Contents

[NUMPY 2](#_Toc210314881)

[NUMPY ARRAYS 2](#_Toc210314882)

[KEY FEATURES OF NUMPY ARRAY 2](#_Toc210314883)

[CREATING NUMPY ARRAY 2](#_Toc210314884)

[ATTRIBUTES AND FUNCTIONS 7](#_Toc210314885)

[NUMPY INDEXING AND SELECTION 9](#_Toc210314886)

[INDEXING & SLICING IN 1-D ARRAY 9](#_Toc210314887)

[INDEXING & SLICING IN 2-D ARRAY 9](#_Toc210314888)

[BROADCASTING 10](#_Toc210314889)

[CONDITIONAL SELECTION 11](#_Toc210314890)

[NUMPY OPERATIONS 11](#_Toc210314891)

[ARITHMETIC OPERATIONS 12](#_Toc210314892)

# NUMPY

* NumPy is a Python library that stands for "Numerical Python". It’s a python library to create N-dimensional array
* It has ability to quickly broadcast functions
* Built in lot of features like linear algebra, statistical distributions, trigonometric functions and random number capabilities.

## NUMPY ARRAYS

* A NumPy array, also known as ndarray, is a central data structure in NumPy library. It is a multi-dimensional grid of elements of the same type. NumPy arrays can have any number of dimensions, but most commonly they are 1D (one-dimensional), 2D (two-dimensional), or 3D (three-dimensional).
* NumPy arrays are widely used in scientific computing, data analysis, machine learning, and numerical computations due to their efficiency and versatility.

### KEY FEATURES OF NUMPY ARRAY

Key features and characteristics of NumPy arrays:

1. **Homogeneous Data**: NumPy arrays contain elements of the **same data type**
2. **Fixed Size**: Once a NumPy array is created, its size is fixed and cannot be changed. To modify the size, a new array needs to be created.
3. **Fast and Efficient**: NumPy arrays are implemented in C, making them faster and more efficient compared to Python lists. They allow for vectorized operations, which can perform computations on entire arrays without the need for loops.
4. **Powerful Indexing and Slicing**: NumPy arrays support advanced indexing and slicing operations, allowing easy access to elements or sub-arrays based on specific conditions or criteria.
5. **Mathematical Operations**: NumPy arrays provide a wide range of mathematical functions and operations to perform calculations efficiently on arrays**, such as element-wise operations, linear algebra operations, statistical functions.**

### CREATING NUMPY ARRAY

A graph paper with text and arrow pointing at the center

Description automatically generated

#### CREATING NUMPY ARRAYS USING STANDARD LIST

|  |  |
| --- | --- |
| **import numpy as np**    **mylist= [1, 2, 3, 4, 5]**  **type(mylist) 🡪 list**  **# Creating a 1D array from standard list**  **arr1 = np.array(mylist)**  **print(arr1)🡪 array([1, 2, 3, 4, 5])**  **type(arr1) 🡪 numpy.ndarray** | **# Creating a 2D array from standard list**  **list\_2d = [[1,2,3],[4,5,6],[7,8,9]]**  **np\_2d = np.array(list\_2d)**  **O/p**  **array([[1, 2, 3],**  **[4, 5, 6],**  **[7, 8, 9]])** |
| **# Creating a 3D array from standard list**  **arr3 = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])**  **print(arr3)**  **# Output:**  **# [[[1 2]**  **# [3 4]]**    **# [[5 6]**  **# [7 8]]]** | np.array([[4], [5], [6]]) 🡨 3x1 ARRAY  **[[4]**  **[5]**  **[6]]**  np.array([4, 5, 6]) 🡨 1x3 ARRAY = **[4 5 6]** |

#### CREATING NUMPY ARRAYS USING NUMPY BUILT IN FUNCTIONS

##### arange

* The `**np.arange()`** function in NumPy is used to create an array with regularly spaced values within a specified range.
* **It is similar to the Python built-in `range()` function but returns an array instead of a list**.
* **SYNTAX**

