A Beginner's Guide to NumPy- Part 2

Numpy is a fundamental library to master in Python. In Part 1, we discussed various methods for creating arrays, including through various random functions. We covered basic arithmetic operations and broadcasting principles, along with examples useful for numpy operations.

In this Part 2, we will understand concepts such as statistical operations, scientific operations, trigonometric operations, array manipulation, and slicing.

Let's dive in.

Statistical Operations:

Statistical operations are commonly used in data analysis to summarize and understand the distribution of data. In the below code we will calculate Mean, Median, Standard Deviation, Variance, 75th percentile as an example.

In [19]:
```
import numpy as np

# Create a NumPy array
data = np.array([1, 2, 3, 4, 5, 20]) #purposely giving non-symmetric data

# Calculate mean
mean_value = np.mean(data)
print("Mean:", mean_value.round(2))

median_value = np.median(data)
print("Median:", median_value)

# Calculate standard deviation
std_deviation = np.std(data)
print("Standard Deviation:", std_deviation.round(2))

# Calculate variance
variance = np.var(data)
print("Variance:", variance.round(2))

# Calculate percentile (e.g., 75th percentile)
percentile_75 = np.percentile(data, 75)
print("75th Percentile:", percentile_75.round(2))

#Note: I have used round function to round the output to two decimal places
```

```
Mean: 5.83
Median: 3.5
Standard Deviation: 6.47
Variance: 41.81
75th Percentile: 4.75
```

Scientific Operations:

Scientific operations involve mathematical functions commonly used in scientific computing and data analysis. In the below examples we will calculate exponential values, natural logarithmn

values.

In [22]:
```
x = np.array([1, 2, 3, 4, 5])
# Compute exponential of each element
exp_values = np.exp(x)
print("Exponential:", exp_values)

# Compute natural logarithm of each element
log_values = np.log(x)
print("Natural Logarithm:", log_values)
```

```
Exponential: [  2.71828183   7.3890561   20.08553692  54.59815003 148.4131591 ]
Natural Logarithm: [0.         0.69314718 1.09861229 1.38629436 1.60943791]
```

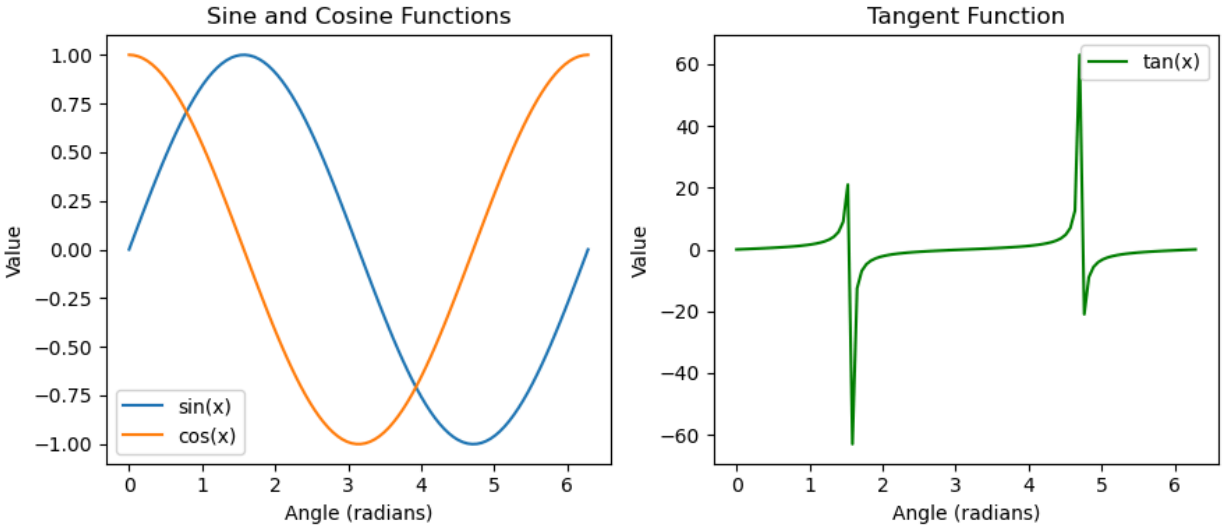
Trigonometric Operations:

Trigonometric operations are often used to analyze periodic data and model cyclic phenomena. Trigonometric functions (sin, cos, tan) are useful for handling cyclic data, such as time series or periodic signals.

