NumPy

Introduction

- NumPy (Numerical Python) is a Python library used to work with numerical data.
- NumPy includes functions and data structures that can perform a wide variety of mathematical operations.
- To start using NumPy, we first need to import it:
  import numpy as np
- np is the most common name used to import numpy.
- In Python, lists are used to store data. NumPy provides an array structure for performing operations with data.
- NumPy arrays are **faster** and **more compact** than lists.
- A NumPy array can be created using the np.array() function, providing it a list as the argument:
- x = np.array([1, 2, 3, 4])
- NumPy arrays are homogeneous, meaning they can contain only a **single data type**, while lists can contain multiple different types of data.

Creating NumPy Arrays

```python
import numpy as np
# pprint for pretty printing
from pprint import pprint

# Creating arrays from different sources
# From a list or tuple
arr1 = np.array([1, 2, 3, 4, 5])
print("Array from list/tuple:")
pprint(arr1)
# From nested lists
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
print("Array from nested lists:")
pprint(arr2)
```

Column 0	Column 1	Column 2	Column 3	Column 4
1	**2**	**3**	**4**	**5**

	Column 0	Column 1	Column 2
Row 0	**1**	**2**	**3**
Row 1	**4**	**5**	**6**

NumPy Arrays

- Arrays have properties, which can be accessed using a dot.
- ndim returns the number of dimensions of the array.
- size returns the total number of elements of the array.
- shape returns a tuple of integers that indicate the number of elements stored along each dimension of the array.
- dtype returns the array type

```
import numpy as np
x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(x.ndim)  # 2
print(x.size)  # 9
print(x.shape) # (3, 3)
print(x.dtype) # int64
```

So, the array in our example is integer array has 2 dimensions, 9 elements and is a 3x3 matrix (3 rows and 3 columns).

NumPy Arrays

Changing one element to be float will force the whole array to be float

```
import numpy as np
x = np.array([1, 2, 3])
print(x.ndim)  # 1
print(x.size)  # 3
print(x.shape) # (3,)
print(x.dtype) # int64

y = np.array([1., 2, 3])
print(y.ndim)  # 1
print(y.size)  # 3
print(y.shape) # (3,)
print(y.dtype) # float64
```

Indexing and Slicing

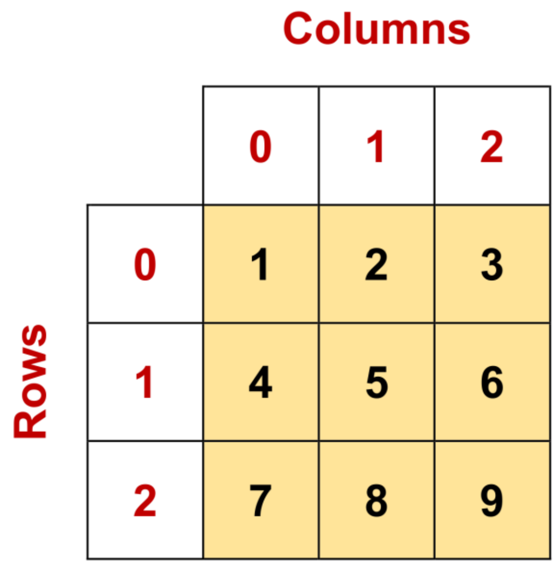
- NumPy arrays can be indexed and sliced the same way that Python lists are
- Negative indexes count from the end of the array, so, [-3:] will result in the last 3 elements.
- We can omit any of the three parameters in arr[lower:upper:step]

```
import numpy as np
from pprint import pprint

arr = np.array([10, 20, 30, 40, 50])
# Accessing elements
print(arr[0])      # Output: 10
print(arr[-1])     # Output: 50
# Slicing
print(arr[1:4])    # Output: [20, 30, 40]
print(arr[:2])     # Output: [10, 20]
print(arr[-3:])    # Output: [30, 40, 50]
print(arr[0:4:2]) # Output: [10, 30]
```

