**NUMPY & PANDAS FOR DATA MANIPULATION**

**NumPy** is a core Python library for numerical computing, built for handling large arrays and matrices efficiently.

* **ndarray object** – Stores homogeneous data in n-dimensional arrays for fast processing.
* **Vectorized operations** – Perform element-wise calculations without explicit loops.
* **Broadcasting** – Apply operations across arrays of different shapes.
* **Linear algebra functions** – Matrix multiplication, inversion, eigenvalues, etc.
* **Statistical tools** – Mean, median, standard deviation, and more.
* **Fourier transforms** – Fast computation for signal and image processing.
* **Integration with other libraries** – Works seamlessly with Pandas, SciPy, and scikit-learn.

**NOTE**:

* NumPy arrays are homogeneous, meaning all elements must be the same type, allowing efficient computation.
* Vectorized operations in NumPy can be 10 to 100 times faster than equivalent Python loops.

**What is NumPy Used for?**

With NumPy, you can perform a wide range of numerical operations, including:

* Creating and manipulating arrays.
* Performing element-wise and matrix operations.
* Generating random numbers and statistical calculations.
* