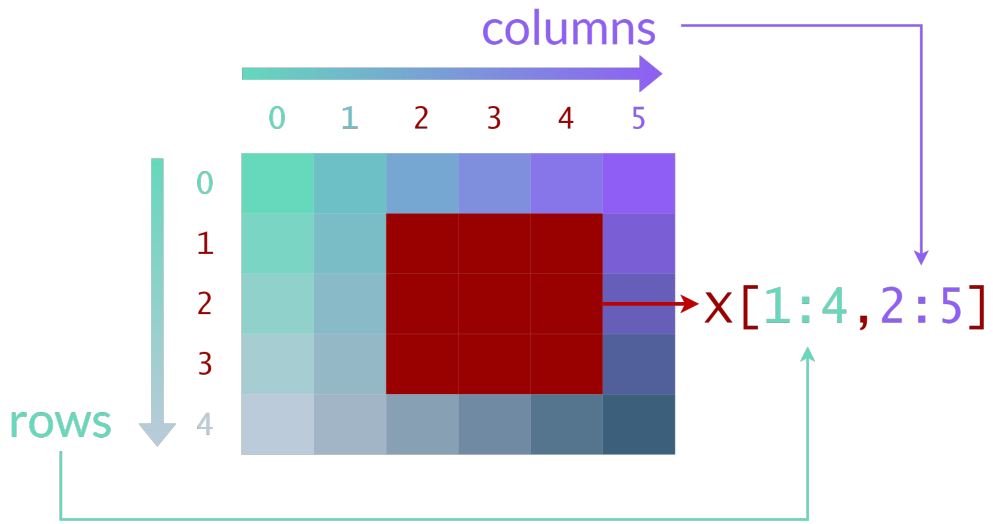
DataScientest • com

NumPy for Data Science

Manipulation of Numpy arrays

1. Conditional indexing of a Numpy array

In the previous notebook, we saw how to index an array by **slicing** along multiple dimensions:



A more advanced technique is to index the elements of an array using a **condition**:

```
# Creation of an array of shape 3x3
X = np.array ([[-1, 0, 30],
               [-2, 3, -5],
               [5, -5, 10]])

# We assign to all negative elements the value 0
X[X < 0] = 0

# Displaying the modified matrix
print(X)

>>> [[0, 0, 30],
>>>  [0, 3, 0 ],
>>>  [5, 0, 10]]
```

This allows you to easily access and modify elements that meet a specific condition.

In addition, it is possible to index an array using a condition **evaluated on another array**:

```
# Creation of 2 arrays with 8 elements
```

In [2]:

```
import numpy as np

items = np.array(["grid paper", "plate", "rubber band", "key chain", "bread", "speakers", "chocolate",
                  "fridge", "bowl", "shirt", "truck", "canvas", "monitor", "piano", "sailboat", "clamp",
                  "spring", "picture frame", "knife", "hanger", "pool stick", "buckel", "vase", "wagon",
                  "balloon", "thread", "couch", "drawer", "packing peanuts", "bottle", "needle",
                  "rusty nail", "blanket", "lamp", "box", "cookie jar", "washing machine", "paint brush",
                  "puddle", "sketch pad", "sandal", "doll", "floor", "sidewalk", "sand paper", "stockings",
                  "bag", "perfume", "magnet", "fake flowers", "street lights", "carrots", "purse", "thermostat",
                  "candle", "mouse pad", "remote", "clothes", "rubber duck", "hair brush", "computer", "toe ring",
                  "scotch tape", "nail file", "window", "table", "model car", "toothbrush", "shoes", "leg warmers",
                  "cat", "pillow", "rug", "hair tie", "phone", "tooth picks", "brocolli", "newspaper", "towel",
                  "watch", "lotion", "apple", "pants", "air freshener", "pen", "lace", "car", "headphones",
                  "charger", "toilet", "candy wrapper", "soy sauce packet", "sticky note", "shoe lace",
                  "zipper", "soda can", "bed", "cell phone", "lip gloss", "thermometer"])

quantities = np.array([310, 455, 295, 613, 812, 907, 564, 904, 829, 167, 517, 272, 416,
                        14, 251, 476, 757, 343, 472,  71, 160, 996, 182, 721, 565, 582,
                        279,  66, 297, 800, 914,  69, 498, 885, 114, 876, 635, 295, 146,
                        601, 941, 100, 370, 467, 423, 101, 504, 298, 757, 291, 163, 970,
                        921, 953, 458, 381, 692, 393, 749, 285, 454, 174,  37, 289, 863,
                        885, 331, 585, 678, 834, 349, 732, 149, 486, 993, 869, 967, 537,
                        220,  15, 457, 483, 387, 180, 579, 155, 134, 163, 314, 334, 429,
                        154,  18, 426, 363, 146, 454, 902, 145,  95])

discounts = np.array([25, 25, 50, 25, 50, 50, 50, 25, 50, 50, 25, 25, 25, 25, 50, 75, 25,
                      50, 50, 50, 25, 25, 25, 25, 75, 50, 25, 25, 25, 25, 90, 50, 25, 25,
                      25, 50, 50, 25, 25, 75, 75, 50, 25, 25, 50, 25, 90, 90, 50, 90, 25,
                      25, 25, 25, 25, 25, 25, 50, 25, 25, 75, 50, 50, 25, 50, 25, 25, 50,
                      25, 75, 25, 25, 50, 25, 25, 50, 75, 25, 25, 90, 25, 75, 25, 25, 25,
                      25, 25, 25, 50, 50, 75, 25, 50, 25, 25, 50, 25, 25, 25, 75])
```