0	1	2	3	4
10	**20**	**30**	**40**	**50**
-5	-4	-3	-2	-1

Indexing and Slicing 2D-array

```
import numpy as np
from pprint import pprint

# Create a 2D NumPy matrix
matrix_2d = np.array([[1, 2, 3],[4, 5, 6],[7, 8, 9]])

print("2D Matrix:")
pprint(matrix_2d)

print("Element at Row 0, Column 1:")
print(matrix_2d[0, 1])    # Output: 2

print("Element at Row 1, Column 1:")
print(matrix_2d[1, 1])    # Output: 5

print("Row 0:")
print(matrix_2d[0])       # Output: [1 2 3]

print("Column 1:")
print(matrix_2d[:, 1])    # Output: [2 5 8]
```



Columns

Rows

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

numpy functions

Here are some of the most important numpy functions commonly used in academic research and data analysis:

- **numpy.array()**: This function is used to create numpy arrays, which are the fundamental data structure in numpy for handling numerical data.
- **numpy.zeros()** , **numpy.ones() and numpy.random.rand()**: These functions create arrays filled with zeros or ones or random between 0 and 1 respectively, which can be used as placeholders for data or for initializing matrices.
- **numpy.arange()**: It generates evenly spaced values within a specified range, which is particularly useful for creating sequences of numbers.
- **numpy.linspace()**: This function creates an array of evenly spaced values over a specified range, which can be useful for creating data points for plotting.
- **numpy.reshape()**: It allows you to change the shape of a numpy array, which can be crucial for preparing data for various operations or visualizations.

zeros , ones and rand

```python
import numpy as np
from pprint import pprint
# Using built-in functions
zeros_arr = np.zeros((3, 4), dtype = 'int16')
# Creates a 3x4 array of zeros
print("Array of zeros:")
pprint(zeros_arr)
ones_arr = np.ones((2, 3), dtype = 'int16')
# Creates a 2x3 array of ones
print("Array of ones:")
pprint(ones_arr)
random_arr = np.random.rand(3, 3)
# Creates a 3x3 array of random values between 0 and 1
print("Array of random values between 0 and 1:")
pprint(random_arr)
```

	Column 0	Column 1	Column 2	Column 3
Row 0	0	0	0	0
Row 1	0	0	0	0
Row 2	0	0	0	0

	Column 0	Column 1	Column 2
Row 0	1	1	1
Row 1	1	1	1

arange

np.arange(start,stop,step) allows you to create an array that contains a range of evenly spaced intervals between start and stop(not inclusive) similar to a Python **range()**

```
import numpy as np
arr = np.arange(10)
print(arr) # [0 1 2 3 4 5 6 7 8 9]

arr1 = np.arange(2, 10, 3)
print(arr1) #[2 5 8]

arr2 = np.arange(12, 3, -2)
print(arr2) #[12 10  8  6  4]

arr3 = np.arange(12, 7, -1.5)
print(arr3) #[12.  10.5  9.   7.5]
```

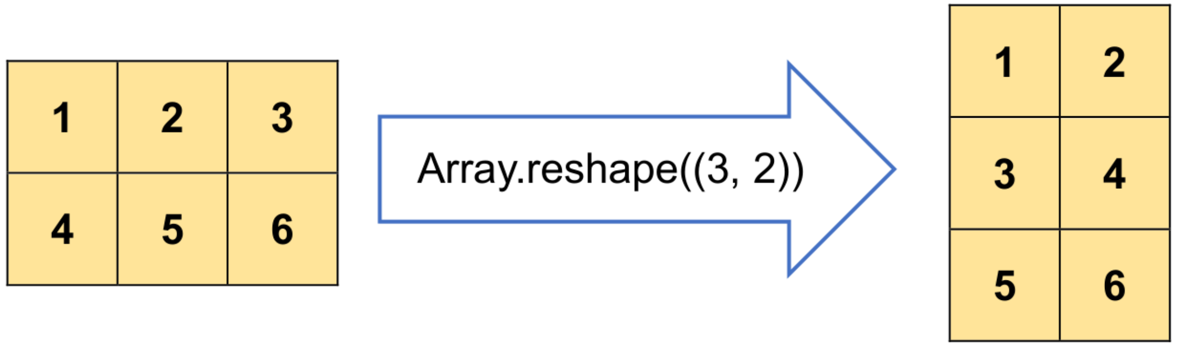
linspace

Use **numpy.linspace(start,stop,n)** to create an array of n evenly spaced values between start and stop(inclusive)

```
import numpy as np

# Use numpy.linspace() to create an array of 5 evenly
spaced values between 0 and 1
arr = np.linspace(0, 1, 5)
print(arr)       # [0. 0.25 0.5 0.75 1. ]
arr = np.linspace(0, 10, 6)
print(arr)       #[ 0. 2. 4. 6. 8. 10.]
```

