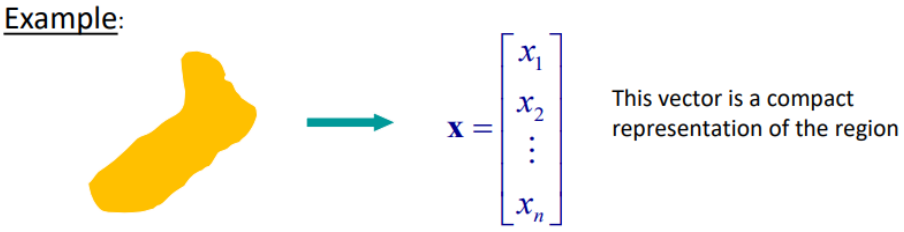
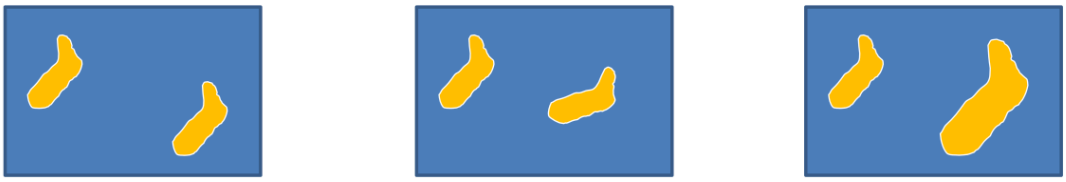
6.1 Shape descriptors

The main objective of region description is to obtain a mathematical representation of a segmented region from an image consisting of a vector of features $\mathbf{x} = [x_1, \ldots, x_n]$.



In this notebook we will see a branch of region description called **shape analysis**. Shape analysis aims to construct this feature vector using only shape features (e.g., size, perimeter, circularity or compactness).

Depending on the application, it could be needed that the used descriptor be **invariant** to the position in the image in which the regions appears, its orientation, and/or its size (scale). Some examples:



Position invariance | Orientation invariance | Scale invariance

This notebook **covers simple shape descriptors of regions** based on their area, perimeter, minimal bounding-box, etc (sections 6.1.1 and 6.1.2). We will also study **if these descriptors are invariant to position, orentation and size** (section 6.1.3). Let's go!

Problem context - Number-plate recognition

So here we are again! UMA called for us to join a team working on their parking access system. This time, they want to upgrade their obsolete number-plate detection algorithm by including better and more efficient methods.

Here is where our work starts, we are going to **apply shape analysis to each of the characters** that can appear on a license plate, that is, numbers from 0 to 9, and letters in the alphabet. The idea is to **produce a unique feature vector** for each character that could appear on a plate (e.g. $\mathbf{x}^0$, $\mathbf{x}^1$, . . ., $\mathbf{x}^\mathbf{A}$, $\mathbf{x}^\mathbf{B}$, etc.) so it could be later used to **train an automatic classification system** (we will see this in the next chapter!).

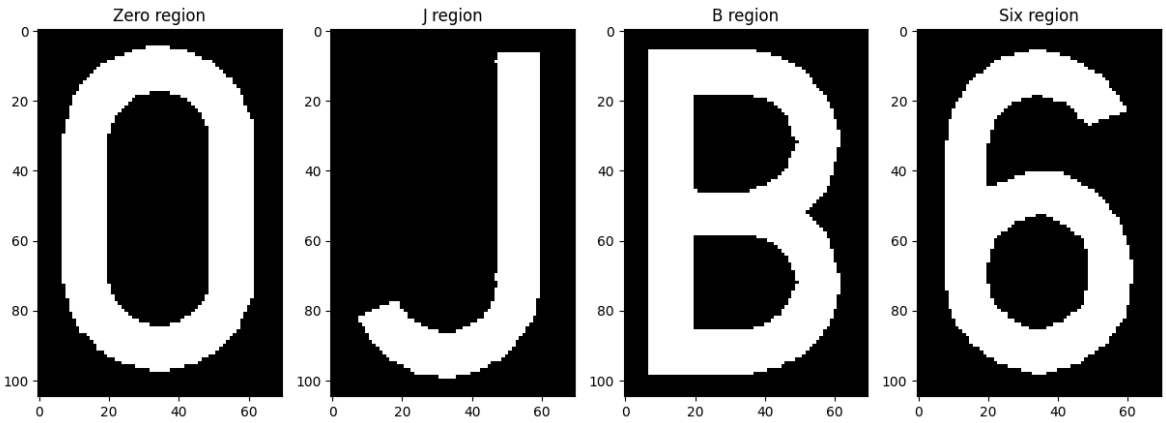
In [3]:
```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import matplotlib
matplotlib.rcParams['figure.figsize'] = (15.0, 8.0)

images_path = './images/'
```

Initial data

UMA's parking security team have sent us some segmented plate characters captured by their camera in the parking. They have binarized and cropped these images, providing us with regions representing such characters as white pixels. These cropped images are `region_0.png` (region with a zero), `region_6.png` (region with a six), `region_B.png` (region with a B), and `region_J.png` (region with a J).

Let's visualize them!

In [3]:
```
# Read the images
zero = cv2.imread(images_path + 'region_0.png',0)
J = cv2.imread(images_path + 'region_J.png',0)
B = cv2.imread(images_path + 'region_B.png',0)
six = cv2.imread(images_path + 'region_6.png',0)

# And show them!
plt.subplot(141)
plt.imshow(zero, cmap='gray')
plt.title('Zero region')

plt.subplot(142)
plt.imshow(J, cmap='gray')
plt.title('J region')

plt.subplot(143)
plt.imshow(B, cmap='gray')
plt.title('B region')

plt.subplot(144)
plt.imshow(six, cmap='gray')
plt.title('Six region');
```

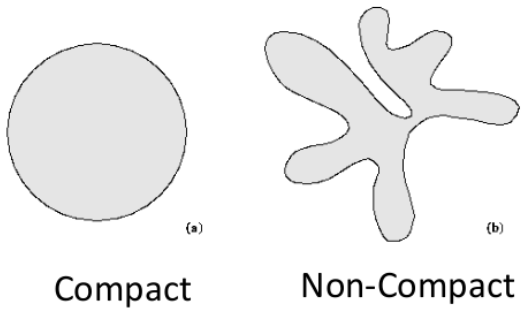


6.1.1 Compactness

The first feature we are going to work with is **compactness**:

$$\mathbf{compactness} = \frac{area}{perimeter^2}$$\[5pt]$

As you can see, this feature associates the area with the permeter of a region. Informally, it tells how *rounded* and *closed* is a region. The most compact shape is the circle, with $\mathbf{compactness} = 1/(4\pi)$.



OpenCV pill

OpenCV uses contours for analysing shapes. A contour is a list of points that defines a region. We can obtain the contours of a region using `cv2.findContours()`.

ASSIGNMENT 1: Computing compactness

What to do? Complete the function bellow, named `compactness()`, which computes the compactness of an input region.

For that, we are going to use the `cv2.findContours()` function, which takes as input:

- A binary image (containing the region as white pixels).
- Contour retrieval mode, it can be:
    - `RETR_EXTERNAL`: only returns the external contour

  - `RETR_LIST` : returns all contours (e.g. the character 0 would contain two contours: external and internal)
  - `RETR_CCOMP` : returns all contours and organize them in a two-level hierarchy. At the top level, there are external boundaries of the components. At the second level, there are boundaries of the holes.

- Method: controls how many points of the contours are being stored, this is for optimization purposes.

  - `CHAIN_APPROX_NONE` : stores absolutely all the contour points.
  - `CHAIN_APPROX_SIMPLE` : compresses horizontal, vertical, and diagonal segments and leaves only their end points.
  - `CHAIN_APPROX_TC89_L1` : applies an optimization algorithm.

And returns:

- a list containing the contours,
- and a list containing information about the image topology. It has as many elements as the number of contours.

For simplicity, we are going to take into account **only the external boundary** (as if the regions have not holes), so the second output is not relevant.

Having the contours, you can obtain the **area** and the **perimeter** of the region through `cv2.contourArea()` and `cv2.arcLength()`. Both functions take the contours of the region as input.