In [44]:
```
import numpy as np
import matplotlib.pyplot as plt

# Generate an array of angles from 0 to 2*pi
angles = np.linspace(0, 2*np.pi, 100)

# Compute sine, cosine, and tangent of the angles
sin_values = np.sin(angles)
cos_values = np.cos(angles)
tan_values = np.tan(angles)

# Print these values will be long unreadable list- Lets visualize.
```

Cannot visualize tangent curve with Cosine and sine curve in the same axis. Hence creating them as subplots.

In [43]:
```
plt.figure(figsize=(9,4))

# Plot sine and cosine curves in the first subplot
plt.subplot(1, 2, 1)
plt.plot(angles, sin_values, label='sin(x)')
plt.plot(angles, cos_values, label='cos(x)')
plt.xlabel('Angle (radians)')
plt.ylabel('Value')
plt.title('Sine and Cosine Functions')
plt.legend()

# Plot tangent function in the second subplot
plt.subplot(1, 2, 2)
plt.plot(angles, tan_values, label='tan(x)', color='green')
plt.xlabel('Angle (radians)')
plt.ylabel('Value')
plt.title('Tangent Function')
plt.legend()

plt.tight_layout()  # Adjust subplot layout to prevent overlap
plt.show()
```



Array Manipulation in NumPy

Array manipulation refers to the process of changing the shape, size, or content of arrays. NumPy provides a variety of functions for performing array manipulation tasks efficiently.

Reshaping Arrays:

Reshaping arrays involves changing the shape of an array without changing its data. This operation is useful for converting arrays between different dimensions or rearranging elements.

In [51]:
```
import numpy as np

# Create a 1D array
arr = np.arange(6)

# Reshape the array into a 2x3 matrix
reshaped_arr = arr.reshape(2, 3)

print("Original Array:",arr.shape)
print(arr)
print("\nReshaped Array:",reshaped_arr.shape)
print(reshaped_arr)
```

```
Original Array: (6,)
[0 1 2 3 4 5]

Reshaped Array: (2, 3)
[[0 1 2]
 [3 4 5]]
```

Concatenation and Splitting:

Concatenation combines multiple arrays into a single array, while splitting divides a single array into multiple smaller arrays.

In [59]:
```
# Concatenate two arrays along rows
arr1 = np.array([[1, 2, 3], [4, 5, 6]])
arr2 = np.array([[7, 8, 9], [10, 11, 12]])
```

```
concatenated_arr = np.concatenate((arr1, arr2), axis=0)

# Split an array into multiple arrays along columns
split_arrays = np.array_split(concatenated_arr, 2, axis=1)

print ('Input array:\n',arr1,'\n\n',arr2)

print("\nConcatenated Array:")
print(concatenated_arr)
print("\nSplit Arrays:")
for arr in split_arrays:
    print(arr)
```

```
Input array:
 [[1 2 3]
 [4 5 6]]

 [[ 7  8  9]
 [10 11 12]]

Concatenated Array:
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]

Split Arrays:
[[ 1  2]
 [ 4  5]
 [ 7  8]
 [10 11]]
[[ 3]
 [ 6]
 [ 9]
 [12]]
```

Flattening Arrays:

Flattening arrays converts multi-dimensional arrays into a 1D array by collapsing all dimensions.

