open()`: Loads image files.
- `convert('RGB')`: Ensures images are in RGB format before processing.

### 3. io (File Handling)

- `io.BytesIO()`: Handles image files in memory instead of writing them to disk.

### 4. Custom Currency Service (CurrencyService)

- Loads a **ResNet34** model trained on currency images.
  - Processes images and predicts the currency type.
-

# How the Code Works

## 1. API Endpoint (detect\_currency)

- `@currency_bp.route('/detect_currency', methods=['POST', 'OPTIONS'])`
- The endpoint supports two HTTP methods:
  - POST: Accepts image uploads for currency detection.
  - OPTIONS: Handles preflight requests for CORS (Cross-Origin Resource Sharing).

## 2. Handling CORS Preflight Requests

- `if request.method == 'OPTIONS':`
  - `response = jsonify({'status': 'ok'})`
  - `response.headers.add('Access-Control-Allow-Origin', '*')`
  - `response.headers.add('Access-Control-Allow-Headers', 'Content-Type')`
  - `response.headers.add('Access-Control-Allow-Methods', 'POST')`
  - `return response`
- Some browsers send a **preflight request** before making a POST request to verify if the server allows cross-origin access.
- The OPTIONS method responds with the necessary CORS headers to allow API access from any origin (\*).

## 3. Checking for an Uploaded Image

- `if 'image' not in request.files:`
- `return jsonify({"error": "No image provided"}), 400`
- If no image is uploaded, the API returns an error.

## 4. Processing the Uploaded Image

- `file = request.files["image"]`
- `image_bytes = file.read()`

- `image = Image.open(io.BytesIO(image_bytes))`

- Reads the uploaded image file and converts it into a PIL Image object.

## 5. Detecting Currency

- `result = currency_service.detect_currency(image)`

- Calls the `detect_currency()` method from `CurrencyService`, which uses the deep learning model to classify the currency.

## 6. Sending the Response

- `response = jsonify({"result": result})`
- `response.headers.add('Access-Control-Allow-Origin', '*')`
- `return response`

- Returns the detected currency type in JSON format.

## 7. Error Handling

- `except Exception as e:`
- `print(f"Error in detect_currency: {str(e)}")`
- `return jsonify({"error": str(e)}), 500`

- If an error occurs (e.g., invalid image format), it returns a `500 Internal Server Error`.

## Currency Service (CurrencyService)

This class is responsible for loading the deep learning model and performing inference.



## 1. Initializing the Model

- `class CurrencyService:`
  - `def __init__(self):`
  - `model_path = os.path.join(os.path.dirname(__file__),`  
`'../src/models/IC_ResNet34_9880.pth')`
  - `self.model = Inference(model_path)`
- 
- Loads a pre-trained **ResNet34** model from `IC_ResNet34_9880.pth`.
  - The model is wrapped inside the `Inference` class (from `src.inference.inference`).

## 2. Processing the Image

- `def detect_currency(self, image: Image.Image) -> str:`
  - `try:`
  - `if image.mode != 'RGB':`
  - `image = image.convert('RGB')`
- 
- Converts the image to **RGB mode** if it's not already in that format.
  - This ensures compatibility with the model.

## 3. Running Inference

- `self.model.run_image(image, show=False)`
  - `result = self.model.return_result()`
  - `return result`
- 
- Calls `run_image()` on the model to process the image.
  - `return_result()` retrieves the classification result (e.g., "Indian Rupee - ₹500").

## 4. Error Handling

- `except Exception as e:`
- `print(f"Error in currency detection: {str(e)}")`

- raise e

- If an error occurs, it prints and raises the exception.

## Step-by-Step Execution

### 1. Client Sends a Request

#### Example Request

A client sends a POST request with an image of a currency note.

- `curl -X POST http://127.0.0.1:5000/detect_currency \`
- `-H "Content-Type: multipart/form-data" \`
- `-F "image=@currency_note.jpg"`

- This request uploads `currency_note.jpg` to the API.

### 2. API Processes the Request

- Flask reads the image file.
- Converts it into a PIL Image object.
- Calls `CurrencyService.detect_currency()`.

### 3. Deep Learning Model Predicts Currency

- The **ResNet34 model** classifies the image.
- The predicted currency name (e.g., "Indian Rupee - ₹500") is returned.

