Step-by-Step Explanation

This code implements a document scanner that takes an image of a document (e.g., a piece of paper) and processes it to produce a clean, flat, scanned version. Below is a detailed explanation of each step, designed to be clear for a student new to image processing.

How It All Comes Together

The code takes an image of a document (e.g., a piece of paper on a table), identifies its boundaries, corrects its perspective to make it look flat, and enhances it to resemble a scanned document. Each step builds on the previous one:

- **Load and Resize**: Prepare the image for processing.
- **Grayscale and Edge Detection**: Simplify the image and find its edges.
- **Contour Detection**: Identify the document's shape.
- **Perspective Transform**: Flatten the document.
- **Thresholding**: Enhance the document for clarity.

Step 1: Setting Up the Environment

- **What's Happening**: Before running the code, you need to install the required Python libraries: opencv-python (for image processing), imutils (for resizing images), scikit-image (for thresholding), and numpy (for array operations).
- **Code**:

```
pip install opencv-python imutils scikit-image numpy
```

- **Explanation**: These libraries provide tools to manipulate images. OpenCV (cv2) is the main library for image processing, imutils simplifies tasks like resizing, scikit-image helps with advanced image filtering, and numpy handles numerical data like image pixel arrays.
- **Why It Matters**: Installing these libraries ensures you have the tools needed to process images effectively.

Step 2: Importing Libraries

- **What's Happening**: The code imports the necessary libraries and defines a custom order_points function to arrange the document's corner points.
- **Python Code**:

```
import cv2

import numpy as np

import imutils

from skimage.filters import threshold_local

from scipy.spatial import distance as dist
```

- **Explanation**:
    - cv2 is OpenCV for image processing tasks like reading images, converting colors, and detecting edges.
    - numpy (np) handles arrays, which represent images in this context.
    - imutils provides a simple function to resize images.
    - threshold_local from scikit-image applies adaptive thresholding to enhance the document's text.
    - scipy.spatial.distance is used in the order_points function to calculate distances between points.
- **Why It Matters**: These imports give you access to all the functions needed to process the image step-by-step.

Step 3: Loading and Resizing the Image

- **What's Happening**: The code loads an image (e.g., sample.jpg) and resizes it to a height of 500 pixels while maintaining the aspect ratio.
- **Python Code**:

```
image_path = 'images/sample.jpg'

original_img = cv2.imread(image_path)

copy = original_img.copy()

ratio = original_img.shape[0] / 500.0

img_resize = imutils.resize(original_img, height=500)
```

- **Explanation**:
    - cv2.imread loads the image from the specified path into a NumPy array.
    - copy creates a duplicate of the original image for later use in perspective transformation.
    - original_img.shape[0] gets the image's height in pixels. Dividing by 500 calculates the ratio to maintain the aspect ratio.
    - imutils.resize resizes the image to a height of 500 pixels, making it easier to process while keeping proportions intact.

- **Why It Matters**: Resizing reduces computational load and standardizes the image size for consistent processing. The copy preserves the original image for later steps.

Step 4: Converting to Grayscale

- **What's Happening**: The resized image is converted from color (RGB) to grayscale.
- **Python Code**:

```
gray_image = cv2.cvtColor(img_resize, cv2.COLOR_BGR2GRAY)

cv2.imshow("Grayscale Image", gray_image)

cv2.waitKey(0)
```

- **Explanation**:
    - cv2.cvtColor converts the image from BGR (OpenCV's default color format) to grayscale (cv2.COLOR_BGR2GRAY), reducing it to a single channel (intensity values).
    - Grayscale images are simpler to process because they have one value per pixel (0–255) instead of three (red, green, blue).
    - cv2.imshow displays the grayscale image, and cv2.waitKey(0) pauses execution until a key is pressed.
- **Why It Matters**: Grayscale simplifies subsequent steps like edge detection, as it reduces the complexity of the image data.

Step 5: Applying Gaussian Blur and Edge Detection

- **What's Happening**: The grayscale image is blurred to reduce noise, and then edges are detected using the Canny edge detector.
- **Python Code**:

```
blurred_image = cv2.GaussianBlur(gray_image, (5, 5), 0)

edged_image = cv2.Canny(blurred_image, 75, 200)

cv2.imshow("Edged Image", edged_image)

cv2.waitKey(0)
```

- **Explanation**:
    - cv2.GaussianBlur applies a 5x5 Gaussian blur to smooth the image, reducing noise that could interfere with edge detection.
    - cv2.Canny detects edges by identifying areas with significant intensity changes. The parameters 75 and 200 are thresholds for edge strength.
    - The result is a binary image (black background, white edges) highlighting the document's boundaries.

    - The edged image is displayed for visualization.

- **Why It Matters**: Edge detection helps identify the document's outline, which is crucial for finding its shape in the next step.

Step 6: Finding Contours and Identifying the Document

- **What's Happening**: The code finds contours (outlines) in the edged image, selects the largest ones, and identifies the document by looking for a quadrilateral (four-sided shape).

- **Python Code**:

```
cnts = cv2.findContours(edged_image, cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)

cnts = cnts[0] if len(cnts) == 2 else cnts[1]

cnts = sorted(cnts, key=cv2.contourArea, reverse=True)[:5]

doc = None

for c in cnts:

  peri = cv2.arcLength(c, True)

  approx = cv2.approxPolyDP(c, 0.02 * peri, True)

  if len(approx) == 4:

    doc = approx

    break
```

- **Explanation**:
    - cv2.findContours detects contours in the edged image. cv2.RETR_LIST retrieves all contours, and cv2.CHAIN_APPROX_SIMPLE simplifies their representation.
    - The code handles OpenCV version differences by checking the length of the returned tuple.
    - Contours are sorted by area (cv2.contourArea), and the top 5 largest are kept to focus on significant shapes.
    - For each contour, cv2.arcLength calculates its perimeter, and cv2.approxPolyDP approximates it to a simpler polygon. The 0.02 * peri parameter controls the approximation accuracy.
    - If a contour has 4 sides (len(approx) == 4), it's likely the document, so it's stored in doc and the loop breaks.

