02 - Array Mathematics

Consider this scenario in an Excel sheet. You are given 4 columns, `principal`, `rate_of_interest`, `num_of_years` and `simple_interest` as shown below.

principal	rate_of_interest	num_of_years	simple_interest
12000	6	3	
18000	6.25	4	
82000	7	2	
45000	6.75	5	

How would you calculate the `simple_interest` column in this case? Of course, you will multiply the three cells in the first row and then divide it by 100. You will then drag down the formula to the cells below to obtain the result for the remaining cells. Quite Easily Done!

But how will you do this in Python? Remember this thumb rule, we represent the columns in the spreadsheet as arrays in Python! So,

```
principal = [12000, 18000, 82000, 45000]
rate_of_interest = [6, 6.25, 7, 6.75]
num_of_years = [3, 4, 2, 5]
simple_interest = []

# pythonic way of finding simple interest
for i in range(0, len(principal)):
    si = principal[i]*rate_of_interest[i]*num_of_years[i]/100
    simple_interest.append(si)
print(simple_interest)
```

But this is quite tedious! It was so simple to do the same thing in Excel, but in Python - we need loops, lists and what not! Can we improve upon this?? The answer is yes! And for that, we will use numpy.

02.1 - Element-wise Arithmetic Operations

What will happen in Python if we add two lists as shown below?

```
a = [1, 2, 3, 4]
b = [5, 6, 7, 8]
print(a + b)
```

The output will be `[1, 2, 3, 4, 5, 6, 7, 8]`. Two lists got appended! Moreover, operations like `a - b, a*b and a/b` doesn't make sense!

What if I need output as `[6, 8, 10, 12]` which is the element-wise addition? In Python, we will need to write loop for that. However, numpy arrays allow us to do this element-wise operation easily.

Note that: To perform element-wise operations, you will need two arrays of same shape and size. Otherwise, you will get an error.

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

a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])
print(a + b)   # Addition of two arrays
print(b - a)   # Subtraction of two arrays
print(a*b)     # Multiplication of two arrays
print(b/a)     # Division of two arrays
print(a**b)    # One array raised to power of another
```

```
[ 6  8 10 12]
[4 4 4 4]
[ 5 12 21 32]
[5.         3.         2.33333333 2.        ]
[    1    64  2187 65536]
```

Once we have the ability to perform this element-wise operation, finding the `simple_interest` becomes easy and a one line instruction.

In [2]:
```
principal = np.array([12000, 18000, 82000, 45000])
rate_of_interest = np.array([6, 6.25, 7, 6.75])
num_of_years = np.array([3, 4, 2, 5])

simple_interest = principal*rate_of_interest*num_of_years/100
print(simple_interest)
```

```
[ 2160.   4500.  11480.  15187.5]
```

02.2 Element-wise Comparison

Imagine you have two groups of students: Group A and Group B. Each group has a list of test scores representing the performance of the students. You want to compare the test scores of the two groups to determine which group performed better overall.

Using NumPy arrays, you can represent the test scores of Group A and Group B as follows:

In [3]:
```
group_a_scores = np.array([85, 92, 78, 90, 88])
group_b_scores = np.array([80, 95, 85, 92, 87])
```

Now, let's perform some comparisons using NumPy array operations:

1. **Single Element Comparison:**

Single element comparison allows you to compare corresponding elements of two arrays and returns a new array with the result of each element-wise comparison.

In [4]:
```
# Comparing scores element-wise
comparison = group_a_scores > group_b_scores
print(comparison)
```

```
[ True False False False  True]
```

In this example, the resulting comparison array indicates True for elements where Group A has a higher score than Group B, and False for elements where Group A has a lower or equal score to Group B.

2. **Overall Comparison:**

You can use aggregate functions like `np.all()` and `np.any()` to determine if all or any elements of an array meet a specific condition.

In [5]:
```
# Check if all scores in Group A are greater than Group B
all_greater = np.all(group_a_scores > group_b_scores)
print(all_greater)

# Check if any score in Group A is greater than Group B
any_greater = np.any(group_a_scores > group_b_scores)
print(any_greater)
```

```
False
True
```

In this example, the all_greater variable returns False because not all scores in Group A are greater than Group B. However, the any_greater variable returns True because there is at least one score in Group A that is greater than Group B.

Comparing NumPy arrays allows you to analyze and make decisions based on the elements they contain. Just like comparing real-life objects, you can use NumPy array comparison to determine relationships, find patterns, filter data, or make logical decisions in data analysis or scientific computing scenarios.

02.3 Aggregate Functions

Aggregate functions in NumPy allow us to perform calculations on arrays, aggregating multiple values into a single result. You have studied these functions in Excel as well.

1. **np.sum():**

This function calculates the sum of all elements in an array. It can be used to find the total of a set of values or to calculate cumulative sums.

In [6]:
```
expenses = np.array([100, 150, 200, 75, 125])
total_expenses = np.sum(expenses)
print(total_expenses)
```

```
650
```

2. **np.cumsum():**

This function calculates the cumulative sum of elements in an array. It gives the running total of the values as we iterate through the array.

In [7]:
```
sales = np.array([50, 30, 40, 20, 25])
cumulative_sales = np.cumsum(sales)
print(cumulative_sales)
```

```
[ 50  80 120 140 165]
```

3. **np.min() and np.max():**

These functions return the minimum and maximum values from an array, respectively. They are useful for finding the smallest or largest value in a dataset.

In [8]:
```
temperatures = np.array([25, 30, 20, 35, 28])
min_temp = np.min(temperatures)
max_temp = np.max(temperatures)
print(min_temp, max_temp)
```

```
20 35
```

4. **np.mean() and np.median():**

np.mean() calculates the arithmetic mean of an array, giving the average value. np.median() returns the median value, which is the middle value when the array is sorted.

In [9]:
```
ages = np.array([25, 30, 35, 40, 45])
average_age = np.mean(ages)
median_age = np.median(ages)
print(average_age, median_age)
```

```
35.0 35.0
```

5. **np.std():**

This function calculates the standard deviation of an array, which measures the spread or dispersion of the values. It provides a measure of how much the values deviate from the mean.

In [11]:
```
scores = np.array([80, 85, 90, 75, 95])
std_dev = np.std(scores)
print(std_dev)
```

```
7.0710678118654755
```

6. **np.corrcoef():**

This function calculates the correlation coefficient between two arrays. It measures the strength and direction of the linear relationship between the two arrays.

In [12]:
```
x = np.array([1, 2, 3, 4, 5])
y = np.array([5, 4, 3, 2, 1])
correlation = np.corrcoef(x, y)
print(correlation)
```

```
[[ 1. -1.]
 [-1.  1.]]
```

These aggregate functions provide useful insights and statistical measures for analyzing data. Whether it's finding sums, calculating averages, determining the spread of values, or examining relationships between variables, these functions enable efficient computations on NumPy arrays in real-life scenarios.