- **(b)** Using conditional indexing on `items` and `quantities` arrays, display the name and quantity of objects that will have a 90% reduction.

In [3]:

```
# Insert your code here

print(items[discounts == 90])
print(quantities[discounts == 90])
```

```
['needle' 'bag' 'perfume' 'fake flowers' 'watch']
[914 504 298 291  15]
```

Hide solution

In [4]:
```
# Displaying the name of the objects with a 90% discount
print(items[discounts == 90])

# Displaying the quantity of objects with a 90% discount
print(quantities[discounts == 90])
```

```
['needle' 'bag' 'perfume' 'fake flowers' 'watch']
[914 504 298 291  15]
```

- **(c)** You want to buy a new cell phone (`"cell phone"`) and speakers (`"speakers"`). Using conditional indexing on `discounts` , find the discount that will be granted to these items.

In [5]:
```
# Insert your code here

print("Cell phone discount is %", discounts[items == "cell phone"][0])
print("Speakers discount is %", discounts[items == "speakers"][0])
```

```
Cell phone discount is % 25
Speakers discount is % 50
```

Hide solution

In [6]:
```
discount_cellphone = discounts[items == 'cell phone']
print("Cell phones will be discounted by", discount_cellphone[0], "percent.")


discount_speaker = discounts[items == 'speakers']
print("Speakers will be discounted by", discount_speaker[0], "percent.")
```

```
Cell phones will be discounted by 25 percent.
Speakers will be discounted by 50 percent.
```

- **(d)** The manager of the supermarket would like to know which items are likely to be sold very quickly. Display the name of the items whose **quantity is less or equal to 50** and the discount granted to them.
- **(e)** Which object is likely to be sold out **extremely** quickly?

In [13]:
```
# Insert your code here

print(items[quantities <= 50])
print(discounts[quantities <= 50], "\n")

for i in range(len(items)):
    for j in items[quantities <= 50]:
        if items[i] == j:
            print(j, ", Quantities =",quantities[i], ", Discount =",discounts[i])


print('\nIt seems Watch will be sold quickly. It has %90 discount and just 15 watch in the stock.')
```

```
['piano' 'scotch tape' 'watch' 'sticky note']
[25 50 90 50]

piano , Quantities = 14 , Discount = 25
scotch tape , Quantities = 37 , Discount = 50
watch , Quantities = 15 , Discount = 90
sticky note , Quantities = 18 , Discount = 50

It seems Watch will be sold quickly. It has %90 discount and just 15 watch in stock.
```