### 4. API Sends a JSON Response

#### Example Response

- `{`
- `"result": "Indian Rupee - ₹500"`
- `}`

- The detected currency is returned to the client.

## Approach & Benefits

### 1. Modular Structure

- The API routes are defined separately ( `currency_bp` ) .
- The currency detection logic is encapsulated in `CurrencyService` .

### 2. Deep Learning-Based Detection

- Uses a **ResNet34** model, which is robust for image classification.
- Can recognize multiple currency denominations.

### 3. Efficient Image Handling

- Uses `io.BytesIO()` to process images in memory without writing them to disk.

### 4. CORS Support

- The API allows cross-origin requests, making it accessible from web applications.

## Possible Enhancements

### 1. Support for More Currencies

- Extend the model to detect multiple currencies from different countries.

### ● Confidence Score in Output

- {  
 "result": "Indian Rupee - ₹500",  
 "confidence": 98.7  
}

### 2.

- Modify `return_result()` to include a confidence score.

### 3. Asynchronous Processing

- Use Celery to queue and process requests asynchronously.

### 4. Web Interface

- Develop a React/Flask frontend to allow users to upload images via a browser.

## Conclusion

- This Flask API allows real-time currency detection using a ResNet34 model.
- It efficiently processes images and provides JSON responses.
- The architecture is scalable, modular, and extensible.

## Bar Code voice feedback of cost

### Code Overview

This application is a barcode scanner that retrieves product details from an online database (Open Food Facts API) and, if unavailable, searches Google for product information. The scanned barcode data is displayed on the screen, and if a product is found, its details (name, brand, category, image, etc.) are fetched. Additionally, the top Google search result for the barcode can be opened in a web browser.

It uses computer vision (OpenCV) to scan barcodes from a webcam, Pyzbar for barcode decoding, Requests to fetch product information from an API, and BeautifulSoup to scrape Google search results.

## 1. Libraries and Modules Used

### OpenCV (cv2)

- Used for real-time video capture from the webcam.
- Displays the video feed and detects barcodes in frames.

### Requests

- Used to send HTTP requests to Open Food Facts API to fetch product details.

### **Pyzbar**

- Decodes barcodes from images or video frames.
- Extracts barcode data (e.g., product UPC, EAN codes) .

### **Webbrowser**

- Opens the top Google search result in a web browser if the barcode is not found in Open Food Facts.

### **BeautifulSoup**

- Helps extract information from web pages (used for Google search results if needed).

### **Googlesearch-Python**

- Performs a Google search for the barcode when the product is not found in the Open Food Facts API.
- Retrieves the top search result for the scanned barcode.

## **2. How the App Works**

### **Barcode Scanning & Processing**

1. Captures live video from the webcam using OpenCV.
2. Detects and decodes barcodes from video frames using Pyzbar.
3. Extracts the barcode number and searches for product details.
4. Displays product information from Open Food Facts (name, brand, category, description, image, etc.).
5. If the product is not found, performs a Google search for additional details.
6. The top search result is displayed and opened in a web browser.
7. The user can exit the scanner by pressing ESC.

### 3. Step-by-Step Explanation of the Code

#### Importing Required Libraries

```
import cv2
```

```
import requests
```

```
import webbrowser
```

```
from pyzbar import pyzbar
```

```
from bs4 import BeautifulSoup
```

- cv2: Used for webcam access and displaying video frames.
- requests: Sends API requests to Open Food Facts.
- webbrowser: Opens Google search results in a browser.
- pyzbar: Decodes barcodes from images.
- BeautifulSoup: Parses HTML content for extracting product information.

```
try:
```

```
    from googlesearch import search
```

```
except ImportError:
```

```
    print("⚠️ 'googlesearch-python' module not found! Install it using: pip install  
googlesearch-python")
```

```
    search = None
```

- **Tries to import the Google search module.**
- If missing, it prints a warning message but allows the script to continue running.