reshape

```python
import numpy as np
from pprint import pprint

# Creating a 2D array
Arr = np.array([[1, 2, 3], [4, 5, 6]])

# Printing the shape of the array
print("Shape of the Array:")
print(Arr.shape)            # (2, 3)

# Displaying the original array
print("\nOriginal Array:")
pprint(Arr)

# Reshaping the array
Reshaped_Arr = Arr.reshape((3, 2))

print("\nReshaped Array:")
pprint(Reshaped_Arr)
```



arange + reshape

```
import numpy as np
from pprint import pprint

# Creating a 2D array
Arr = np.arange(1,10).reshape((3,3))
pprint(Arr)

#array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Advanced Indexing

NumPy offers a range of advanced indexing and index tricks that provide users with powerful tools to access and manipulate specific elements or subarrays within arrays, utilizing arrays as indices.

Advanced indexing in NumPy empowers you to utilize arrays or tuples as indices, enabling you to retrieve particular elements or subarrays from the array. Within advanced indexing, there are two main types:

- **Integer array indexing**
- **Boolean array indexing**.

Both methods open up new possibilities for array manipulation and data extraction.

Integer array indexing

```python
import numpy as np
from pprint import pprint  # Import the pprint function

data = np.array([10, 20, 30, 40, 50])

# Create an array of indices to select elements
indices = np.array([0, 2, 4])

# Use integer array indexing
selected_elements = data[indices]

# Print the selected elements
print("Selected Elements:")
pprint(selected_elements)  # [10, 30, 50]
```

0	1	2	3	4
10	**20**	**30**	**40**	**50**
-5	-4	-3	-2	-1

Boolean array indexing

0	1	2	3	4
10	**20**	**30**	**40**	**50**
-5	-4	-3	-2	-1

```
import numpy as np
from pprint import pprint

# Create a NumPy array
data = np.array([10, 20, 30, 40, 50])

# Create a Boolean array for indexing (select elements
greater than 30)
boolean_index = data > 30

print("Boolean Index:")
print(boolean_index)  # [False, False, False, True, True]

# select specific elements greater than 30
selected_elements = data[boolean_index]
print("\nSelected Elements (greater than 30):")
print(selected_elements)   # [40, 50]
```

Boolean array indexing

You can provide a condition as the index to select the elements that fulfill the given condition. To select the elements < 4:

```
x = np.arange(1, 10)
print(x[x<4])# [1 2 3]
```

Conditions can be combined using the & (and) and | (or) operators. For example, let's take the even numbers that are greater than 5:

```
print(x[(x>5) & (x%2==0)])# [6 8]
```

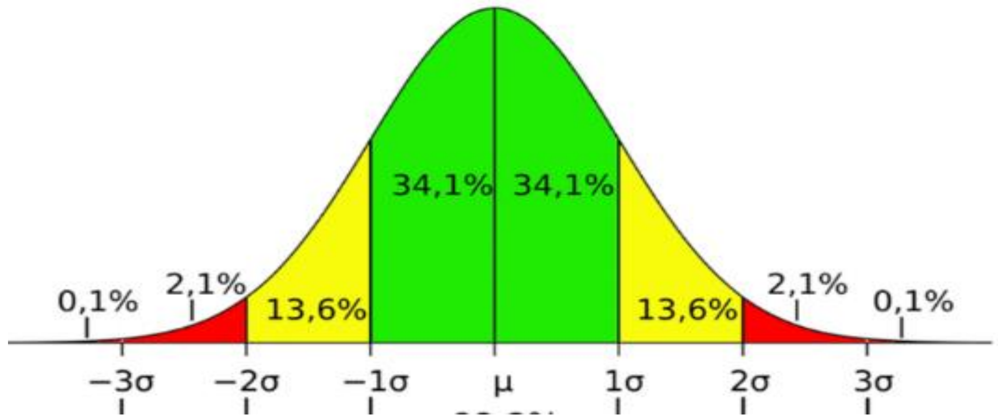
More numpy functions

- **numpy.mean()**, **numpy.median()**, and **numpy.std()**: These functions are used for basic statistical calculations on numpy arrays, such as calculating the mean, median, and standard deviation.
- **numpy.sum()** and **numpy.prod()**: These functions compute the sum and product of array elements, respectively, which can be essential for various mathematical operations.
- **numpy.min()** and **numpy.max()**: They return the minimum and maximum values in a numpy array, helping you identify extreme values in your data.
- **numpy.where()**: This function is employed to locate the indices where a specified condition is met within a numpy array, facilitating conditional data manipulation and selection.
- **numpy.dot()** : This function are used for matrix multiplication.

Statistics

- **mean**: The average of the values.
- **median**: The middle value of an ordered dataset.
- **standard deviation**: The measure of spread.

A low standard deviation indicates that the values tend to be close to the mean of the set, while a high standard deviation indicates that the values are spread out over a wider range.



Statistics