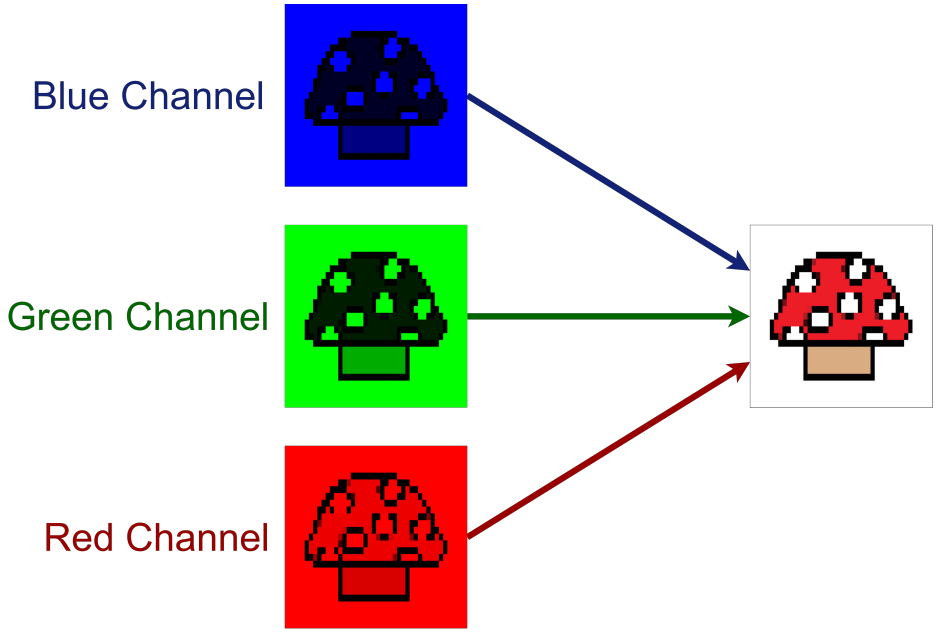
Hide solution

In [16]:
```
print("Objects", items[quantities <= 50])
print("Discount", discounts[quantities <= 50])

print("\n")
print("Watches ('watch') could be sold out very quickly because they are in small quantities and have a discount of 90%.")
```

```
Objects ['piano' 'scotch tape' 'watch' 'sticky note']
Discount [25 50 90 50]


Watches ('watch') could be sold out very quickly because they are in small quantities and have a discount of 90%.
```

2. Iterating over the elements of an array

A screen pixel can simulate a color by superimposing 3 channels corresponding to red, green and blue colors. By varying the light intensity of these channels, it is possible to cover a large part of the range of colors visible by the human eye.



For this reason, a **colored image** of size 32x32 pixels is often represented by an array of shape 32x32x3 where the 3rd dimension corresponds to the brightness intensity of each channel for each pixel of the image :

In [54]:
```
import cv2
import matplotlib.pyplot as plt

# Image import
img = cv2.imread("mushroom32_32.png")
img = np.int64(img)

# Image display
#_ = plt.imshow(img[:,:, ::-1])
_ = plt.imshow(img[:,:,::-1])
_ = plt.axis("off")
```

- **(b)** The image is stored in the `img` array. Display the **shape** of the created array.

In [16]:
```
# Insert your code here
img.shape
```

Out[16]: (32, 32, 3)

Hide solution

In [ ]:
```
# Displaying the shape of the array
print(img.shape)
```

To transform a color image into a **gray scale** image, you need to, **for each pixel of the image**, to compute the **brightness mean over the 3 channels**.

We will have to iterate on each pixel of the image, but for numpy arrays with several dimensions, the iterations must be done **dimension by dimension**:

```
# Creation of an array of dimensions 32x32x3 (rows x columns x channels)
X = np.zeros(shape = (32, 32, 3))

# Iteration over the first dimension of the array (the rows)
for row in X:
    # Iteration over the second dimension of the array (the pixels of the rows)
    for pixel in row:
        # Iteration over the third dimension of the array (the channels of the pixel)
        for channel in pixel:
            ...
            ...
```

A pixel of an image is an array with 3 elements corresponding to the brightness intensities of the channels. We can take the average of these three elements to obtain the brightness intensity of the pixel in gray scale.