In [61]:
```
# Create a 2D array
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])

# Flatten the array
flattened_arr = arr_2d.flatten()

print("Original 2D Array:")
print(arr_2d)
print("\nFlattened Array:")
print(flattened_arr)
```

```
Original 2D Array:
[[1 2 3]
 [4 5 6]]

Flattened Array:
[1 2 3 4 5 6]
```

Adding and Removing Elements:

Adding and removing elements involve inserting, appending, or deleting elements from arrays.

In [65]:
```
# Append elements to an array
arr = np.array([1, 2, 3])
appended_arr = np.append(arr, [4, 5])

# Delete elements from an array
arr1 = np.array([1, 2, 3, 4, 5])
deleted_arr = np.delete(arr1, [1, 3])

print('Input array:\n', arr)
print("Appended Array:")
print(appended_arr)
print('\nInput array:\n', arr1)
print("\nDeleted Array:")
print(deleted_arr)
```

```
Input array:
 [1 2 3]
Appended Array:
[1 2 3 4 5]

Input array:
 [1 2 3 4 5]

Deleted Array:
[1 3 5]
```

Indexing and Slicing :

Indexing and slicing are essential techniques for accessing and manipulating elements in NumPy arrays. They allow you to extract subsets of data from arrays based on specific criteria, such as indices or conditions.

Indexing: In NumPy, indexing works similarly to standard Python lists, where you use square brackets [] to specify the indices of the elements you want to access.

In [66]:
```
import numpy as np

# Create a 1D array
arr = np.array([1, 2, 3, 4, 5])

# Access individual elements using indices
print("First element:", arr[0])
print("Last element:", arr[-1])
```

```
First element: 1
Last element: 5
```

In [67]:
```
import numpy as np

# Create a 1D array
arr = np.array([1, 2, 3, 4, 5])

# Access individual elements using indices
print("Second element:", arr[1])
```

```
# Slice the array to extract a subset
print("Elements from index 1 to 3:", arr[1:4])
```

```
Second element: 2
Elements from index 1 to 3: [2 3 4]
```

Slicing:

Slicing allows you to extract a subset of elements from an array by specifying a range of indices. You use the colon : operator to indicate the start, stop, and step size of the slice.

In [71]:
```
# Create a 1D array
arr = np.array([1, 2, 3, 4, 5])

# Slice the array to extract a subset
print("First three elements:", arr[:3])
print("Last two elements:", arr[-2:])
print("Every other element:", arr[::2])
print("Elements from index 1 to 3:", arr[1:4])
```

```
First three elements: [1 2 3]
Last two elements: [4 5]
Every other element: [1 3 5]
Elements from index 1 to 3: [2 3 4]
```

In [76]:
```
# Create a 2D array
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Slice the array along rows and columns
print ("Input\n",arr_2d)
print("\nFirst row:\n", arr_2d[0, :])
print("\nFirst two rows:\n", arr_2d[:2, :])
print("\nLast column:\n", arr_2d[:, -1])
print("\nSubmatrix:\n", arr_2d[:2, :2])
print("\nElement at row 1, column 2:\n", arr_2d[1, 2])
```

```
Input
 [[1 2 3]
 [4 5 6]
 [7 8 9]]

First row:
 [1 2 3]

First two rows:
 [[1 2 3]
 [4 5 6]]

Last column:
 [3 6 9]

Submatrix:
 [[1 2]
 [4 5]]

Element at row 1, column 2:
 6
```

Slicing with Step Size:

Let's include the Step-size parameter. It determines the stride between each element selected in the slice.