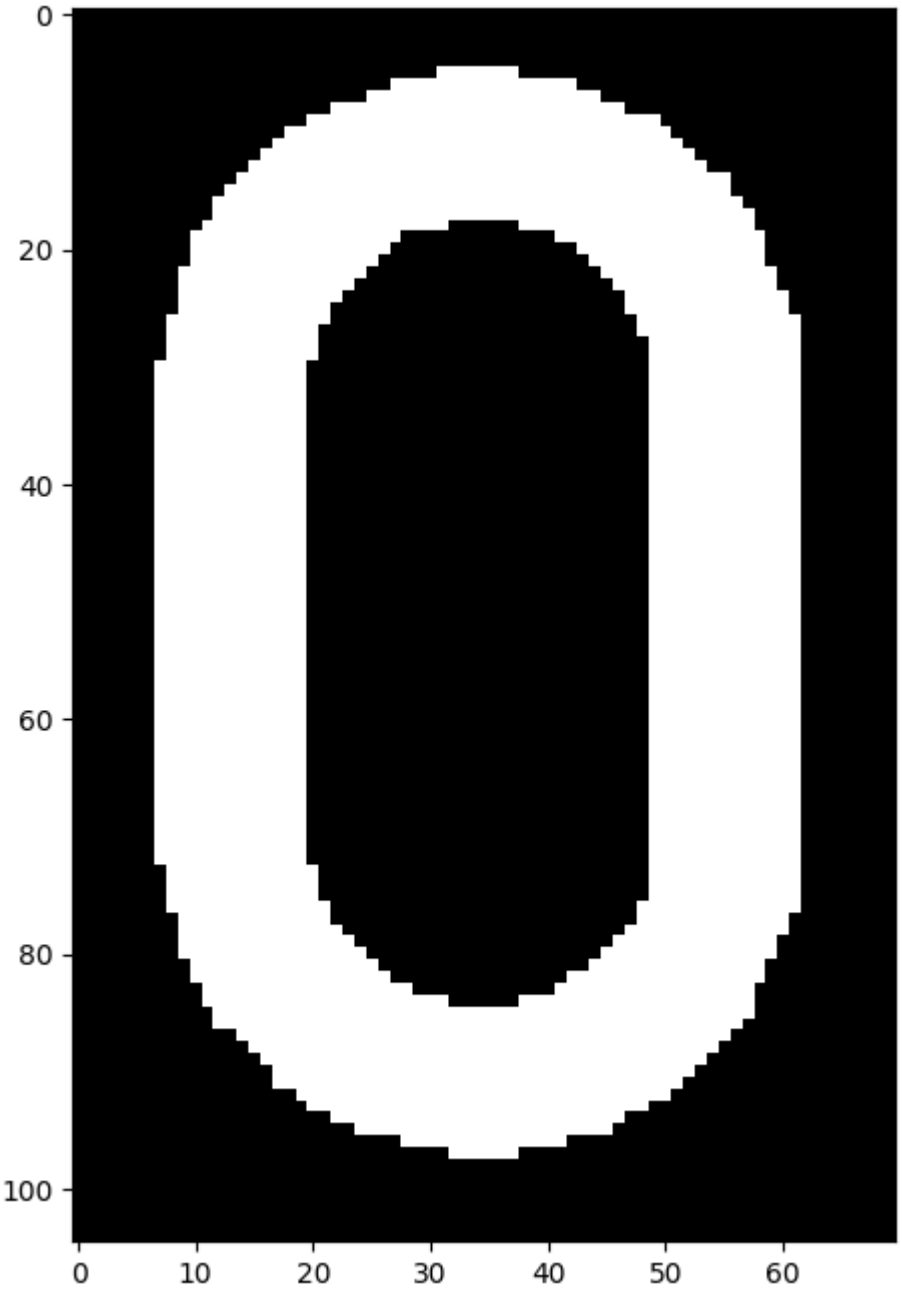
Note: Use `cv2.RETR_EXTERNAL` and `cv2.CHAIN_APPROX_NONE`.

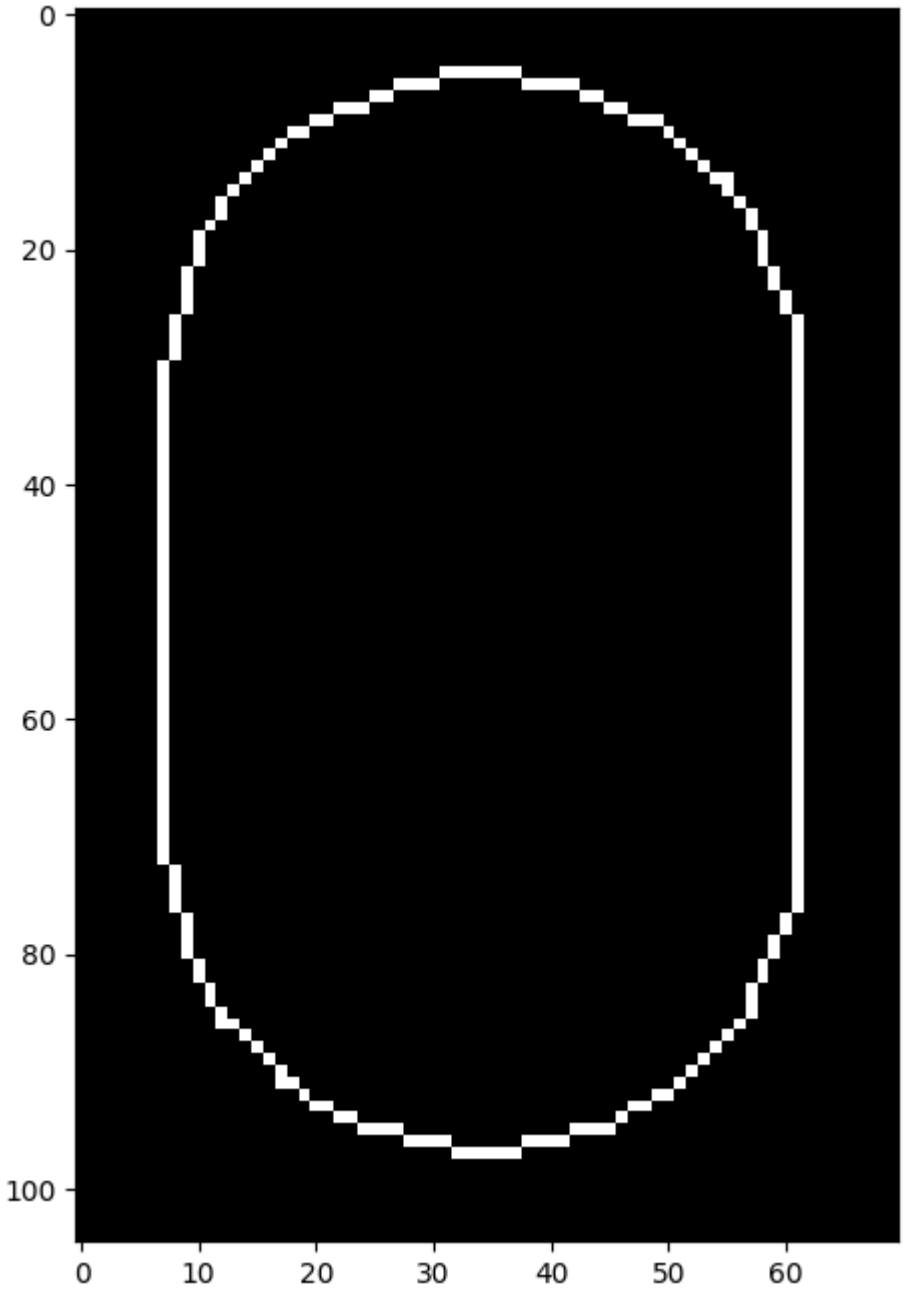
In [11]:
```
# Assignment 1
def compactness(region):
    """ Compute the compactness of a region.

        Args:
            region: Binary image

        Returns:
            compactness: Compactness of region (between 0 and 1/4pi)
    """
    plt.imshow(region,cmap='gray')
    plt.show()
    # Get external contour
    contours,_ = cv2.findContours(region, cv2.RETR_EXTERNAL, cv2.CHAIN_AP
    cnt = contours[0]

    img_contours = np.zeros(region.shape)
    # draw the contours on the empty image
    cv2.drawContours(img_contours, contours, -1, (255,255,255), 1)
    plt.imshow(img_contours,cmap='gray')
    plt.show()

    # Calcule area
    area = cv2.contourArea(cnt)
```

```
    # Calcule perimeter
    perimeter = cv2.arcLength(cnt,True)

    print("Area:",area)
    print("Perimeter:", perimeter)

    # Calcule compactness
    compactness = area / (perimeter ** 2)

    return compactness
```

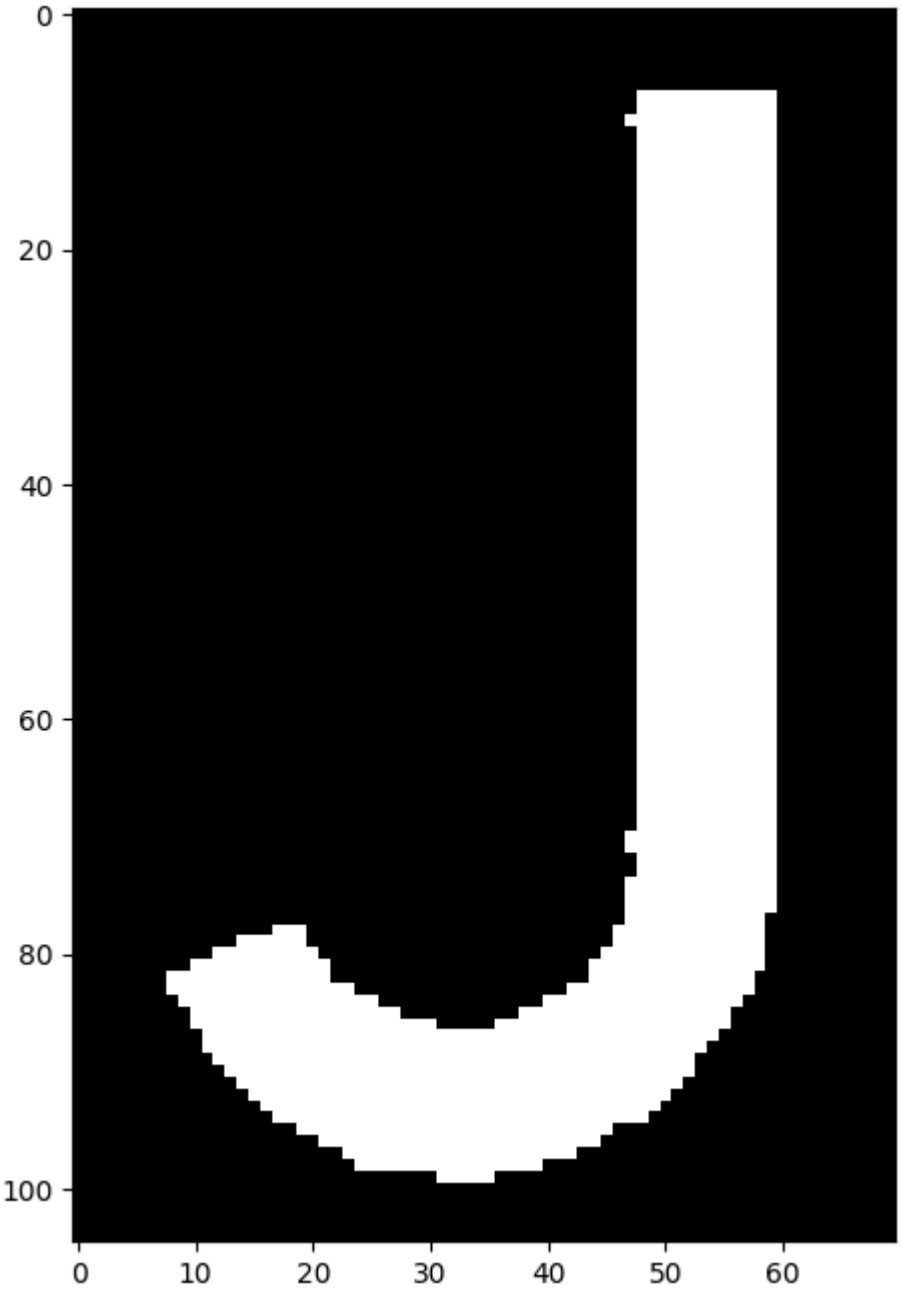
You can use next code to **test if the results are right**:

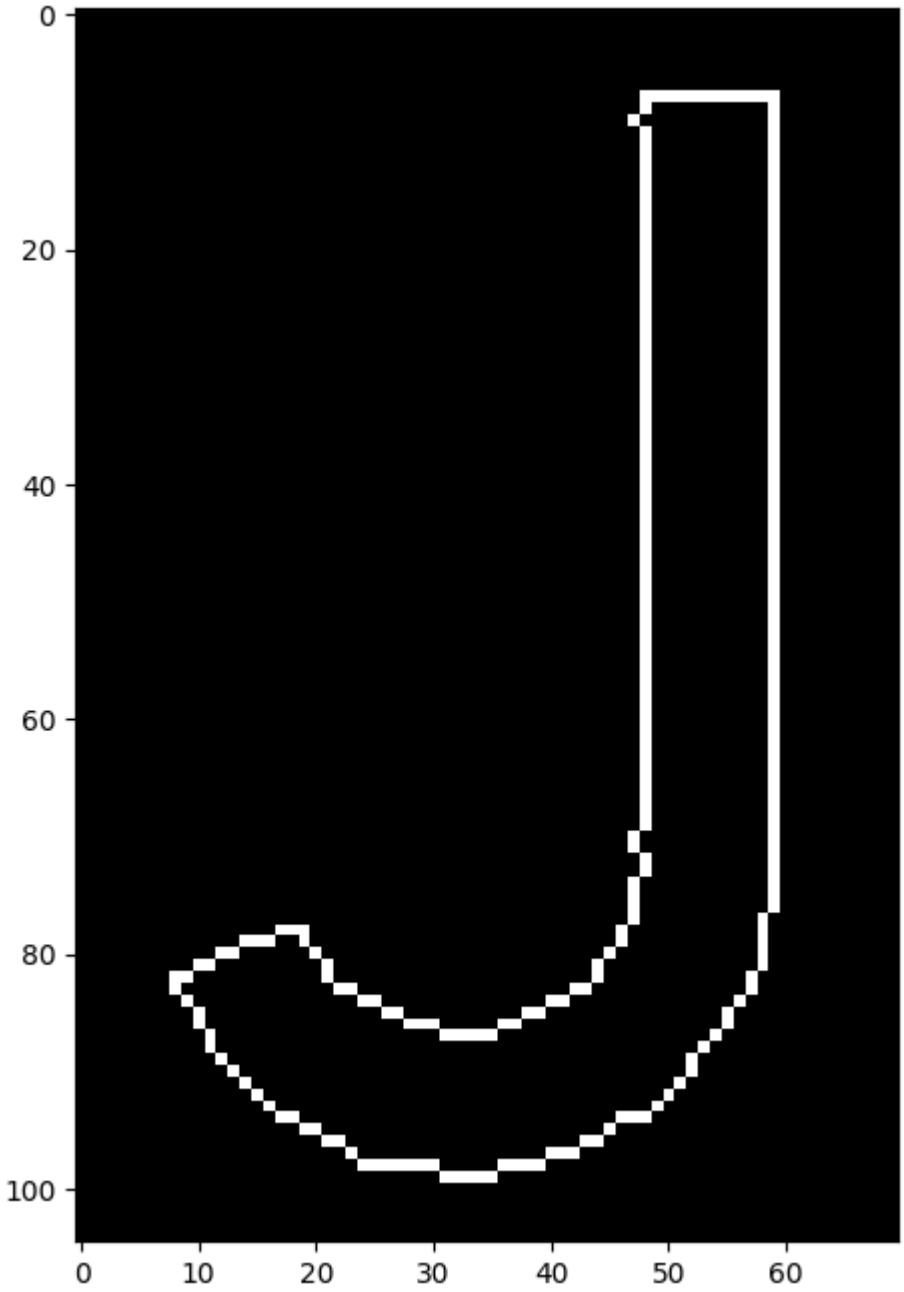
In [7]:
```
# Read the images
zero = cv2.imread(images_path + 'region_0.png',0)
J = cv2.imread(images_path + 'region_J.png',0)
B = cv2.imread(images_path + 'region_B.png',0)
six = cv2.imread(images_path + 'region_6.png',0)

# And show their compactness!
print(" Compactness of 0: ", round(compactness(zero),5), "\n",
      "Compactness of J: ", round(compactness(J),5), "\n",
      "Compactness of B: ", round(compactness(B),5), "\n",
      "Compactness of 6: ", round(compactness(six),5))
```