#### Fetching Product Details from Open Food Facts API

```
def fetch_product_info(barcode_data):
```

```
api_url = f"https://world.openfoodfacts.org/api/v0/product/{barcode_data}.json"
```

```
response = requests.get(api_url)
```

- Constructs the API URL using the scanned barcode (barcode\_data).
- Sends an HTTP GET request to Open Food Facts to retrieve product information.

```
if response.status_code == 200:
```

```
    product_info = response.json()
```

```
    if 'product' in product_info:
```

```
        item = product_info['product']
```

- If the request is **successful (200 OK)**, the response is **converted to JSON**.
- If a **product is found**, it extracts the product details.

```
print("\n--- Product Details ---")
```

```
if 'product_name' in item:
```

```
    print(f"✅ Product Name: {item['product_name']}")
```

```
if 'brands' in item:
```

```
    print(f"✅ Brand: {item['brands']}")
```

```
if 'categories' in item:
```


```
    print(f"✅ Category: {item['categories']}")
```

```
if 'generic_name' in item:
```

```
    print(f"✅ Description: {item['generic_name']}")
```

```
if 'image_url' in item:
```

```
    print(f"✅ Image URL: {item['image_url']}")
```

- Prints the **product name, brand, category, description, and image URL**.
-  Indicates that the information was successfully retrieved.


```
return True # Product found
```

- If product details are found, it **returns True** to indicate success.

```
print("❌ Product not found in Open Food Facts database.")
```

```
fetch_from_google(barcode_data) # If not found, search Google
```

```
return False
```

- If the **product is not found**, it prints  and calls `fetch_from_google()` to search Google for additional details.

## Searching Google for Product Details

```
def fetch_from_google(barcode):
```

```
    if search is None:
```

```
        print("⚠️ Google search module not available. Install using: pip install  
googlesearch-python")
```

```
    return
```

- **Checks if the googlesearch module is available.**
- If missing, prints a warning message and **does not proceed with Google search**.

```
print("\n Searching Google for product details...")
```

```
search_query = f"Product with barcode {barcode}"
```



```
search_results = list(search(search_query))
```

- Creates a **Google search query** using the barcode number.
- Fetches search results and converts them into a **list**.

```
if search_results:
```

```
    google_url = search_results[0] # Get the first search result
```

```
    print(f"🔗 Top Google Result: {google_url}")
```

```
    webbrowser.open(google_url)
```

- If **results are found**, it selects the **first search result** and prints the URL.
- The **top result is automatically opened** in a web browser.

```
else:
```

```
    print("❌ No Google search results found.")
```

- If **no results are found**, it prints ❌.

## Detecting and Processing Barcodes

```
def read_barcodes(frame):
```

```
    barcodes = pyzbar.decode(frame)
```

```
    for barcode in barcodes:
```

- Uses **Pyzbar** to **detect barcodes in the frame**.

```
        barcode_text = barcode.data.decode('utf-8')
```

```
        print(f"\n Scanned Barcode: {barcode_text}")
```

```
fetch_product_info(barcode_text)
```

- Extracts and **decodes** the barcode data.
- Prints **the scanned barcode number**.
- Calls `fetch_product_info()` to retrieve product details.

```
x, y, w, h = barcode.rect
```

```
cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)
```

- **Draws a rectangle around the detected barcode** in the video feed.

## Opening the Webcam for Barcode Scanning

```
def main():
```

```
    camera = cv2.VideoCapture(0) # Open webcam
```

```
    ret, frame = camera.read()
```

```
    while ret:
```

```
        ret, frame = camera.read()
```

```
        frame = read_barcodes(frame)
```

```
        cv2.imshow(' Barcode Scanner', frame)
```

- Opens the **default webcam** (0).
- Continuously **reads frames** from the webcam.
- Calls `read_barcodes()` to **detect and process barcodes**.
- Displays the **video feed with barcode detection**.

```
if cv2.waitKey(1) & 0xFF == 27: # Press 'ESC' to exit
```

```
    break
```

- **Press 'ESC' to close the scanner.**

```
camera.release()
```

```
cv2.destroyAllWindows()
```

- **Releases the webcam and closes the window** after exiting.

## 4. Conclusion

This barcode scanner fetches product details from Open Food Facts and, if unavailable, searches Google for additional information. The application uses computer vision, barcode decoding, and web scraping to enhance product lookup capabilities. It provides real-time scanning, automatic Google searches, and seamless user interaction.