- **(c)** Define a function named **`rgb_to_gray`** which will take an array `X` as argument. In this function, you will have to:

  - Instantiate an array filled with zeros named **`X_gray`** with the **same number of rows and columns as `X`** but **with a single channel**. This array will contain the light intensities of the pixels in gray scale.

In [112]:

```
img = cv2.imread("mushroom32_32.png")

X_gray = np.zeros(shape = (32, 32, 1))
for i, row in enumerate(img):
    print("i = ", i, "row = ",row)
    for j, pixel in enumerate(row):
        print("j = ", j , "pixel = ",pixel)
        average = 0
        for channel in pixel:
            average += channel
            print("average = ", average)
        average /= 3
        print("average / 3 = ", average)

        X_gray[i, j] = average
        print("X_gray[i,j] = " , i,j,X_gray[i, j])
```

```
i =  0 row =  [[255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]
 [255 255 255]]
```

In [ ]:

```
# Insert your code here

def rgb_to_gray(X):
    n_rows, n_columns, n_channels = X.shape

    # Create an array filled with zeros with the same number of rows and columns as X
    # but with only 1 channel
    X_gray = np.zeros(shape = (n_rows, n_columns, 1))


    for i, row in enumerate(X):
        print("row", i, row)
        for j, pixel in enumerate(row):
            print("pixel", j , average)
            average = 0
            for channel in pixel:
                average += channel
                print("average", average)
            average /= 3

            # We store the average of the intensities in X_gray
            X_gray[i, j] = average

    print(average, i, j, X.shape, X_gray[i,j])
    return  X_gray

# Test
  = rgb_to_gray(img)
```

In [71]:

```python
# Insert your code here

def rgb_to_gray(X):
    n_rows, n_columns, n_channels = X.shape

    # Create an array filled with zeros with the same number of rows and columns as X
    # but with only 1 channel
    X_gray = np.zeros(shape = (n_rows, n_columns, 1))

    # Iterating over the rows of the image
    for i, row in enumerate(X):
        # Iterating over the pixels of the row
        for j, pixel in enumerate(row):
            # Compute the average of the intensities of the channels
            average = 0
            for channel in pixel:
                average += channel
            average /= 3

            # We store the average of the intensities in X_gray
            X_gray[i, j] = average

    print(average, i, j, X.shape, X_gray[i,j])
    return  X_gray

# Test
  = rgb_to_gray(img)
```

```
255.0 31 31 (32, 32, 3) [255.]
```

Hide solution

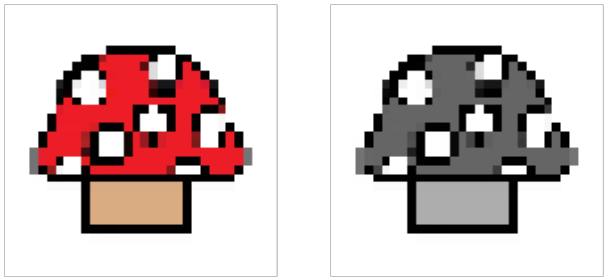
In [64]:

```
def rgb_to_gray(X):
    n_rows, n_columns, n_channels = X.shape

    # Create an array filled with zeros with the same number of rows and columns as X
    # but with only 1 channel
    X_gray = np.zeros(shape = (n_rows, n_columns, 1))

    # Iterating over the rows of the image
    for i, row in enumerate(X):
        # Iterating over the pixels of the row
        for j, pixel in enumerate(row):
            # Compute the average of the intensities of the channels
            average = 0
            for channel in pixel:
                average += channel
            average /= 3

            # We store the average of the intensities in X_gray
            X_gray[i, j] = average

    return X_gray

# Test
  = rgb_to_gray(img)
```

- **(d)** Run the following cell to display the result of `rgb_to_gray` :