0
20
40
60
80
100
0
10
20
30
40
50
60

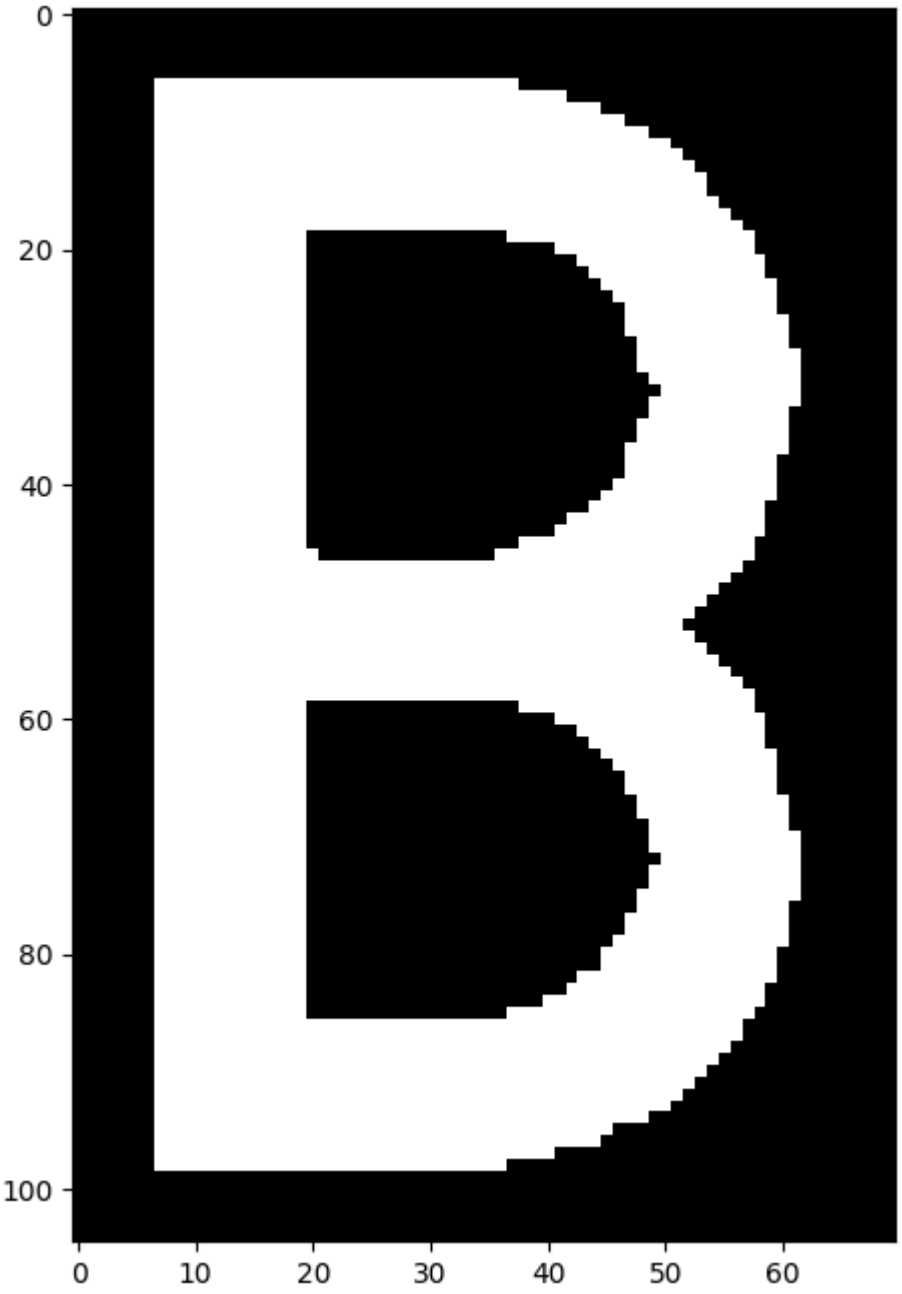


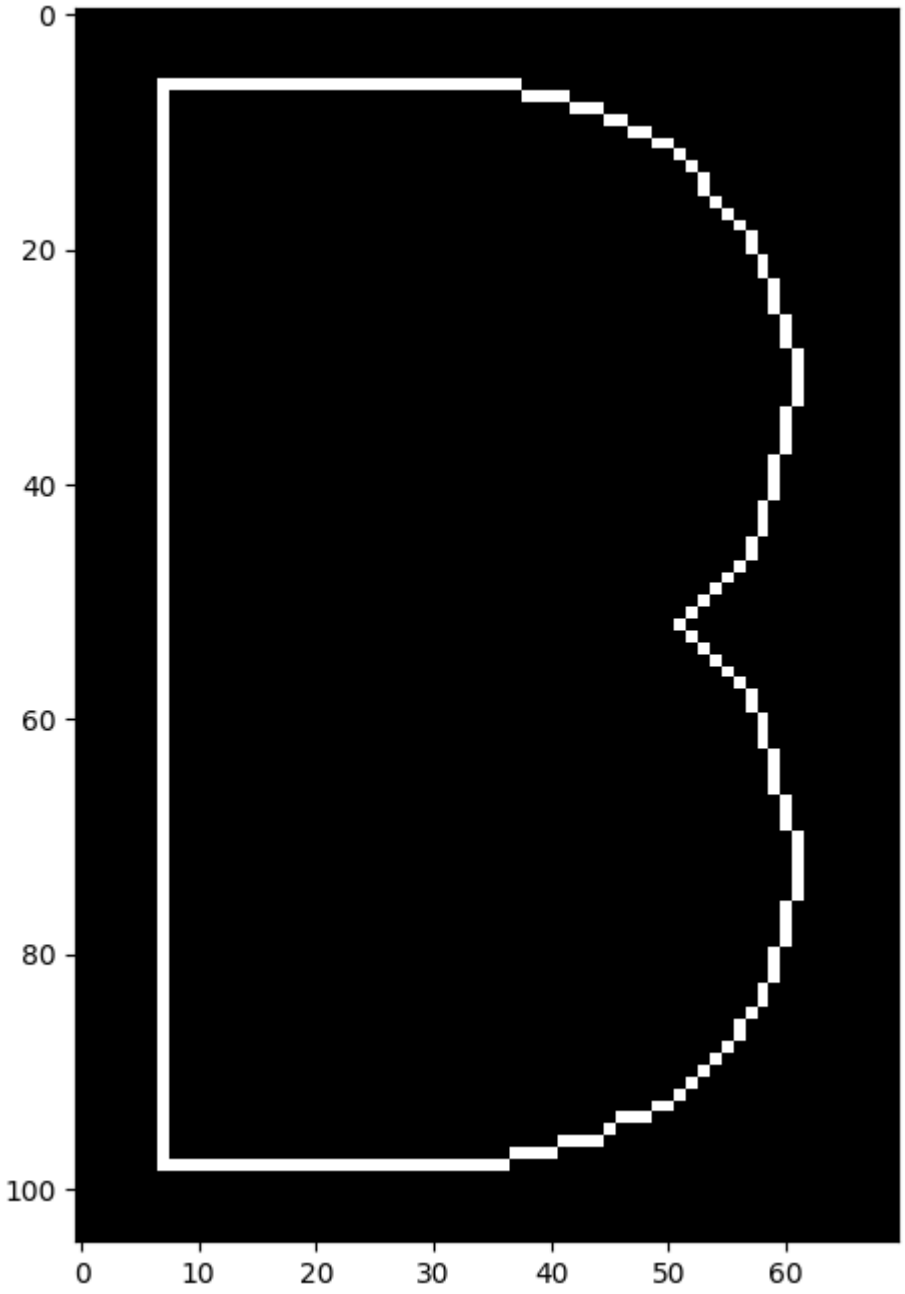

```
Area: 4307.0
Perimeter: 255.68123936653137
```