**np.arange(start, stop, step, dtype=None)**

|  |  |
| --- | --- |
| start (optional): | The starting value of the sequence. If not provided, the default value is 0. |
| stop | * The end value of the sequence. * It is exclusive, so the generated sequence will stop before reaching this value. |
| step | * The step size between consecutive values in the sequence. If not provided, the default value is 1 |
| dtype | * The data type of the elements in the resulting array. * If not specified, NumPy will determine it based on the other input arguments |

###### EXAMPLE

|  |
| --- |
| **import numpy as np**    **# Example 1: Generate a sequence of numbers from 0 to 9**  **arr1 = np.arange(10)**  **print(arr1) # Output: [0 1 2 3 4 5 6 7 8 9]** |
| **# Example 2: Generate a sequence of even numbers from 2 to 10**  **arr2 = np.arange(2, 11, 2)**  **print(arr2) # Output: [2 4 6 8 10]** |
| **# Example 3: Generate a sequence of floating-point numbers from 0 to 1 with a step of 0.1**  **arr3 = np.arange(0, 1, 0.1)**  **print(arr3) # Output: [0. 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]** |
| **print(np.arange(0, 10, 2, dtype=float)) #OUTPUT [0. 2. 4. 6. 8.]** |

##### np.ones() and np.zeros()

* **np.ones()** and **np.zeros()** are used to create arrays filled with ones and zeros, respectively.
* These functions allow we to easily create arrays of desired shapes and sizes filled with the specified values.
* **SYNTAX** - This function creates an array of ones with the specified shape

**np.ones(shape, dtype=None, order='C')**

|  |  |
| --- | --- |
| shape | * The shape of the array, specified as a tuple of integers. * For example, **`(3, 4)`** creates a 2D array with 3 rows and 4 columns |
| dtype(optional): | * The data type of the elements in the array. * If not specified, the default data type is `float64`. |
| order(optional): | * The order in which the array is stored in memory. * It can be `'C'` for row-major (C-style) or `'F'` for column-major (Fortran-style). The default is `'C' |

* **SYNTAX** - This function creates an array of zeros with the specified shape

**np.zeros(shape, dtype=None, order='C')**

###### EXAMPLE

|  |  |
| --- | --- |
| **# Example 1: Create a 1D array of ones with 5 elements**  **arr1 = np.ones(5)**  **print(arr1)**  **# Output: [1. 1. 1. 1. 1.] 🡪 “.”(dot) represents the float** | **# Example 2: Create a 2D array of ones with 5 element with data type specified**  **print(np.ones((2,3), dtype=int))**  **# Output:**  **[[1 1 1]**  **[1 1 1]]** |
| **# Example 2: Create a 2D array of zeros with 3 rows and 4 columns(provides as tuples)**  **arr2 = np.zeros((3, 4)) 🡨 THE ROWS AND COLUMNS NEED TO BE PASSED AS TUPLE**  **print(arr2)**  **# Output:**  **[**  **[0. 0. 0. 0.]**  **[0. 0. 0. 0.]**  **[0. 0. 0. 0.]**  **]** | **# Example 3: Create a 3D array of ones with dimensions 2x3x2**  **arr3 = np.ones((2, 3, 2))**  **print(arr3)**  **# Output:**  **[**  **[**  **[1. 1.]**  **[1. 1.]**  **[1. 1.]**  **]**  **[**  **[1. 1.]**  **[1. 1.]**  **[1. 1.]**  **]**  **]** |
|  | |

##### np.linspace()

* the `np.linspace()` function is used to create an array with evenly spaced values over a specified interval. It is particularly useful when we want to generate a sequence of numbers with a specific number of elements.
* `np.linspace()` is commonly used in various scientific and numerical computing applications, such as plotting graphs, creating test datasets, and generating time series data.

**np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)**

|  |  |
| --- | --- |
| **start** | The starting value of the sequence |
| **stop** | The end value of the sequence |
| **num(optional):** | * The number of equally spaced values to generate between `start` and `stop`. * The default value is 50 i.e How many numbers we want between start and stop (including stop) |
| **endpoint(optional)** | * Whether or not to include the `stop` value in the sequence. * **If `True`, the sequence will include `stop`. The default value is `True** |
| **retstep(optional)** | * Whether or not to return the spacing between consecutive values. * If `True`, **the function will return a tuple containing the array and the step value**. * The default value is `False`. |
| **dtype(optional)** | * The data type of the elements in the resulting array. * If not specified, NumPy will determine it based on the other input arguments. |

