04 - Array Manipulation

Array manipulation in NumPy refers to the various operations and techniques used to modify, reshape, combine, split, or otherwise transform arrays. It provides powerful functions and methods that allow you to manipulate the shape, size, dimensions, and content of arrays, enabling you to efficiently perform a wide range of data processing tasks.

04.1 - Transposing the array

Consider a scenario where you have collected data on the daily sales of different products over a period of time. You store this data in a 2D NumPy array, where each row represents a specific product, and each column represents the sales for a particular day.

Here's an example array representing the daily sales of three products (A, B, and C) over five days:

In [1]:
```
import numpy as np

sales_data = np.array([[10, 15, 12, 18, 20],
                       [8, 10, 14, 16, 12],
                       [5, 7, 9, 6, 8]])
```

Now, let's say you want to analyze the sales by days rather than by products. In other words, you want to transform the array so that each row represents a specific day, and each column represents the sales of different products on that day.

To achieve this, you can use the transpose function or the .T attribute in NumPy:

In [2]:
```
transposed_data = sales_data.T
print(transposed_data)
```

```
[[10  8  5]
 [15 10  7]
 [12 14  9]
 [18 16  6]
 [20 12  8]]
```

The `transpose` function or `.T` attribute rearranges the dimensions of the array, effectively swapping the rows with columns. In this case, it converts the original (3, 5) array into a transposed (5, 3) array. Now, each row represents a specific day, and each column represents the sales of different products on that day. This transformed array allows you to easily analyze and compare the sales across different days for each product.

The transpose operation is particularly useful when you need to perform matrix operations, work with multi-dimensional data, or reorganize data according to

desired dimensions. It provides a way to conveniently reshape and manipulate arrays to suit your specific analysis needs.

04.2 - Reshaping the array

Consider a scenario where you have collected data on the students' scores in a classroom. You have stored this data in a 1D NumPy array, where each element represents the score of an individual student.

Here's an example array representing the scores of 15 students:

In [3]:
```
scores = np.array([82, 75, 90, 88, 92, 78, 85, 80, 87, 95, 86, 91, 84, 89
```

1. **Reshaping it to 2D Array**

Now, let's say you want to analyze the scores in a 2D format, where each row represents a student, and each column represents a specific attribute related to the student (e.g., score, grade, attendance, etc.). Reshaping the array to 2D can provide a structured representation of the data.

To reshape the array from 1D to 2D, you can use the reshape function in NumPy:

In [4]:
```
reshaped_scores = scores.reshape((5, 3))
print(reshaped_scores)
```

```
[[82 75 90]
 [88 92 78]
 [85 80 87]
 [95 86 91]
 [84 89 93]]
```

The reshape function allows you to specify the desired shape of the array as an argument. In this case, we reshape the original (15,) array into a (5, 3) array, where it has 5 rows (corresponding to students) and 3 columns (representing different attributes).

Now, each row represents a student, and each column represents a specific attribute related to the student. This reshaped array provides a structured representation of the data, making it easier to analyze and perform operations on specific attributes for each student.

2. **Converting back to 1D using flatten():**

While the 2D representation is useful for analysis, there might be situations where you need to convert the array back to its original 1D format. The `flatten()` function in NumPy allows you to achieve this.

In [5]:
```
flattened_scores = reshaped_scores.flatten()
print(flattened_scores)
```

```
[82 75 90 88 92 78 85 80 87 95 86 91 84 89 93]
```

The flatten() function returns a 1D array by flattening the input array. In this case, it converts the (5, 3) reshaped_scores array back to its original 1D format.

04.3 - Append, Insert and Delete

Let's explore the concepts of appending, inserting, and deleting elements in NumPy arrays.

1. **Append**

Appending an element or array to an existing array in NumPy is a common operation. It allows you to add new data to the end of an array. This operation creates a new array with the combined elements.

Consider a scenario where you have a list of temperatures recorded each day for a week. You have stored this data in a NumPy array called temperatures.

In [6]:
```
temperatures = np.array([24.5, 23.8, 26.1, 25.6, 24.9])
```

Now, let's say you want to add the temperature recorded on the 6th day (27.3) to the existing array. You can use the append function in NumPy to achieve this:

In [7]:
```
updated_temperatures = np.append(temperatures, 27.3)
print(updated_temperatures)
```

```
[24.5 23.8 26.1 25.6 24.9 27.3]
```

The `append` function takes two arguments: the original array and the element(s) you want to append. In this case, we append the temperature value 27.3 to the temperatures array. Appending elements is useful when you need to expand the size of an array dynamically or add new data to an existing array without overwriting the original data.

2. **Insert**

Inserting an element or array at a specific position within an existing array is another common operation in NumPy. It allows you to add new data at a specified index while shifting the remaining elements accordingly.