0
20
40
60
80
100
0
10
20
30
40
50
60

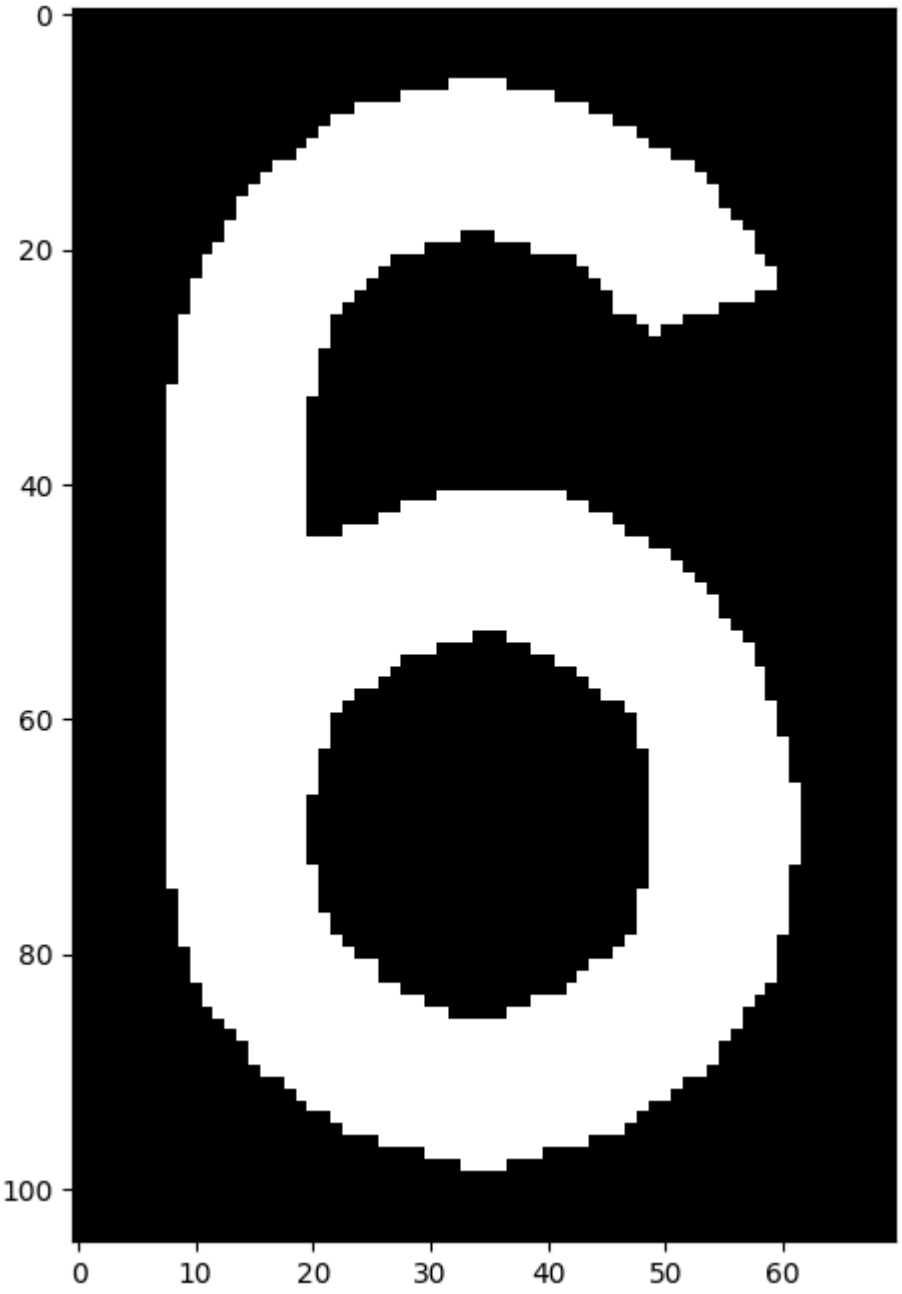


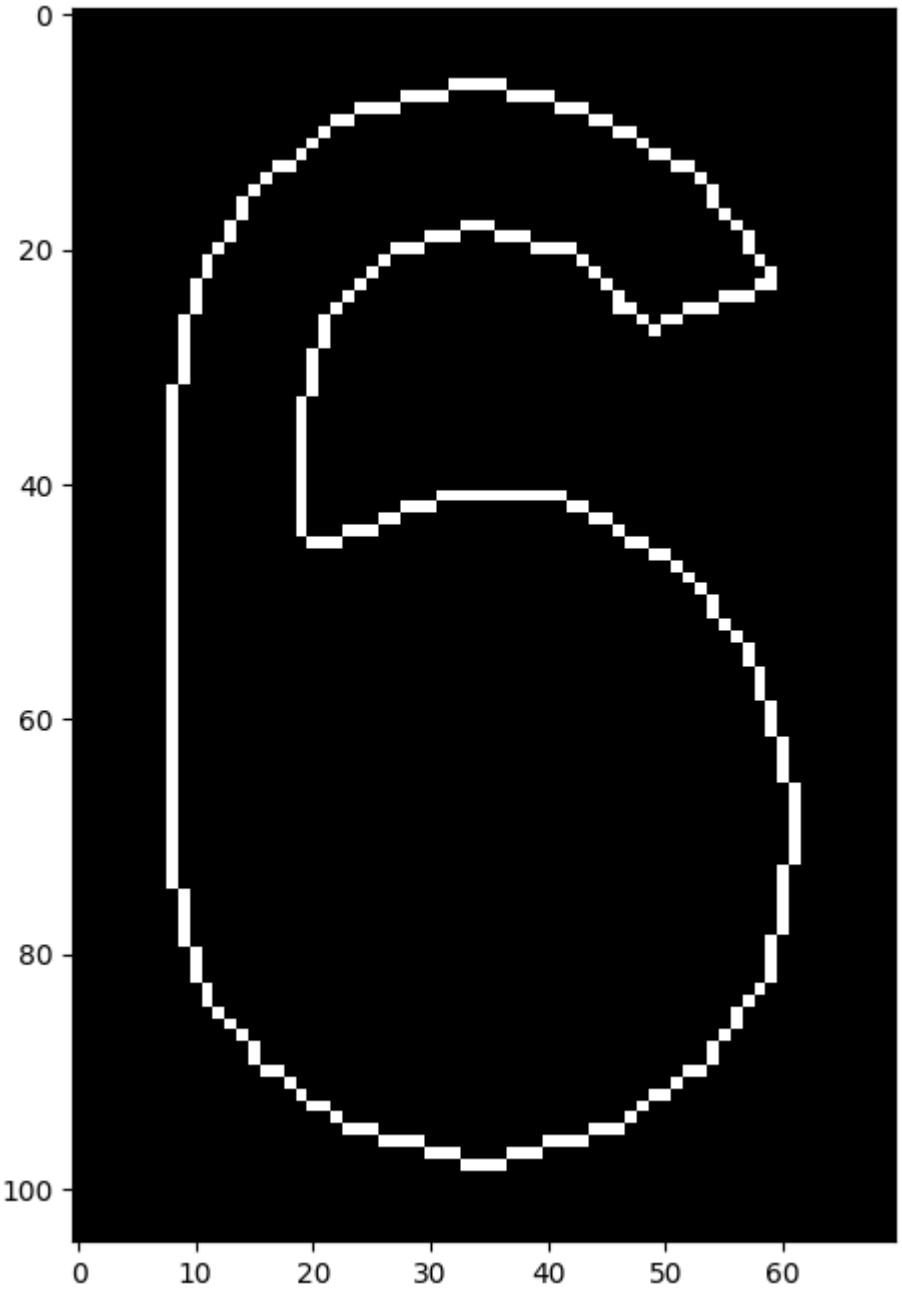

```
Area: 1386.0
Perimeter: 276.3675310611725
```


0
20
40
60
80
100
0
10
20
30
40
50
60




```
Area: 4498.0
Perimeter: 281.53910398483276
```


0
20
40
60
80
100
0
10
20
30
40
50
60




```
Area: 3217.5
Perimeter: 334.4924215078354
 Compactness of 0:  0.06588
 Compactness of J:  0.01815
 Compactness of B:  0.05675
 Compactness of 6:  0.02876
```

Expected output (using `CHAIN_APPROX_NONE`):

```
Compactness of 0:  0.06588
Compactness of J:  0.01815
Compactness of B:  0.05675
Compactness of 6:  0.02876
```

Thinking about it (1)

Excellent! Now, **answer the following questions:**

- Why `region_0.png` have the greatest compactness?

because it represents a shape that is closer to a perfect circle. The compactness metric, calculated is maximized for shapes that have a high area relative to their perimeter. Since a circle is the shape that most efficiently encloses an area, it tends to have the highest compactness among various shapes, which is why region_0.png, resembling a circular shape, has the highest compactness.

- Could we differentiate all characters using only this feature as feature vector?

  While compactness is a useful feature, it is unlikely to be sufficient by itself for distinguishing all characters. Characters with similar shapes, such as "O" and "0" or "6" and "8," may exhibit similar compactness values despite being different. This is because compactness does not capture specific shape details like internal structures or stroke patterns, which are often crucial for distinguishing characters. Therefore, additional features like eccentricity, symmetry, or topological descriptors would likely be needed for accurate differentiation.

- Is compactness invariant to position, orientation or scale?

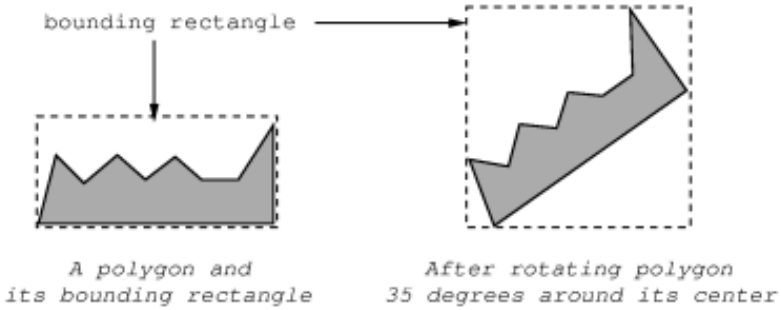
  Compactness is invariant to position and orientation because shifting or rotating the region does not change the perimeter or area. However, compactness is not invariant to scale: as a region scales up or down, the area and perimeter change at different rates, which impacts the compactness. To achieve scale invariance, other normalization techniques or scale-invariant features would need to be used.

6.1.2 Extent

Another shape descriptor is **extent** of a shape:

$$\textbf{extent} = \frac{area}{bounding\ rectangle\ area} \backslash [5pt]$$

This feature associates the area of the region with the area its bounding rectangle. A **bounding rectangle** can be defined as the minimum rectangle that contains all the pixels of a region whose bottom edge is horizontal and its left edge is vertical.



The shape with the highest extent value is the rectangle, with $extent = 1$, while the lowest one is an empty region so $extent = 0$.

ASSIGNMENT 2: Time to compute the extent

Complete the function `extent()`, which receives the `region` to be described as input and returns its `extent`.

Tip: compute the bounding rectangle using cv2.boundingRect(), which also takes the contours as input.

In [8]:
```python
def extent(region):
    """ Compute the extent of a region.

        Args:
            region: Binary image

        Returns:
            extent: Extent of region (between 0 and 1)
    """

    # Get external contour
    contours,_ = cv2.findContours(region,cv2.RETR_CCOMP,cv2.CHAIN_APPROX_

    cnt = contours[0]

    # Calcule area
    area = cv2.contourArea(cnt)

    # Get bounding rectangle
    _,_,w,h = cv2.boundingRect(cnt)

    # Calcule bounding rectangle area
    rect_area = w*h

    # Calcule extent
    extent = float(area) / rect_area

    return extent
```

You can use next code to **test if the obtained results are correct**:

In [9]:
```python
# Read the images
zero = cv2.imread(images_path + 'region_0.png',0)
J = cv2.imread(images_path + 'region_J.png',0)
B = cv2.imread(images_path + 'region_B.png',0)
six = cv2.imread(images_path + 'region_6.png',0)

# And show their extent!
print("Extent of 0: ", round(extent(zero),5), "\n",
      "Extent of J: ", round(extent(J),5), "\n",
      "Extent of B: ", round(extent(B),5), "\n",
      "Extent of 6: ", round(extent(six),5))
```

```
Extent of 0:  0.84203
 Extent of J:  0.2866
 Extent of B:  0.87937
 Extent of 6:  0.64068
```

Expected output (using `CHAIN_APPROX_NONE`):

```
Extent of 0:  0.84203
Extent of J:  0.2866
```

```
Extent of B:  0.87937
Extent of 6:  0.64068
```

Thinking about it (2)

Now, **answer the following questions:**

- Why `region_B.png` have the greatest extent?

  The region_B.png image has the greatest extent because its shape closely approximates a rectangle, allowing it to use most of the area within its bounding rectangle. Extent is defined as the ratio between the area of the shape and the area of its bounding rectangle. Since the letter "B" fills its bounding rectangle more fully compared to other shapes, it achieves a higher extent value.

- Is extent invariant to position, orientation or scale? If not, how could we turn it into a invariant feature?

  Extent is invariant to the position of the shape within the image since it only depends on the area of the shape and its bounding rectangle, not on the location. However, extent is not invariant to orientation or scale.

  - ***Orientation**: If the shape rotates, its bounding rectangle may increase in size to contain the new orientation, altering the extent value.*
  - ***Scale**: If the shape scales up or down, the areas of both the shape and its bounding rectangle change proportionally, but potential rounding errors in pixel values may introduce slight variations in extent.*

  *To make extent invariant to orientation, we could use a **minimum-area bounding box** (a rotated rectangle that tightly fits the region regardless of its orientation) rather than a horizontal-vertical bounding rectangle.*

6.1.3 Building a feature vector

Now that we can compute two different features, compactness (x_1) and extent (x_2), we can build a feature vector ($\mathbf{x}$) for characterizing each region by concatenating both features, that is, $\mathbf{x} = [x_1, x_2]$.

Before sending to UMA our solution for region description, let's see if these features are discriminative enough to differentiate between the considered characters.