###### EXAMPLE

|  |
| --- |
| **# Example 1: Generate a sequence of 10 numbers from 0 to 1 (including 1)**  **arr1 = np.linspace(0, 1, num=10)**  **print(arr1)**  **# Output: [0. 0.11111111 0.22222222 0.33333333 0.44444444 0.55555556**  **# 0.66666667 0.77777778 0.88888889 1. ]** |
| **# Example 2: Generate a sequence of 5 numbers from -2 to 2 (including -2 and 2)**  **arr2 = np.linspace(-2, 2, num=5)**  **print(arr2)**  **# Output: [-2. -1. 0. 1. 2.]** |
| **# Example 3: Generate a sequence of 6 numbers from 1 to 10 (excluding 10) and return the step value**  **arr3, step = np.linspace(1, 10, num=6, retstep=True) 🡨 Will return a tuple containing the array and the step value**  **print(arr3) # [ 1. 3.8 6.6 9.4 12.2 15. ]**  **print(step) 🡺 # 3.2 🡸 THIS IS THE SPACING BETWEEN THE ELEMENTS** |

#### CREATING NUMPY ARRAYS USING RANDOM LIBRARY

* The numpy.random library is a suite of functions based on pseudorandom number generation.
* It contains various functions for generating random data.

|  |
| --- |
| WHAT IS PROBABLITY DISTRIBUTION?  **A probability distribution tells us : What is the chance of each outcome**  WHAT IS UNIFORM DISTRIBUTION?   * A **uniform distribution** is a type of probability distribution in which **all outcomes are equally likely**. * Every value within a given range has the **same probability**. * The graph of a uniform distribution is a **rectangle** (flat line).      * The x-axis shows values from **a to b**. * The y-axis shows a constant height ( **1/(b-a)** ). * The area under the curve equals 1 (total probability).   Types of Uniform Distribution  1.Discrete Uniform Distribution   * Example: Rolling a fair die. * Each outcome (1 to 6) has a probability of 1/6.   2. Continuous Uniform Distribution   * Example: Randomly picking a number between 0 and 1. * Every number in the interval [0, 1] has equal probability density. |

##### random.rand

* **np.random.rand** creates an array of specified shapes and fills it with random values **from the uniform distribution over [0,1)**

|  |  |
| --- | --- |
| import numpy as np    **# Generate a 1-D array containing 5 random floats**  random\_floats = np.random.rand(5)  print(random\_floats)  OUTPUT  **[0.12319511 0.73894491 0.05873792 0.23909406 0.8188274 ]** | **# Generate a 2-D array containing random floats. The size of the array is 3 rows by 2 columns.**  random\_floats\_2d = np.random.rand(3, 2)  print(random\_floats\_2d)  OUTPUT **[[0.15027467 0.70690855]**  **[0.57395591 0.77450563]**  **[0.69494225 0.75765151]]** |

* In both cases, the numbers generated are uniformly distributed over the interval `[0, 1)`, meaning they can be any decimal number from 0 up to but not including 1.
* **np.random.rand() is uniform distribution in the interval [0, 1)**

##### random.randint

* **np.random.randint()** returns random integers from the "**discrete uniform**" distribution of the specified dtype in the "half-open" interval [`low`, `high`). If `high` is None (the default), then results are from [0, `low`).



|  |  |  |
| --- | --- | --- |
| **# Generate a single random integer between 0 (inclusive) and 10 (exclusive)**  random\_int = np.random.randint(10)  print(random\_int) | **# Generate a single random integer between 5 (inclusive) and 20 (exclusive)**  random\_int = np.random.randint(5, 20)  print(random\_int) | |
| **# Generate a 1-D array containing 5 random integers between 0 (inclusive) and 10 (exclusive)**  random\_ints = np.random.randint(10, size=5)  print(random\_ints) | **# Generate a 1-D array containing 5 random integers between 10 (inclusive) and 20 (exclusive)**  np.random.randint(10,20,5)  OUTPUT - [19 10 19 17 19] | |
| **# Generate a 2-D array containing random integers between 0 (inclusive) and 10 (exclusive).**  **# The size of the array is 3 rows by 2 columns.**  random\_ints\_2d = np.random.randint(10, size=(3, 2))  print(random\_ints\_2d) | | **OUTPUT**  **[[ 0 2]**  **[17 2]**  **[ 7 13]]** |

* In each case, the numbers are integers randomly sampled from the specified interval. The **size** parameter determines the shape of the output array.