- **Why It Matters**: The document is typically a rectangle, so finding a four-sided contour helps isolate it from other shapes in the image.

Step 7: Visualizing the Document's Corners

- **What's Happening**: The code draws red circles at the detected document's corners to visualize them.
- **Python Code**:

```
if doc is not None:
    points = []
    for point in doc:
        tuple_point = tuple(point[0])
        cv2.circle(img_resize, tuple_point, 5, (0, 0, 255), -1)
        points.append(tuple_point)
    cv2.imshow("Corner Points", img_resize)
    cv2.waitKey(0)
```

- **Explanation**:
    - If a document contour (doc) is found, the code loops through its 4 points.
    - cv2.circle draws a red circle ((0, 0, 255) in BGR, with radius 5) at each corner on the resized RGB image.
    - The points are stored in a list for potential use and displayed for verification.
- **Why It Matters**: Visualizing the corners confirms that the correct document shape has been detected before proceeding.

Step 8: Ordering the Corner Points

- **What's Happening**: The order_points function arranges the document's 4 corner points in a consistent order: top-left, top-right, bottom-right, bottom-left.
- **Python Code**:

```
def order_points(pts):
    rect = np.zeros((4, 2), dtype="float32")
    s = pts.sum(axis=1)
    rect[0] = pts[np.argmin(s)]
    rect[2] = pts[np.argmax(s)]
    diff = np.diff(pts, axis=1)
    rect[1] = pts[np.argmin(diff)]
```

```python
    rect[3] = pts[np.argmax(diff)]
    return rect
```

- **Explanation**:
    - pts is an array of 4 points (x, y coordinates).
    - The function sums x + y for each point. The point with the smallest sum is top-left, and the largest sum is bottom-right.
    - It computes the difference (y - x) for each point. The smallest difference indicates top-right, and the largest difference indicates bottom-left.
    - The ordered points are returned as a 4x2 array.
- **Why It Matters**: Consistent ordering is necessary for the perspective transform to map the document correctly.

Step 9: Applying Perspective Transform

- **What's Happening**: The code uses the perspective_transform function to flatten the document into a top-down view.
- **Python Code**:

```python
def perspective_transform(image, pts, ratio):
    rect = order_points(pts)
    (tl, tr, br, bl) = rect
    widthA = np.sqrt(((br[0] - bl[0]) ** 2) + ((br[1] - bl[1]) ** 2))
    widthB = np.sqrt(((tr[0] - tl[0]) ** 2) + ((tr[1] - tl[1]) ** 2))
    maxWidth = max(int(widthA), int(widthB))
    heightA = np.sqrt(((tr[0] - br[0]) ** 2) + ((tr[1] - br[1]) ** 2))
    heightB = np.sqrt(((tl[0] - bl[0]) ** 2) + ((tl[1] - bl[1]) ** 2))
    maxHeight = max(int(heightA), int(heightB))
    dst = np.array([[0, 0], [maxWidth - 1, 0], [maxWidth - 1, maxHeight - 1], [0, maxHeight - 1]],
dtype="float32")
    M = cv2.getPerspectiveTransform(rect, dst)
    warped = cv2.warpPerspective(image, M, (maxWidth, maxHeight))
    return warped
```

- **Explanation**:
    - The function calls order_points to get the corners in order.
    - It calculates the document's width and height by computing Euclidean distances between points (e.g., bottom-right to bottom-left for width).

- The maximum width and height are used to define the output image size.
- A destination array (dst) defines the new image's corners (top-left at (0,0), etc.).
- cv2.getPerspectiveTransform computes a transformation matrix M to map the document's corners to the destination rectangle.
- cv2.warpPerspective applies the transformation to the original image, producing a flattened view.

- **Why It Matters**: This step corrects perspective distortions, making the document appear as if scanned flat.

Step 10: Enhancing the Scanned Document

- **What's Happening**: The warped image is converted to grayscale and thresholded to create a clean, black-and-white scanned effect.
- **Python Code**:

```
warped_gray = cv2.cvtColor(warped_image, cv2.COLOR_BGR2GRAY)

warped_threshold = threshold_local(warped_gray, 11, offset=10, method="gaussian")

warped_image = (warped_gray > warped_threshold).astype("uint8") * 255

cv2.imshow("Scanned Document", warped_image)

cv2.waitKey(0)
```

- **Explanation**:
  - cv2.cvtColor converts the warped image to grayscale.
  - threshold_local applies adaptive thresholding (using a 11x11 pixel neighborhood and Gaussian method) to enhance text by separating it from the background.
  - The result is a binary image (black text on white background) achieved by comparing pixel values to the threshold and scaling to 0 or 255.
  - The final image is displayed.
- **Why It Matters**: Thresholding enhances readability, mimicking the look of a scanned document.

Step 11: Cleaning Up

- **What's Happening**: The code closes all display windows.
- **Python Code**:

```
cv2.destroyAllWindows()
```

- **Explanation**: This ensures all OpenCV windows are closed, freeing up system resources.
- **Why It Matters**: Proper cleanup prevents memory leaks and keeps the program tidy.