ASSIGNMENT 3: Plotting feature vectors

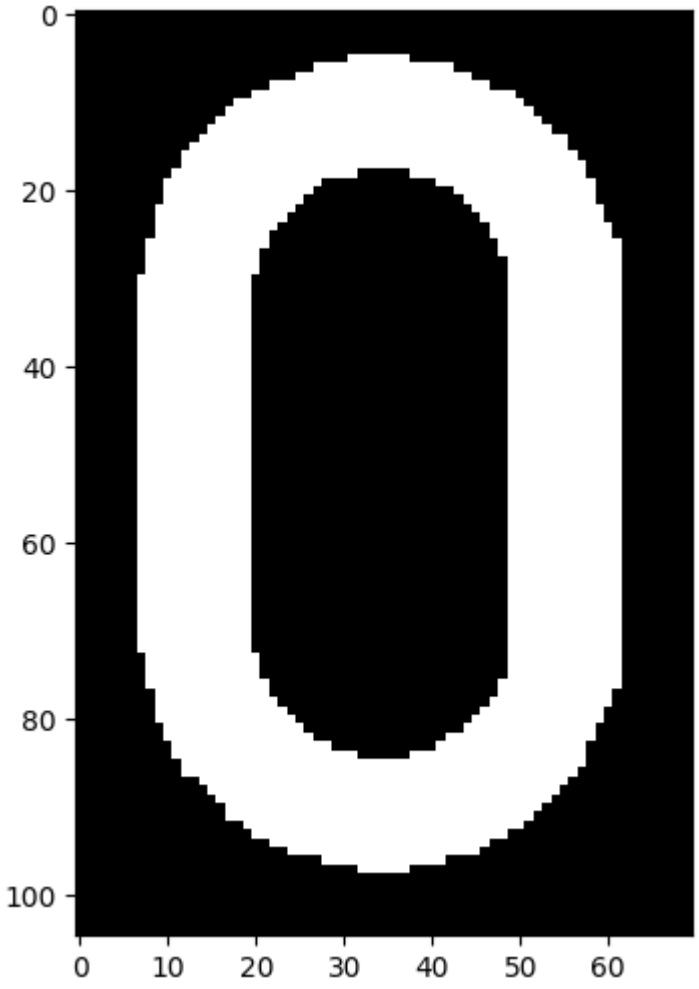
You task is to plot the feature vectors, computed by the functions `compactness()` and `extent()`, in a 2D-space called the **feature space!**. In such a space, the **x-axis represents the compactness** of a region and the **y-axis its extent**.

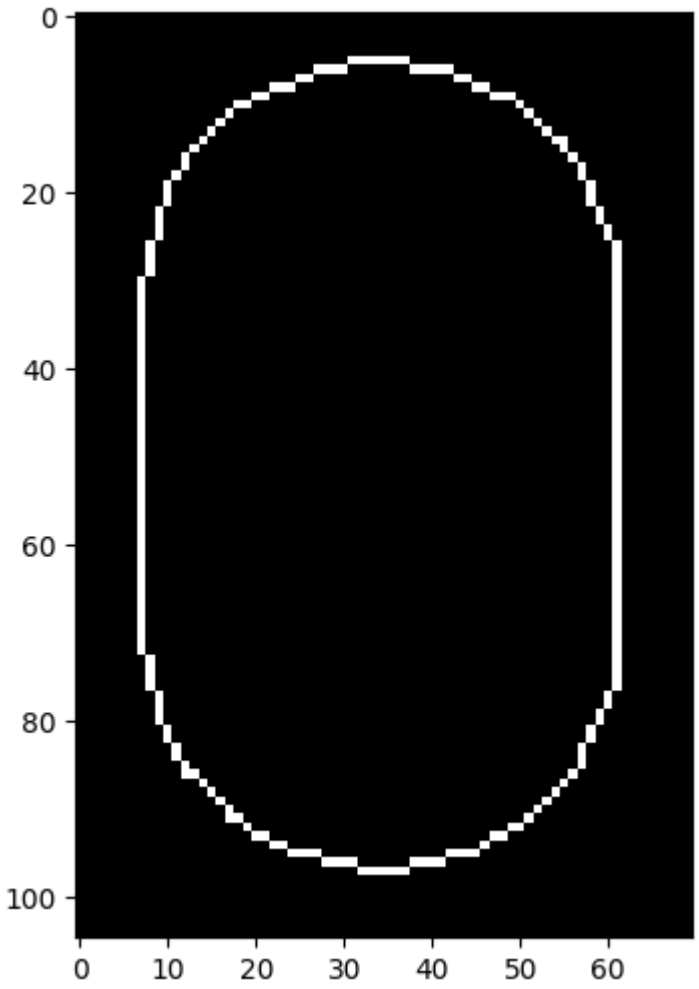
In this way, if the descriptions of the considered characters in this space don't appear close to each other, that means that they can be differentiated by relying on those

features. **The problem appears if two or more characters have similar features** (their respective points are near). This tell us that **those features are just not enough** for automatically detect the plate characters.

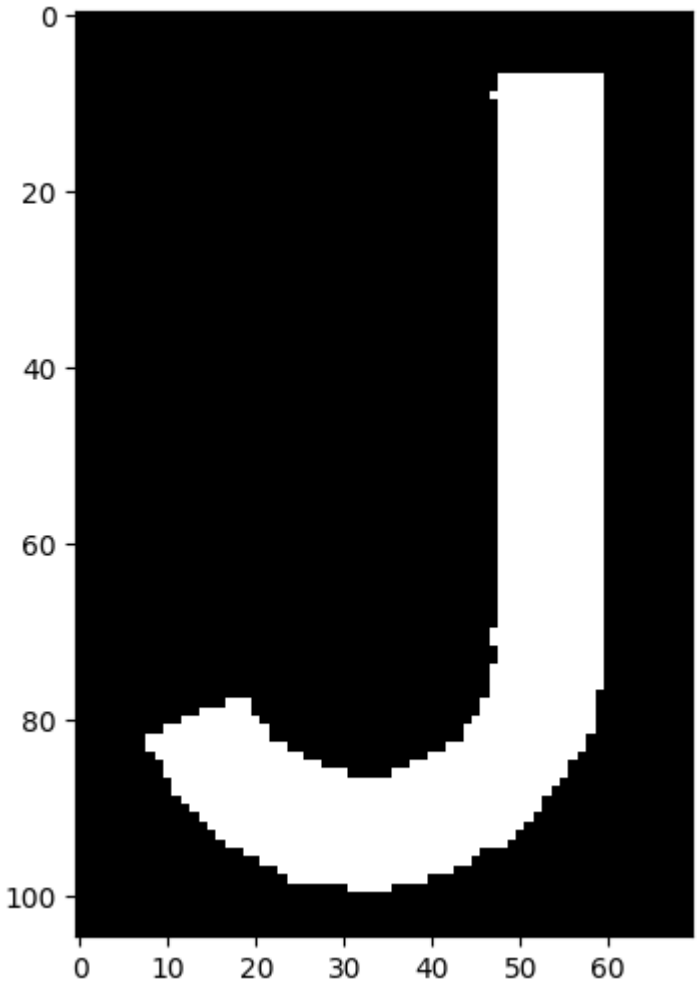
Tip: intro to pyplot.

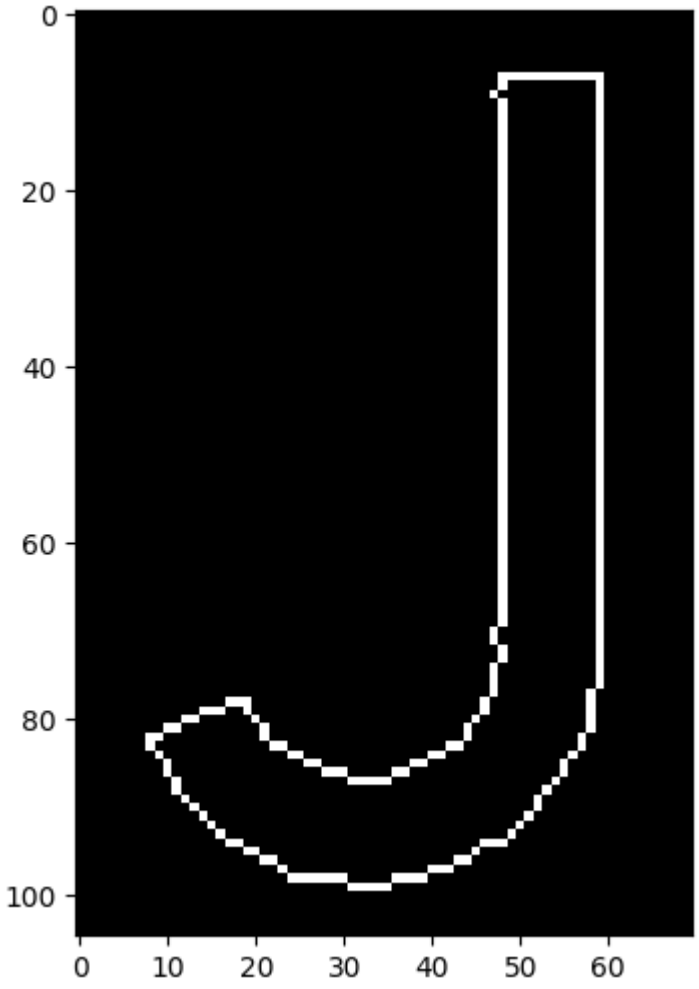
In [10]:
```
# Assignment 3
matplotlib.rcParams['figure.figsize'] = (6.0, 6.0)

# Read the images
zero = cv2.imread(images_path + 'region_0.png',0)
J = cv2.imread(images_path + 'region_J.png',0)
B = cv2.imread(images_path + 'region_B.png',0)
six = cv2.imread(images_path + 'region_6.png',0)

# Build the feature vectors
x_zero = np.array([compactness(zero), extent(zero)])
x_J = np.array([compactness(J), extent(J)])
x_B = np.array([compactness(B), extent(B)])
x_six = np.array([compactness(six), extent(six)])

# Define the scatter plot
fig, ax = plt.subplots()
plt.axis([0, 1/(4*np.pi), 0, 1])
plt.xlabel("Compactness")
plt.ylabel("Extent")

# Plot the points
plt.plot(x_zero[0], x_zero[1], 'go')
plt.text(x_zero[0]+0.005,  x_zero[1]+0.05, '0', bbox={'facecolor': 'green
plt.plot(x_J[0], x_J[1], 'ro')
plt.text(x_J[0]+0.005,  x_J[1]+0.05, 'J', bbox={'facecolor': 'red', 'alph
plt.plot(x_B[0], x_B[1], 'mo')
plt.text(x_B[0]+0.005,  x_B[1]+0.05, 'B', bbox={'facecolor': 'magenta', '
plt.plot(x_six[0], x_six[1], 'bo')
plt.text(x_six[0]+0.005,  x_six[1]+0.05, '6', bbox={'facecolor': 'blue',
```