##### random.randn

* **np.random.randn()** **returns a sample (or samples) from the "standard normal" distribution, also known as a Gaussian distribution.**
* **np.random.randn()** is centered at zero and has a standard deviation of one, which means the values are spread out around zero, and about 68% of the values will be within -1 and 1.

|  |
| --- |
| **Normal Distribution** — A bell-shaped curve that shows how data is symmetrically distributed around the mean.    WHAT IS STANDARD NORMAL DISTRIBUTION? |

|  |
| --- |
| # Generate a single random number from the standard normal distribution  random\_num = np.random.randn()  print(random\_num) |
| **# Generate a 1-D array containing 5 random numbers from the standard normal distribution**  random\_nums = np.random.randn(5)  print(random\_nums) |
| **# Generate a 2-D array containing random numbers from the standard normal distribution. The size of the array is 3 rows by 2 columns.**  random\_nums\_2d = np.random.randn(3, 2)  print(random\_nums\_2d) |

* In each case, the numbers are randomly sampled from the standard normal distribution.Hence the value closer to zero is more likely to appear that farther to zero

##### random.seed

* **np.random.seed()**is used to initialize the pseudorandom number generator. This can be useful to reproduce the results from the random number generator.
* In random processes, a seed is required to start the generation of a sequence of numbers, which appear random. If we use the same seed, we will get the same sequence of numbers, which can be useful for debugging and testing.

|  |  |
| --- | --- |
| # Set the seed np.random.seed(42)  # Generate a random number print(np.random.rand(5))  # Reset the seed np.random.seed(42)  # Generate another random number print("Random Number with same seed", np.random.rand(5)) | **[0.37454012 0.95071431 0.73199394 0.59865848 0.15601864]**  **Random Number with same seed [0.37454012 0.95071431 0.73199394 0.59865848 0.15601864]**   * In this example, we will see that the two random numbers printed are the same, because we reset the seed before generating the second number. If the seed was not reset, the two numbers would be different. * Seed number is completely arbitrary choice. The set random numbers will be same till the seed number is same |

### ATTRIBUTES AND FUNCTIONS

#### FUNCTIONS

##### reshape()

* np.reshape() function in numpy gives a new shape to an array without changing its data.
* It creates a new array and does not modify the original array itself.

|  |  |
| --- | --- |
| import numpy as np    # Create a 1-D array  a = np.arange(6)  print(a) **# Outputs: [0 1 2 3 4 5]**    # Reshape the 1-D array to a 2-D array with 2 rows and 3 columns  b = np.reshape(a, (2, 3))  print(b)  **# Outputs:**  **# [[0 1 2]**  **# [3 4 5]]** | In this example the original 1-D array with 6 elements is reshaped into a 2-D array with 2 rows and 3 columns. The total number of elements before and after reshaping remains the same. |
| We can also use the reshape method directly on the numpy array:  # Create a 1-D array  a = np.arange(6)  print(a) **# Outputs: [0 1 2 3 4 5]**  c = a.reshape((2, 3))  print(c) | |

##### min() and max()

|  |  |
| --- | --- |
| ran\_10 = np.random.randint(10, size=20) print(ran\_10)  # Max and Mini print("Maximum Value =", ran\_10.max()) print("Index of Maximum Value =",ran\_10.argmax()) print("Minimum Value =",ran\_10.min()) print("Index of Minimum Value =",ran\_10.argmin()) | OUTPUT  **[6 8 8 5 2 8 5 1 8 2 5 7 9 2 9 0 4 8 7 4]**  **Maximum Value = 9**  **Index of Maximum Value = 12**  **Minimum Value = 0**  **Index of Minimum Value = 15** |

#### ATTRIBUTES

##### shape

* The `shape` attribute in numpy is used to get the current shape of an array.
* It returns a tuple representing the dimensionality of the array.
* The length of the tuple is the rank or number of dimensions (ndim) of the array. The values in the tuple indicate the size of the array in each dimension.