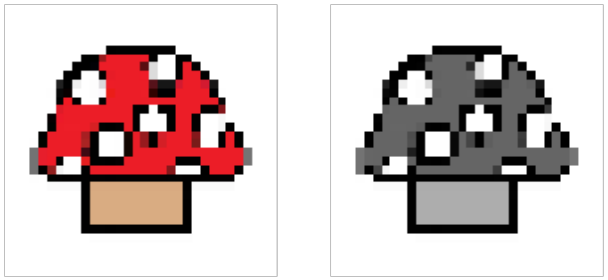
In [65]:

```
img = cv2.imread("mushroom32_32.png")

# Displaying the color image
plt.subplot(1, 2, 1)
_ = plt.imshow(img[:,:, :: - 1])
_ = plt.axis("off")

# Displaying the gray scale image
plt.subplot(1, 2, 2)
_ = plt.imshow(rgb_to_gray(img)[..., 0], cmap = 'gray')
_ = plt.axis("off")
```



In [66]:

```
img = cv2.imread("mushroom32_32.png")

# Displaying the color image
plt.subplot(1, 2, 1)
_ = plt.imshow(img[:,:, :: - 1])
_ = plt.axis("off")

# Displaying the gray scale image
plt.subplot(1, 2, 2)
_ = plt.imshow(rgb_to_gray(img)[..., 0], cmap = 'gray')
_ = plt.axis("off")
```



3. Reshaping an array

The **shape** of an array corresponds to its dimensions. Resizing an array is known as **reshaping**.

It happens that an array is not of the appropriate dimensions to be **visualized** or **processed with machine learning algorithms** which often can only process **vectors** and not matrices.

Thus, it is possible to use the **`reshape`** method of an array to reconstruct the data of the array with different dimensions. The argument of the `reshape` method is the shape we want to obtain:

```
# Create an array from a list of 10 elements
X = np.array([i for i in range(1, 11)]) # 1, 2, ..., 10

# Displaying the dimensions of X
print(X.shape)
>>> (10,)

# Displaying X
print(X)
>>> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Reshaping the array into a matrix with 2 rows and 5 columns
X_reshaped = X.reshape((2, 5))

# Displaying the new array
print(X_reshaped)
>>> [[1, 2, 3, 4, 5],
>>>  [6, 7, 8, 9, 10]]
```

It is possible to resize an array to any **shape** as long as **both shapes have the same number of elements**. In the previous example, the array `X` contains 10 elements and the shape we desire too (2 x 5 = 10).

In the following, we will briefly explore the **`digits`** dataset from the **`scikit-learn`** module, a Python library used for machine learning models.

The **`digits`** dataset contains **1797** images of handwritten digits ranging from 0 to 9. The objective of this dataset is to find a machine learning algorithm able to read handwritten numbers.

The images in this dataset have a resolution of **8x8 pixels in gray scale**.

In [72]:
```
# The load_digits function is used to load the "digits" dataset into an array
from sklearn.datasets import load_digits

# The load_digits function returns a dictionary containing
# the data as well as other information about the dataset
digits = load_digits()

# The data of the images are in the "data" key
X = digits['data']
```

> All images have been loaded into the `X` array.

- **(b)** Using the `shape` attribute, display the dimensions of `X`.

In [75]:
```
# Insert your code here

print(X)
X.shape
```

```
[[ 0.  0.  5. ...  0.  0.  0.]
 [ 0.  0.  0. ... 10.  0.  0.]
 [ 0.  0.  0. ... 16.  9.  0.]
 ...
 [ 0.  0.  1. ...  6.  0.  0.]
 [ 0.  0.  2. ... 12.  0.  0.]
 [ 0.  0. 10. ... 12.  1.  0.]]
```

Out[75]: (1797, 64)