0
20
40
60
80
100
0
10
20
30
40
50
60

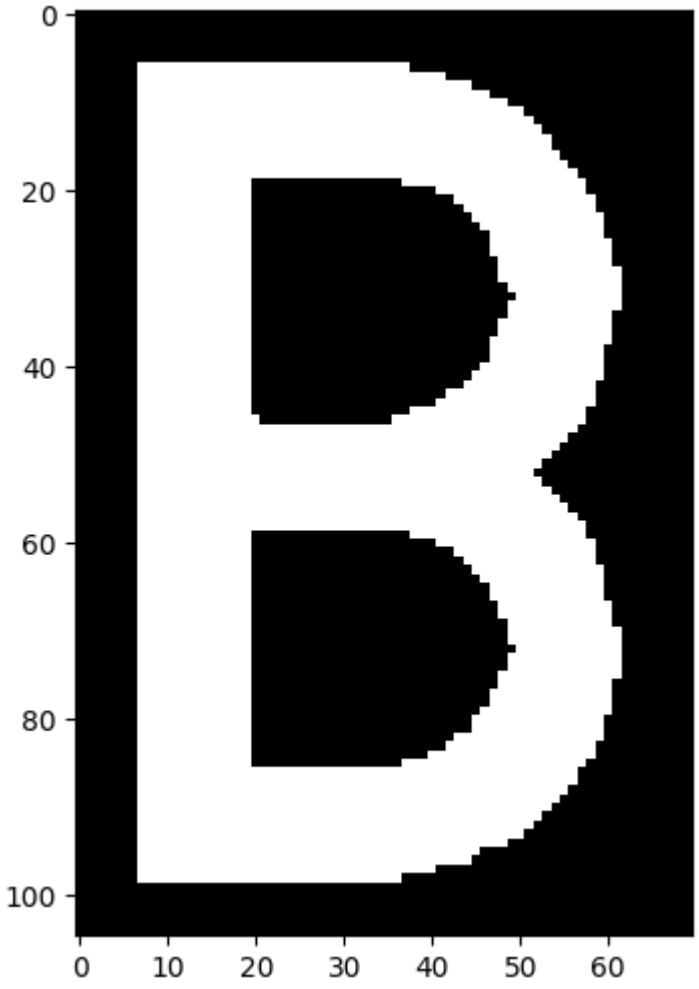

0
20
40
60
80
100
0
10
20
30
40
50
60

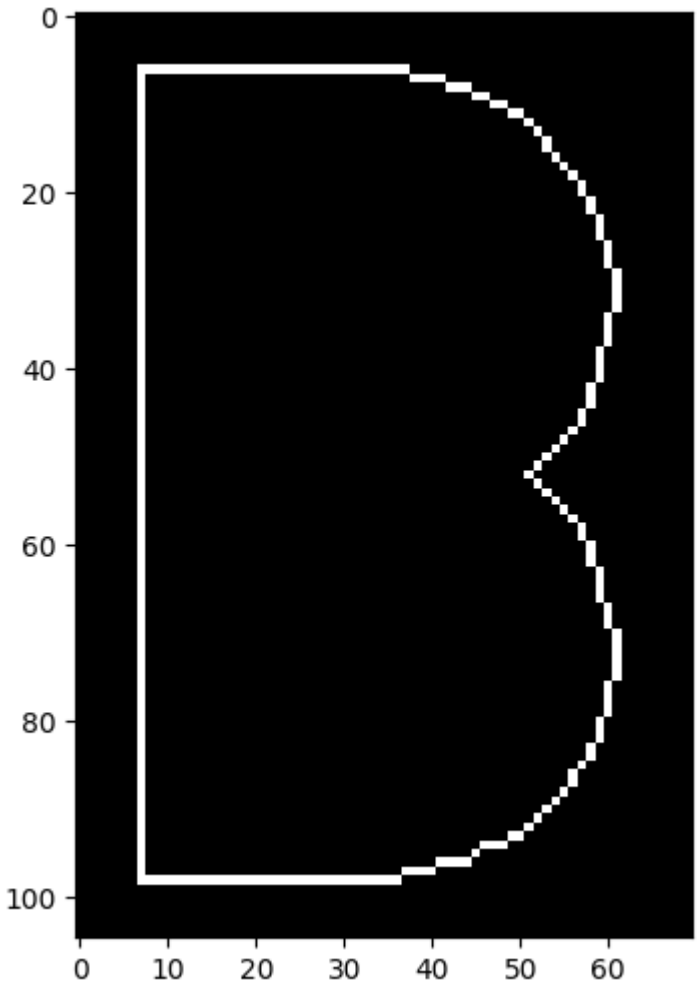
```
Area: 4307.0
Perimeter: 255.68123936653137
```



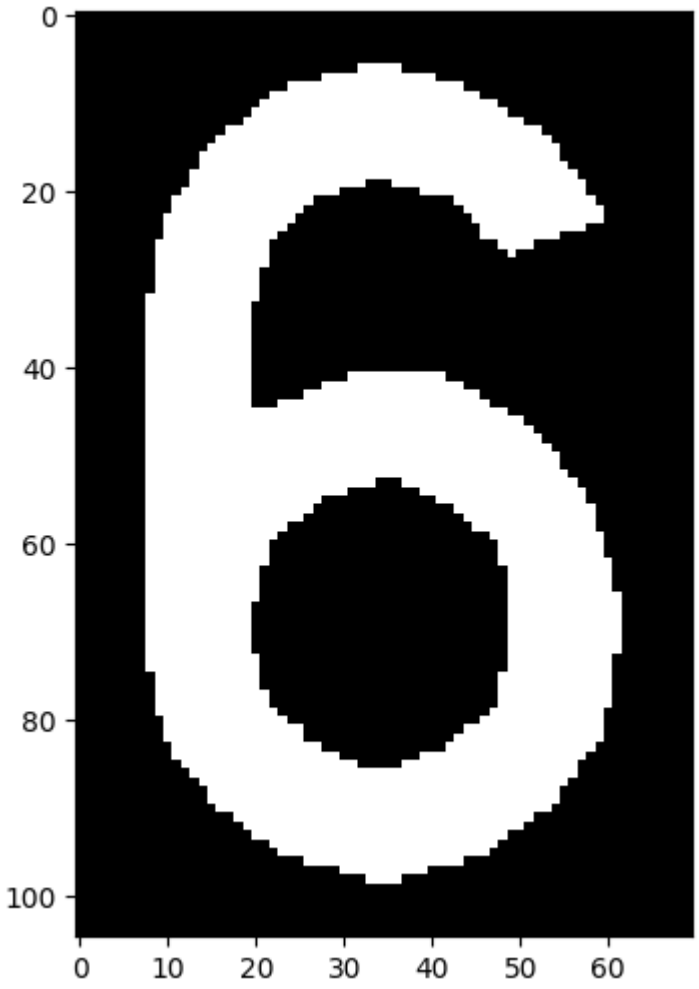



```
Area: 1386.0
Perimeter: 276.3675310611725
```


