NumPy Arithmetic and Statistics - Computations and Aggregations

WE'LL COVER THE FOLLOWING

1. Computations on NumPy Arrays

The reason for NumPy's importance in the *Pythonic* data science world is its ability to perform computations in a fast and efficient manner. Now that we are familiar with the basic nuts and bolts of NumPy, we are going to dive into learning it to perform computations.

NumPy provides the so-called *universal functions* (*ufuncs*) that can be used to make repeated calculations on array elements in a very efficient manner. These are functions that operate on *nD*-arrays in an element-by-element fashion. Remember the vectorized operations from earlier.

Mathematical Functions

What are some of the most common and useful mathematical *ufuncs* available in the NumPy package? Let's explore them with some concrete examples.

The arithmetic operators, as shown in the code widget below, are conveniently wrapped around specific functions built into NumPy; for example, the + operator is a wrapper for the *add* ufunc.

Run the code in the widget below, **tweak** the inputs, and **observe** the outputs of the print statements.

```
import numpy as np

x = np.arange(10)

# Native arithmentic operators
print("x =", x)
print("x + 5 =", x + 5)
print("x - 5 =", x - 5)
print("x * 5 =", x * 5)
print("x / 5 =", x / 5)
print("x ** 2 = ", x ** 2)
print("x % 2  = ", x % 2)

# OR we can use explicit functions, ufuncs, e.g. "add" instead of "+"
print(np.add(x, 5))
print(np.subtract(x, 5))
print(np.multiply(x, 5))
print(np.divide(x, 5))
print(np.power(x, 2))
print(np.mod(x, 2))
```

Some of the most useful functions for data scientists are the **trigonometric functions**. Let's look into these. Let's define an array of angles first and then compute some trigonometric functions based on those values:

```
theta = np.linspace(0, np.pi, 4)
print("theta      = ", theta)
print("sin(theta) = ", np.sin(theta))
print("cos(theta) = ", np.cos(theta))
print("tan(theta) = ", np.tan(theta))
```

Similarly, we can also obtain logarithms and exponentials.

> **Note**: *These might not seem useful to you at the moment, but you will see their direct application in our final project.*

```
x = [1, 2, 3]
print("x     =", x)
print("e^x   =", np.exp(x))
print("2^x   =", np.exp2(x))
print("3^x   =", np.power(3, x))

print("ln(x)    =", np.log(x))

print("log2(x)  =", np.log2(x))
print("log10(x) =", np.log10(x))
```

Universal Function Methods

ufuncs provide some methods that take two input parameters and return one output parameter. `reduce` and `accumulate` are two of the most important ones, so let's look into those.

a. Calling the reduce method

Say we want to apply some operation to reduce an **array to a single value**. We can use the `reduce()` method for this. This method repeatedly applies the given operation to the elements of an array until only a single result remains. For example, calling *reduce* on the add functions returns the sum of all elements in the array:

```
x = np.arange(1, 6)
sum_all = np.add.reduce(x)

print(x)
print(sum_all)
```

> **Note:** *add.reduce()* is equivalent to calling *sum()*. In fact, when the argument is a NumPy array, *np.sum* ultimately calls *add.reduce* to do the job. This overhead of handling its argument and dispatching to *add.reduce* can make *np.sum* slower. For more details, you can refer to this answer on StackOverflow.

b. Calling the accumulate method

If we need to store all the **intermediate results** of the computation, we can use `accumulate()` instead:

```
x = np.arange(1, 6)
sum_acc = np.add.accumulate(x)

print(x)
print(sum_acc)
```

2. Aggregations

When we have large amounts of data, as a first step, we like to get an understanding of the data first by computing its summary statistics, like mean and standard deviation.

> **Note**: *We will look into the theoretical aspects of these statistical concepts in the "Statistics for Data Science" section, so don't worry if you don't remember what standard deviation is, for instance!*

NumPy provides some very handy built-in aggregate functions that allow us to summarize our data, e.g., `np.mean(x)` gives us the mean value of the array. Understanding with code in a hands-on way is always better, so let's explore these aggregates with code.

```
import numpy as np

x = np.random.random(100)

# Sum of all the values
print("Sum of values is:", np.sum(x))
# Mean value
print("Mean value is: ", np.mean(x))

#For min, max, sum, and several other NumPy aggregates,
#a shorter syntax is to use methods of the array object itself,
# i.e. instead of np.sum(x), we can use x.sum()
print("Sum of values is:", x.sum())
print("Mean value is: ", x.mean())
print("Max value is: ", x.max())
print("Min value is: ", x.min())
```

Similarly, we can perform aggregate operations on multi-dimensional arrays as well. Also, if we want to **compute the minimum row wise or column wise**, we can use the `np.amin` version instead. Let’s see how:

```
import numpy as np

grid = np.random.random((3, 4))
print(grid)

print("Overall sum:", grid.sum())
print("Overall Min:", grid.min())

# Row wise and column wise min
print("Column wise minimum: ", np.amin(grid, axis=0))
print("Row wise minimum: ", np.amin(grid, axis=1))
```

Now we know how to perform mathematical operations and aggregations. In the next lesson, we will learn about some subtle data operations like using boolean masks and performing comparisons.