```python
import numpy as np

# Create a NumPy array 'data'
data = np.array([12, 15, 18, 22, 25])

mean = np.mean(data)
median = np.median(data)
std_dev = np.std(data)

print("Statistical Calculations:")
print("Mean:", mean)
print("Median:", median)
print("Standard Deviation:", std_dev)
```

```
Statistical Calculations:
Mean: 18.4
Median: 18.0
Standard Deviation: 4.673328578219168
```

Sum, prod, min, max

```
import numpy as np

# Create a NumPy array
arr = np.array([2, 3, 4, 5, 8, 7, 9, 11, 6, 10])

# using functions
print(np.sum(arr))  # 65
print(np.prod(arr)) # 39916800
print(np.min(arr))  # 2
print(np.max(arr))  # 11

# using methods
print(arr.sum())  # 65
print(arr.prod()) # 39916800
print(arr.min())  # 2
print(arr.max())  # 11
```

where (basic operation)

```python
# Import the numpy library and alias it as np
import numpy as np

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

# Define a condition: elements in the array greater than 3
condition = arr > 3

# Use np.where() to find the indices where the condition is met
indices = np.where(condition)

# Print indices
print('Indices:')
print(indices)

# Print the result
print('The result:')
print(arr[indices])
```

```
Indices:
array([3, 4])
The result: [4 5]
```

Replacing Values Based on Condition

Here, we use np.where(cond, element, arrray) to replace values meeting the condition with element, leaving other values unchanged

```
import numpy as np

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

# Define a condition to check elements greater than 3
condition = arr > 3

# Use numpy.where() to replace values based on the condition
# If the condition is True, replace with 10; otherwise, keep the
original value from arr
new_values = np.where(condition, 10, arr)

# Print the resulting array with replaced values
print(new_values)
```

Linear Algebra

We can use the usual arithmetic operators to multiply, add, subtract, and divide arrays with scalar numbers.

```
import numpy as np
from pprint import pprint

v1 = np.arange(0, 5)
print("Addition")
print(v1 + 2) # [2 3 4 5 6]
print("Subtraction")
print(v1 - 2) # [-2 -1  0  1  2]
print("multiplication")
print(v1 * 2) # [0 2 4 6 8]
print("Division")
print(v1 / 2) # [0.  0.5 1.  1.5 2. ]

# element-wise multiplication
arr1 = np.arange(9).reshape(3,3)
arr2 = arr1 + 5
print("element-by-element multiplication")
pprint(arr1*arr2)
```

```
Addition [2 3 4 5 6]
Subtraction [-2 -1 0 1 2]
multiplication [0 2 4 6 8]
Division [0. 0.5 1. 1.5 2. ]
element-by-element multiplication
array([[ 0, 6, 14],
       [ 24, 36, 50],
       [ 66, 84, 104]
])
```

Matrix multiplication

numpy.dot(): This function are used for matrix multiplication

```
import numpy as np
from pprint import pprint

# test 1-D
a = np.array([1, 2, 3, 4])
b = np.array([-1, 4, 3, 2])
c = np.dot(a, b)
print("NumPy 1-D np.dot(a, b) = ")
pprint(c)

# test 2-D
A = np.arange(9).reshape(3,3)
B = np.dot(A, A)
print("NumPy 2-D np.dot(A, A) = ")
pprint(B)
```

```
NumPy 1-D np.dot(a, b) = 24
NumPy 2-D np.dot(A, A) =
array([[ 15, 18, 21],
       [ 42, 54, 66],
       [ 69, 90, 111]
])
```