Let's continue with the temperature example. Suppose you want to insert the temperature recorded on the 4th day (25.2) into the temperatures array at index 3 (0-based indexing). You can use the insert function in NumPy:

In [8]:
```
updated_temperatures = np.insert(temperatures, 3, 25.2)
print(updated_temperatures)
```

```
[24.5 23.8 26.1 25.2 25.6 24.9]
```

The `insert` function takes three arguments: the original array, the index at which you want to insert the element(s), and the element(s) you want to insert. In this case, we insert the temperature value 25.2 into the temperatures array at index 3. Inserting

elements allows you to place data at specific positions within an array, enabling you to maintain order and structure in your data.

3. **Delete**

Deleting an element or a set of elements from an existing array is a useful operation in NumPy. It allows you to remove unwanted data and resize the array accordingly.

Continuing with the temperature example, let's say you want to delete the temperature recorded on the 2nd day (23.8) from the temperatures array. You can use the delete function in NumPy:

In [9]:
```
updated_temperatures = np.delete(temperatures, 1)
print(updated_temperatures)
```

```
[24.5 26.1 25.6 24.9]
```

The `delete` function in NumPy takes two arguments: the original array and the index or indices of the element(s) you want to delete. In this case, we delete the temperature value at index 1 (representing the 2nd day) from the temperatures array. Deleting elements is beneficial when you want to remove specific data points from an array, adjust the size of the array, or clean up unwanted values.

04.4 - Combining arrays in Numpy

Combining arrays refers to the process of joining multiple arrays together to create a single array. This operation is useful when you want to merge data from different sources or concatenate arrays along different dimensions.

Consider a scenario where you have two arrays representing the heights and weights of individuals:

In [10]:
```
heights = np.array([165, 170, 155, 180])
weights = np.array([68, 72, 60, 85])
```

1. **Combining vertically using vstack**

The `vstack` function in NumPy is used to vertically stack arrays, meaning to concatenate them along the vertical axis (rows). This function allows you to combine arrays with the same number of columns.

Using the vstack function, you can merge the heights and weights arrays to create a new array representing the height and weight measurements of the individuals:

In [11]:
```
measurements = np.vstack((heights, weights))
print(measurements)
```

```
[[165 170 155 180]
 [ 68  72  60  85]]
```

Now, the measurements array has two rows, where the first row represents the heights and the second row represents the weights of the individuals. The `vstack`

function is useful when you want to combine arrays vertically, such as when merging data that corresponds to the same set of individuals or objects.

2. **Combining horizontally using hstack**

The `hstack` function in NumPy is used to horizontally stack arrays, meaning to concatenate them along the horizontal axis (columns). This function allows you to combine arrays with the same number of rows.

Continuing with the previous example, let's say you have another array representing the ages of the individuals:

In [12]:
```
ages = np.array([25, 32, 28, 36])
data = np.hstack((heights.reshape(-1, 1), weights.reshape(-1, 1), ages.re
print(data)
```

```
[[165  68  25]
 [170  72  32]
 [155  60  28]
 [180  85  36]]
```

Now, the data array has four rows, where the first column represents the heights, the second column represents the weights, and the third column represents the ages of the individuals.

The hstack function allows you to combine arrays horizontally, which is useful when you want to merge data that corresponds to different attributes or variables of the same individuals or objects.

04.5 - Splitting arrays in Numpy

Splitting arrays refers to dividing a single array into multiple smaller arrays. This operation is useful when you want to separate and extract specific parts of an array for further analysis or processing.

Consider a scenario where you have a 2D array representing the temperatures recorded at different locations for each month of the year:

In [13]:
```
temperatures = np.array([[25, 28, 30, 26],
                         [22, 24, 26, 20],
                         [28, 30, 32, 29],
                         [20, 23, 25, 22]])
```

1. **Splitting vertically using vsplit**

The `vsplit` function is used to vertically split an array into multiple smaller arrays along the rows.

In [14]:
```
split_rows = np.vsplit(temperatures, 2)
print(split_rows)
```

```
[array([[25, 28, 30, 26],
       [22, 24, 26, 20]]), array([[28, 30, 32, 29],
       [20, 23, 25, 22]])]
```

Now, `split_rows[0]` represents the temperatures for the first half of the year (first 2 rows) and `split_rows[1]` represents the temperatures for the second half of the year (last 2 rows).

2. **Splitting horizontally using hsplit**

The `hsplit` function is used to horizontally split an array into multiple smaller arrays along the columns.

In [15]:
```
split_columns = np.hsplit(temperatures, 2)
print(split_columns)
```

```
[array([[25, 28],
       [22, 24],
       [28, 30],
       [20, 23]]), array([[30, 26],
       [26, 20],
       [32, 29],
       [25, 22]])]
```

Now, `split_columns[0]` represents the temperatures for the first half of the locations (first 2 columns) and `split_columns[1]` represents the temperatures for the second half of the locations (last 2 columns).

In summary, the vsplit function splits the array vertically along the rows, while the hsplit function splits the array horizontally along the columns. Depending on your data and the specific splitting requirements, you can use these functions to divide an array into smaller, more manageable sub-arrays.