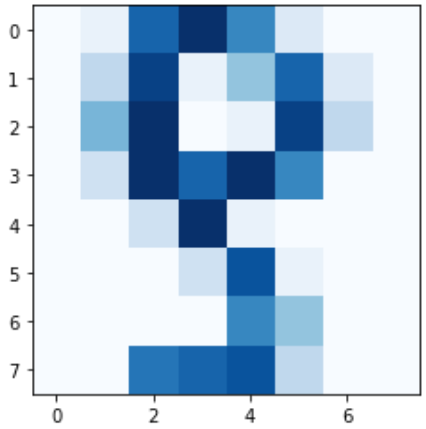
Hide solution

In [ ]:
```
print(X.shape)
```

> The X array containing the images has the dimension **1797x64**, which does not correspond to the dimensions of the images at all.
>
> In fact, the 1797 images of 8x8 dimensions have been **resized into vectors of size 8x8 = 64** to make them **compatible** with the algorithms of scikit-learn which **are not able to process matrix data**.
>
> Then, the 1797 vectors were **vertically stacked** into a **matrix** to form X. This makes it possible to have **the entire dataset** in a single matrix. Each row of X therefore corresponds to an image which has been transformed into a vector.
>
> If we want to visualize the images, we have to resize the vectors of 64 elements in matrices of dimensions 8x8. Indeed, the functions for displaying images

In [96]:
```
# Insert your code here
X_reshaped = X.reshape((1797, 8, 8))
img = X_reshaped[1100]
print(img)
  = plt.imshow(img, cmap="Blues")
```

```
[[ 0.  1. 12. 15. 10.  2.  0.  0.]
 [ 0.  4. 14.  1.  6. 12.  2.  0.]
 [ 0.  7. 15.  0.  1. 14.  4.  0.]
 [ 0.  3. 15. 12. 15. 10.  0.  0.]
 [ 0.  0.  3. 15.  1.  0.  0.  0.]
 [ 0.  0.  0.  3. 13.  1.  0.  0.]
 [ 0.  0.  0.  0. 10.  6.  0.  0.]
 [ 0.  0. 11. 12. 13.  4.  0.  0.]]
```



Hide solution

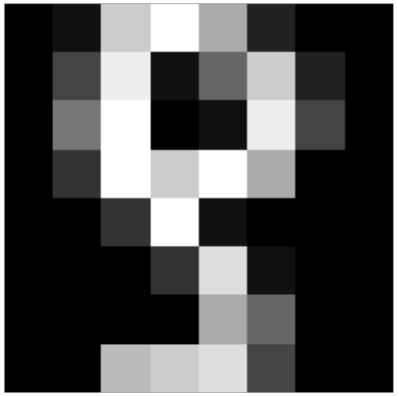
In [90]:
```
X_reshaped = X.reshape((1797, 8, 8))

img = X_reshaped[1100]
```

- **(e)** Run the next cell to display `img` . Can you recognize which number is displayed?

In [92]:

```
import matplotlib.pyplot as plt

_ = plt.imshow(img, cmap = 'gray')
_ = plt.axis("off")

print("Yes I can , it is 9.")
```

```
Yes I can , it is 9.
```



4. Concatenation of arrays

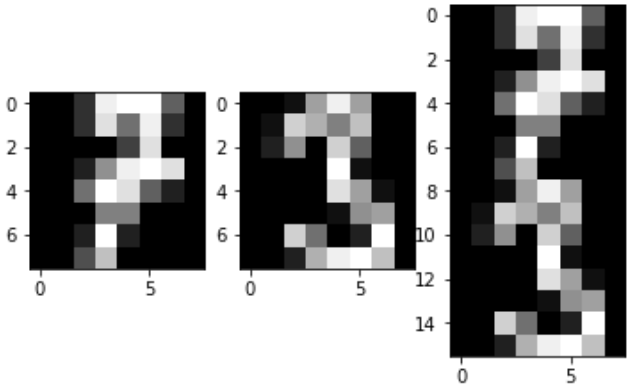
It is sometimes necessary to merge several arrays to build a dataset. For that, we can use the `np.concatenate` function:

```
# Creation of two arrays with 3 rows and 2 columns
# The first is filled with 1
X_1 = np.ones(shape = (3, 2))
print(X_1)

>>> [[1. 1. 1.]
>>>  [1. 1. 1.]]

# The second is filled with 0
X_2 = np.zeros(shape = (3, 2))
print(X_ 2)

>>> [[0. 0. 0.]
>>>  [0. 0. 0.]]

# Concatenation of the two arrays on the row axis
X_3 = np.concatenate([X_1, X_2], axis = 0)
print(X_3)

>>> [[1. 1. 1.]
>>>  [1. 1. 1.]
>>>  [0. 0. 0.]
>>>  [0. 0. 0.]]

# Concatenation of the two arrays on the column axis
X_4 = np.concatenate([X_1, X_2], axis = 1)
print(X_4)

>>> [[1. 1. 1. 0. 0. 0.]
>>>  [1. 1. 1. 0. 0. 0.]]
```

> - The arrays to concatenate must be passed as a **list** or a **tuple**.
> - The **`axis`** argument determines **the dimension on which** the arrays should be concatenated. The arrays must have the **same length** on this dimension.