|  |  |
| --- | --- |
| **# Create a 1-D array**  a = np.array([1, 2, 3])  print(a.shape) # Outputs: (3,) | **# Create a 2-D array**  b = np.array([[1, 2, 3], [4, 5, 6]])  print(b.shape) # Outputs: (2, 3) |
| **# Create a 3-D array**  c = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])  print(c.shape) # Outputs: (2, 2, 3) | ran\_one\_d = np.random.randint(10, size=20) print(ran\_one\_d) print(ran\_one\_d.shape) ran\_two\_d = ran\_one\_d.reshape(4,5) print(ran\_two\_d) print(ran\_two\_d.shape)  OUTPUT  [9 3 2 2 9 7 0 0 9 3 5 7 0 7 1 1 9 5 4 0]  (20,) 🡨 1 Dimensional Array  [[9 3 2 2 9]  [7 0 0 9 3]  [5 7 0 7 1]  [1 9 5 4 0]]  (4, 5) 🡨 2 Dimensional Array |

## NUMPY INDEXING AND SELECTION

Indexing and selection

1. **GRABBING SINGLE ELEMENT**
2. **GRABBING A SLICE OF ELEMENTS**
3. **BROADCASTING SELECTIONS**
4. **INDEXING AND SELECTION IN 2 DIMENSIONS**
5. **CONDITIONAL SELECTION**

### INDEXING & SLICING IN 1-D ARRAY

|  |  |
| --- | --- |
| arr\_index = np.arange(0, 11)  print(arr\_index) | **[ 0 1 2 3 4 5 6 7 8 9 10]** |
| print(arr\_index[8]) | **8** |
| print(arr\_index[1:5]) | **[1 2 3 4]** |
| print(arr\_index[0:5]) | **[0 1 2 3 4]** |
| print(arr\_index[:5]) | **[0 1 2 3 4]** |
| print(arr\_index[5:])  In this case we don’t need to know the length of the array to print the array elements | **[ 5 6 7 8 9 10]** |
| print(arr\_index[:-3]) | **[0 1 2 3 4 5 6 7]** |

### INDEXING & SLICING IN 2-D ARRAY

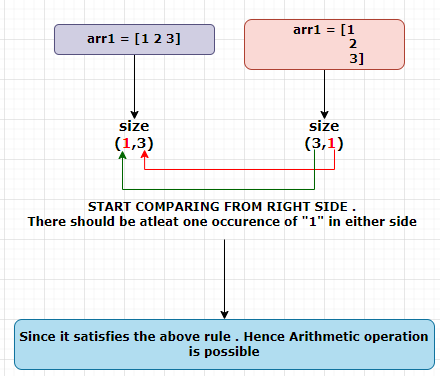
|  |  |  |
| --- | --- | --- |
| **INDEXING** | | |
| arr\_2d = np.array([[5, 10, 15], [20, 25, 30], [35, 40, 45]]) print(arr\_2d) | | **[[ 5 10 15]**  **[20 25 30]**  **[35 40 45]]** |
| print(arr\_2d.shape) | | **(3, 3)** |
| **ACCESSING SPECIFIC SINGLE ROW**  print(arr\_2d[0]) print(arr\_2d[2]) | | **[ 5 10 15]**  **[35 40 45]** |
| **ACCESSING SPECIFIC SINGLE COLUMN**  **print(arr\_2d[:, 1])** | | **[10 25 40]** |
| **ACCESSING ELEMENT**  print(arr\_2d[1][1])  OR  **print(arr\_2d[1,1])** | | **25** |
| **SLICING** | | |
|  | Slice the highlighted array element  **print(arr\_2d[:2,1:])**  :2🡪 Up to 2nd Row(Excluding 2nd )  1: 🡪 Everything after 1st column | **[[10 15]**  **[25 30]]** |
|  | print(arr\_2d[1:,1]) | **[25 40]** |

### BROADCASTING

* Broadcasting in NumPy is a powerful feature that **allows arrays of different shapes to be used in arithmetic operations.**