0
20
40
60
80
100
0
10
20
30
40
50
60

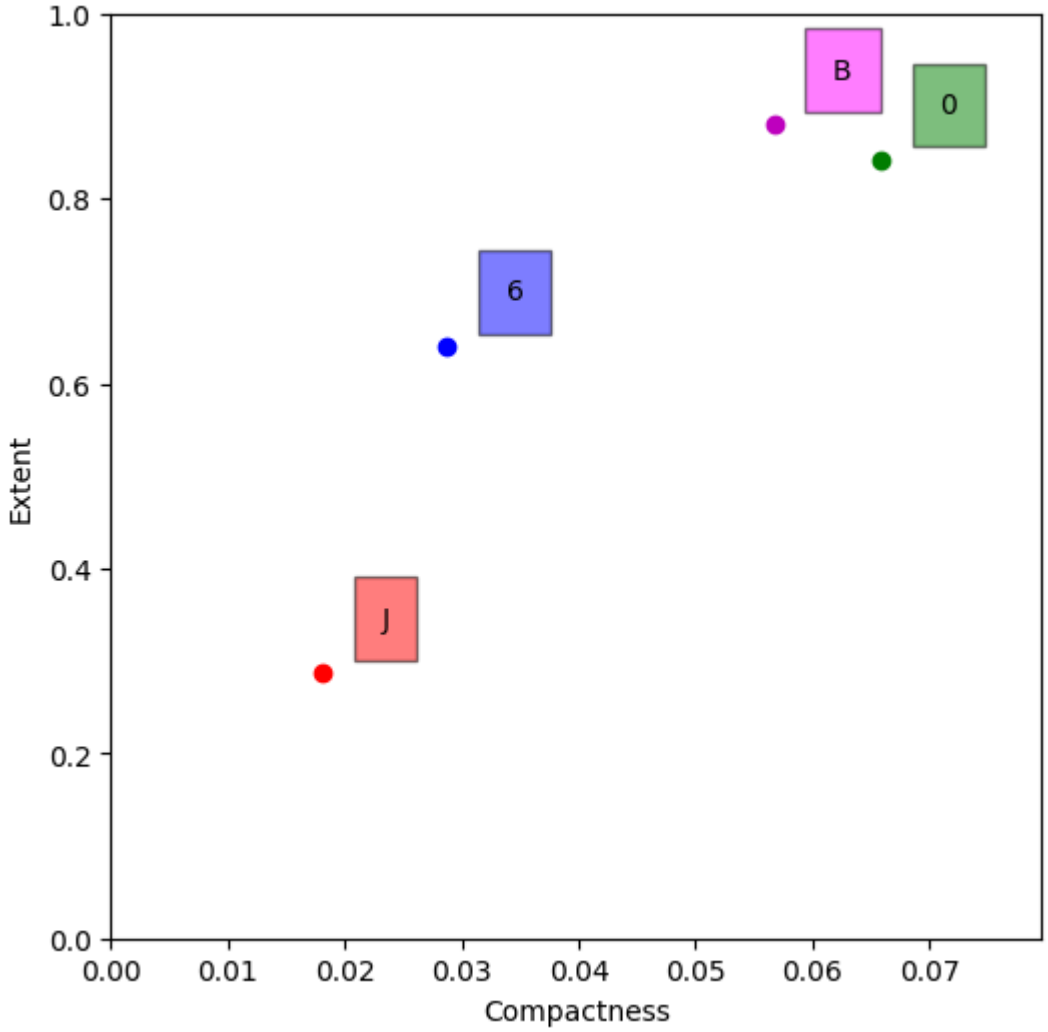

0
20
40
60
80
100
0
10
20
30
40
50
60

```
Area: 4498.0
Perimeter: 281.53910398483276
```




```
Area: 3217.5
Perimeter: 334.4924215078354
```

Out[10]: Text(0.033757159783557804, 0.6906810035842295, '6')



Thinking about it (3)

What do you think?

- Are they discriminative enough?

  No, compactness and extent alone may not be discriminative enough if characters like `0`, `B`, `6`, and `J` appear too close to each other in the feature space. This proximity would imply that these features do not fully capture the distinct characteristics of each character.

- If your answer is no, how could we handle this problem?

  o improve character differentiation, we could consider adding more features, such as aspect ratio, stroke thickness, or symmetry. Additionally, applying machine learning techniques like Principal Component Analysis (PCA) to find a more representative feature set, or using a classifier trained on a broader feature set, could help improve the classification accuracy for each character.

OPTIONAL

Surf the internet looking for **more shape features**, and try to find a pair of them working better than compactness and extent.

1. **Eccentricity**: Measures the elongation of a shape. Eccentricity is defined as the ratio of the distance between the foci of the shape's best-fitting ellipse to its major axis length. It ranges from 0 (perfect circle) to close to 1 (highly elongated shape). This feature is often effective for distinguishing between circular and elongated shapes.

2. **Solidity**: The ratio of the shape's area to its convex hull area. Solidity is close to 1 for compact shapes without concavities, such as circles or squares, and decreases as the shape has more concavities. This feature can be useful to distinguish between smooth, compact shapes and irregular or spiky shapes.

END OF OPTIONAL PART

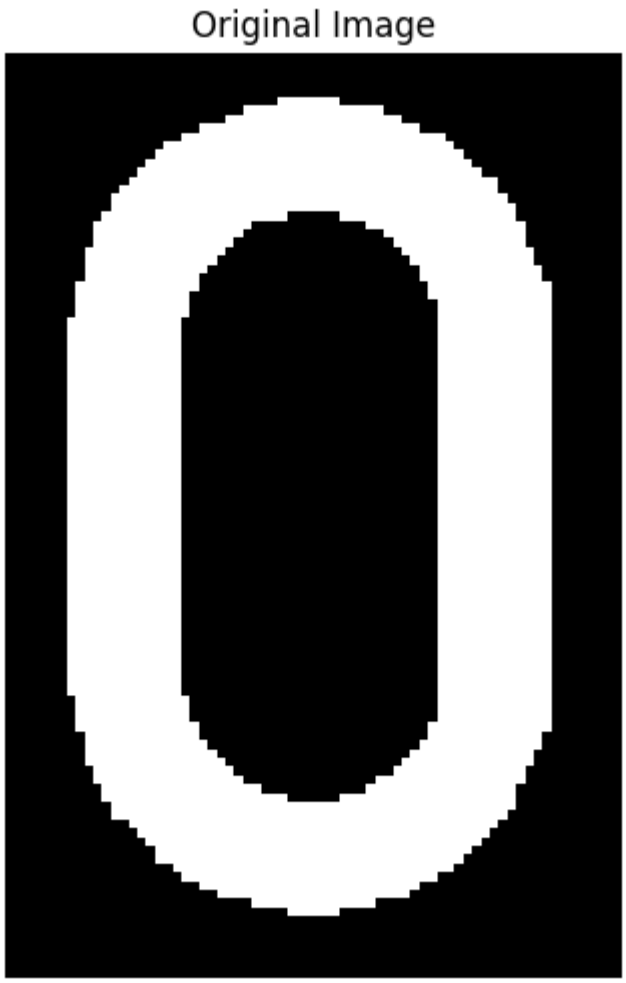
OPTIONAL

Take an image of a car plate, apply the thechniques already studied in the course to improve its quality, and binarize it. Then, extract some shape features and check where the numbers/letters are projected in the feature space.

In [17]:
```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Set plot size
plt.rcParams['figure.figsize'] = (6.0, 6.0)

# Path to images
zero = cv2.imread(images_path + 'region_0.png', 0)  # Load in grayscale

# Step 1: Preprocess the image (noise reduction + contrast enhancement)
preprocessed_img = cv2.GaussianBlur(zero, (5, 5), 0)
preprocessed_img = cv2.equalizeHist(preprocessed_img)

# Step 2: Binarization (Otsu's thresholding)
_, binary_img = cv2.threshold(preprocessed_img, 0, 255, cv2.THRESH_BINARY

# Step 3: Shape Feature Extraction
contours, _ = cv2.findContours(binary_img, cv2.RETR_EXTERNAL, cv2.CHAIN_A
features = []

# Definir los nombres de las características
feature_names = ['Aspect Ratio', 'Extent', 'Solidity'] + [f'Hu Moment {i+

for contour in contours:
    x, y, w, h = cv2.boundingRect(contour)
    aspect_ratio = float(w) / h if h != 0 else 0
    contour_area = cv2.contourArea(contour)
    bounding_box_area = w * h
    extent = contour_area / bounding_box_area if bounding_box_area != 0 e
    hull = cv2.convexHull(contour)
    hull_area = cv2.contourArea(hull)
    solidity = contour_area / hull_area if hull_area != 0 else 0
    moments = cv2.moments(contour)
    hu_moments = cv2.HuMoments(moments).flatten()
```

```
    features.append([aspect_ratio, extent, solidity] + list(hu_moments))

# Convert to NumPy array
features = np.array(features)

# Crear un diccionario de características
features_dict = {f'Contour {i+1}': dict(zip(feature_names, features[i]))

# Imprimir el diccionario
for contour, feats in features_dict.items():
    print(f"{contour}: {feats}")

# Visualizar solo la imagen original
plt.figure(figsize=(6, 6))
plt.title('Original Image')
plt.imshow(zero, cmap='gray')
plt.axis('off')
plt.show()
```

```
Contour 1: {'Aspect Ratio': np.float64(0.5730337078651685), 'Extent': np.f
loat64(0.8237497246089447), 'Solidity': np.float64(0.9882384035945553), 'H
u Moment 1': np.float64(0.1817360649461513), 'Hu Moment 2': np.float64(0.0
07446407505868962), 'Hu Moment 3': np.float64(3.5371844727678194e-07), 'Hu
Moment 4': np.float64(2.0126220361049787e-08), 'Hu Moment 5': np.float64(-
1.5614115234497698e-15), 'Hu Moment 6': np.float64(-1.5555122427996118e-0
9), 'Hu Moment 7': np.float64(6.675748839690858e-16)}
```



END OF OPTIONAL PART

Conclusion

Great work! You have learned about:

- what is the aim of region descriptors,
- the ideas behind two simple shape descriptors: compactness and extent, and
- to build a vector of features and analyze its discriminative power.

Unfortunately, it seems that those two features are not enough to differentiate the plate characters, so let's try more complex descriptors in the next notebook!

Extra work

Surf the internet looking for **more shape features**, and try to find a pair of them working better than compactness and extent.