In [83]:
```
array_1d= np.arange(1,9)
print("Input array:\n", array_1d)
print("\nEvery second element:\n", array_1d[::2])
```

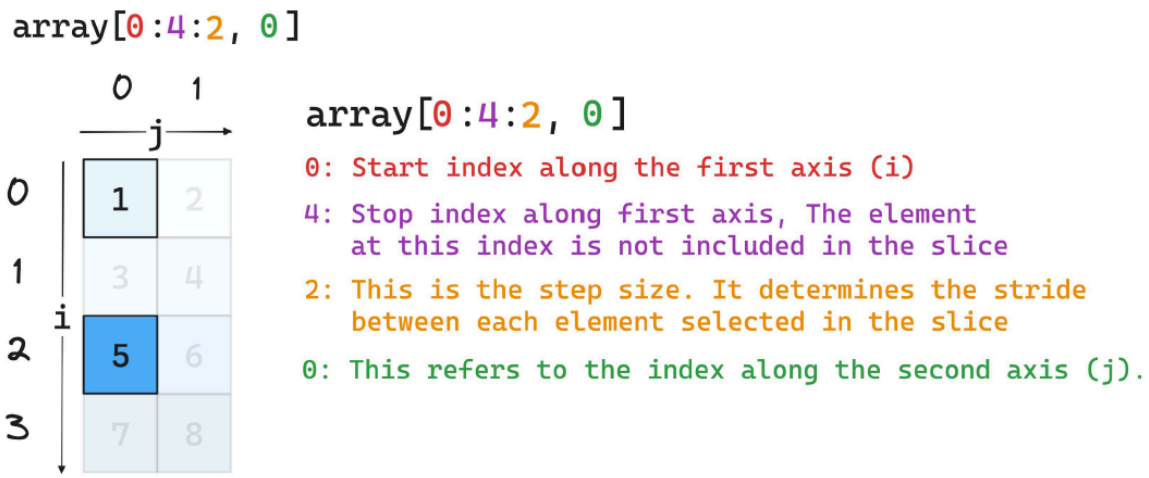
```
Input array:
 [1 2 3 4 5 6 7 8]

Every second element:
 [1 3 5 7]
```

In [87]:
```
#Lets convert as a (4,2)- 2D array
array_2d=array_1d.reshape(4,2)
print(array_2d)
```

```
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
```

In [101…
```
print(array_2d[0:4:2,0])
```

```
[1 5]
```

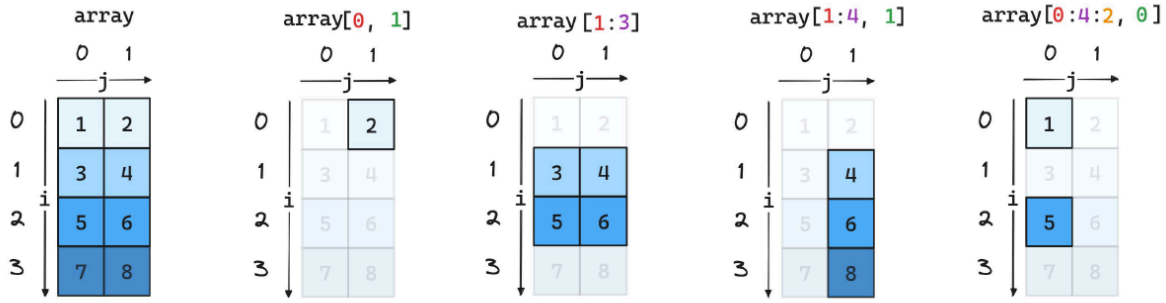


Indexing with Boolean Arrays

In [92]:
```
# Create a 1D array
arr = np.array([1, 2, 3, 4, 5])

print("Elements greater than 2:", arr[arr > 2])
print("Add 10 to all Elements less than 2:", arr[arr < 2]+20)
```

```
Elements greater than 2: [3 4 5]
Add 10 to all Elements less than 2: [21]
```

Visual understanding :



In this notebook, we have explored statistical operations, scientific operations, trigonometric operations, array manipulation, and delved deeply into slicing

Let's wrap up here. In the next part of the series, we will discuss *Numpy in Linear Algebra*. I try to keep my articles short, but this topic cannot be left incomplete.