#### BROADCASTING RULES

1. The arithmetic operation can be done if the arrays are of same dimension
2. If the arrays are not of same dimension, the arithmetic operation is to be only possible if following rules are satisfied otherwise the operation will give broadcasting error



|  |  |  |
| --- | --- | --- |
| **import numpy as np  arr1 = np.array([[1, 2, 3]]) arr2 = np.array([[1], [2], [3]]) print(arr1.shape) 🡪 (1, 3) print(arr2.shape) 🡪 (3, 1) print(arr1 + arr2)** | OUTPUT  **[[2 3 4]**  **[3 4 5]**  **[4 5 6]]** | The output array will be of size of max size of comparison |

##### BROADCASTING ERROR SCENARIO

A diagram of a graph

Description automatically generated

|  |  |
| --- | --- |
| **import numpy as np arr1 = np.array([[1, 2, 3]]) arr2 = np.array([[1, 2]]) print(arr1 +arr2)** |  |

### CONDITIONAL SELECTION

* Conditional selection in NumPy allows us to select elements from an array based on a condition.
* The condition is usually provided in the form of a boolean expression that **returns a boolean array of the same shape.**
* We can then use this boolean array to index the original array, which will return only the elements where the boolean array is `True`.

|  |  |
| --- | --- |
|  | import numpy as np  # Create a numpy array  arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])    # Create a boolean array based on a condition  bool\_arr = arr > 5    # Use the boolean array to index the original array  new\_arr = arr[bool\_arr]    print(new\_arr) # Prints: [6 7 8 9]   * In this example, `arr > 5` is the condition, which creates a new boolean array where each element is `True` if the corresponding element in `arr` is greater than 5 and `False` otherwise. Then, `arr[bool\_arr]` indexes `arr` using this boolean array, returning a new array with only the elements where the condition is `True`. |
| The above steps can be combined into a single line as follows:  **new\_arr = arr[arr > 5]** |

## NUMPY OPERATIONS

### ARITHMETIC OPERATIONS

* Arithmetic between arrays and arithmetic between a scalar are two operations commonly performed in programming and data analysis, particularly in languages such as Python, R, or MATLAB, which heavily use array-based data structures.

#### ARITHMETIC BETWEEN ARRAYS

* This is also called element-wise operations. In this case, arithmetic operations are performed on corresponding elements of the arrays.
* For example, if A and B are two arrays of the same size, the sum A + B will be an array where each element is the sum of the corresponding elements in A and B. Note that for these operations to work, the arrays usually need to be of the same size or shape, or be broadcastable to a common shape.

|  |  |
| --- | --- |
| arr\_10 = np.arange(0, 11)  print(arr\_10) | **[ 0 1 2 3 4 5 6 7 8 9 10]** |
| ADDING ARRAY ELEMENTS  add\_arr = arr\_10 + arr\_10 print(add\_arr) | **[ 0 2 4 6 8 10 12 14 16 18 20]** |
| SUBSTRACT ARRAY ELEMENT  sub\_arr = arr\_10 - arr\_10 print(sub\_arr) | **[0 0 0 0 0 0 0 0 0 0 0]** |
| MULTIPLY ARRAY ELEMENT  mul\_arr = arr\_10 \* arr\_10 print(mul\_arr) | **[ 0 1 4 9 16 25 36 49 64 81 100]** |
| DIVISION OF ARRAY ELEMENT  div\_arr = arr\_10 / arr\_10 print(div\_arr) | **[nan 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]** |
| **We will get warning for “0/0” case but we will get result for other elements** | |

#### ARITHMETIC BETWEEN A SCALAR AND AN ARRAY

* In this case, the scalar is applied to every element of the array.
* For example, if A is an array and x is a scalar, the product x \* A will be an array where each element is the product of x and the corresponding element in A. This is made possible by a process called broadcasting, where the scalar is conceptually expanded to have the same shape as the array, allowing element-wise operations.

|  |  |
| --- | --- |
| arr\_10 = np.arange(0, 11)  print(arr\_10) | **[ 0 1 2 3 4 5 6 7 8 9 10]** |
| ADD TO ARRAY ELEMENT  add\_5\_to\_arr = arr\_10 + 5  print(add\_5\_to\_arr) | **[ 5 6 7 8 9 10 11 12 13 14 15]** |
| SUBSTRACT TO ARRAY ELEMENT  sub\_2\_to\_arr = arr\_10 + 5  print(add\_5\_to\_arr) | **[-2 -1 0 1 2 3 4 5 6 7 8]** |
| DIVISION  div\_arr\_scaler = 1/arr\_10 print(div\_arr\_scaler) | **[ inf 1. 0.5 0.33333333 0.25 0.2**  **0.16666667 0.14285714 0.125 0.11111111 0.1 ]** |
|  | |