In [105]:

```
# Insert your code here
img1 = X_reshaped[560]
img2 = X_reshaped[561]

img3 = np.concatenate([img1,img2], axis=0)

plt.subplot(1,3,1)
_ = plt.imshow(img1, cmap="gray")
plt.subplot(1,3,2)
_ = plt.imshow(img2, cmap="gray")
plt.subplot(1,3,3)
  = plt.imshow(img3, cmap="gray")
```



Hide solution

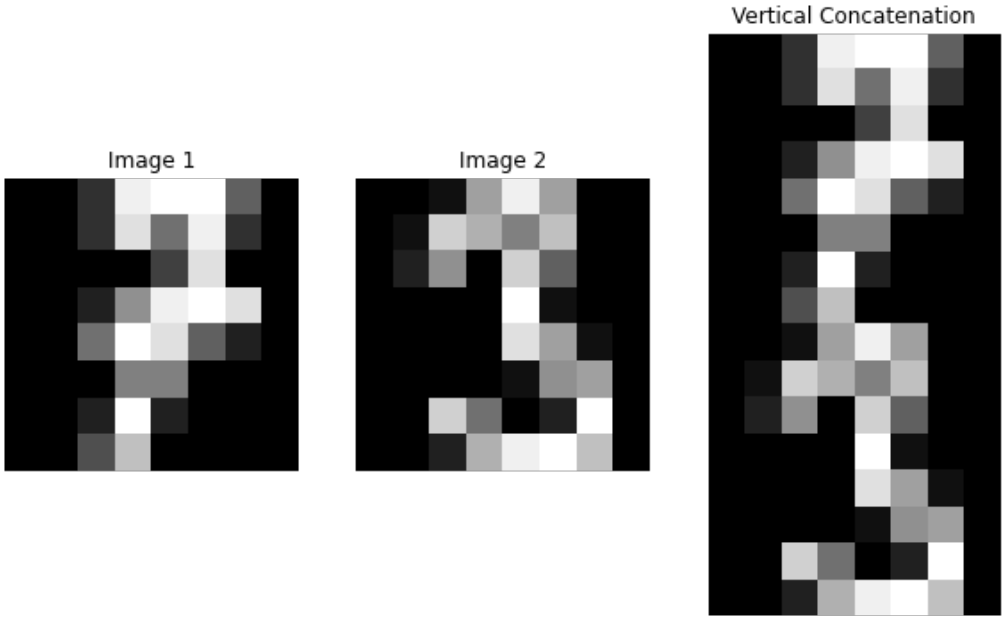
In [ ]:

```
# Image recovery
img1 = X_reshaped[560]
img2 = X_reshaped[561]

# Vertical concatenation of the images
img3 = np.concatenate([img1, img2], axis = 0)
```

- **(d)** Run the next cell to display the result of the concatenation.

In [106]:

```
## Displaying the first image
plt.subplot(1, 3, 1)
_ = plt.imshow(img1, cmap = 'gray')
_ = plt.axis("off")
_ = plt.title("Image 1")

# Displaying the second image
plt.subplot(1, 3, 2)
_ = plt.imshow(img2, cmap = 'gray')
_ = plt.axis("off")
_ = plt.title("Image 2")

# Displaying the concatenation of the images
plt.subplot(1, 3, 3)
_ = plt.imshow(img3, cmap = 'gray')
_ = plt.axis("off")
_ = plt.title("Vertical Concatenation")

# Resizing the figure
fig = plt.gcf()
fig.set_size_inches((10, 6))
```



✖ Unvalidate