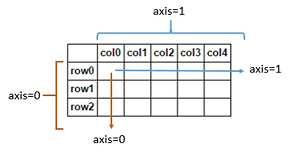
#### UNIVERSAL ARRAY FUNCTIONS

* Documentation : <https://docs.scipy.org/doc/numpy-1.10.1/reference/ufuncs.html#math-operations>

|  |  |
| --- | --- |
| arr\_10 = np.arange(0, 11) | **[ 0 1 2 3 4 5 6 7 8 9 10]** |
| arr\_square = np.**square**(arr\_10) print(arr\_square) | **[ 0 1 4 9 16 25 36 49 64 81 100]** |
| arr\_sqrt = np.**sqrt**(arr\_10) print(arr\_sqrt) | **[0. 1. 1.41421356 1.73205081 2. 2.23606798**  **2.44948974 2.64575131 2.82842712 3. 3.16227766]** |
| arr\_non\_zeros = np.arange(1, 11) arr\_log = np.**log**(arr\_non\_zeros) print(arr\_log) | **[0. 0.69314718 1.09861229 1.38629436 1.60943791 1.79175947**  **1.94591015 2.07944154 2.19722458 2.30258509]** |
| arr\_sum = arr\_10.**sum**() print(arr\_sum) | **55** |
| arr\_mean = arr\_10.**mean**() print(arr\_mean) | **5.0** |
| arr\_max = arr\_10.**max**() print(arr\_max) | **10** |
| **VARIANCE**  arr\_variance = arr\_10.var() print(arr\_variance) | **10.0** |
| **STANDARD DEVIATION**  arr\_std = arr\_10.std() print(arr\_std) | **3.1622776601683795** |

##### AXIS PARAMETER IN SUM UNIVERSAL FUNCTION

**The axis parameter in the sum function is used to determine the dimension along which the sum is computed.**



|  |  |
| --- | --- |
| arr\_sum = np.arange(1,21) print(arr\_sum) | **[ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20]** |
| arr\_sum\_1\_d = arr\_sum.**sum**() print(arr\_sum\_1\_d) | **210**  **When we have a 1D array (a simple list of numbers), the sum function without specifying an axis will just add up all the numbers in the array.** |

* When we have a 2D array (a matrix or a list of lists), things get more interesting:
  + If we use the sum function **with axis=0, it will compute the sum along the columns**. That is, it will add up all the numbers in each column, resulting in a 1D array of sums.
  + If we use the sum function with **axis=1, it will compute the sum along the rows.** That is, it will add up all the numbers in each row, resulting in a 1D array of sums.
* For arrays with more than two dimensions, we can use the axis parameter to specify which dimension we want to sum over. The axis parameter can take any integer from 0 up to the number of dimensions minus 1.
* Remember that in Python and most other programming languages, indexing starts at 0, so axis=0 refers to the first dimension, axis=1 to the second, and so on.

|  |  |
| --- | --- |
| arr\_sum = np.arange(1,21) print(arr\_sum) | **[ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20]** |
| **Lets create a 2 D array**  arr\_2\_d = arr\_sum.**reshape**(4,5) print(arr\_2\_d) | **[[ 1 2 3 4 5]**  **[ 6 7 8 9 10]**  **[11 12 13 14 15]**  **[16 17 18 19 20]]** |
| **ADD BY COLUMN IN 2 D ARRAY**  **arr\_2\_d\_add\_by\_col = arr\_2\_d.sum(axis=0) print(arr\_2\_d\_add\_by\_col)** | **[34 38 42 46 50]** |
| **ADD BY ROW IN 2 D ARRAY**  **arr\_2\_d\_add\_by\_row = arr\_2\_d.sum(axis=1) print(arr\_2\_d\_add\_by\_row)** | **[15 40 65 90]** |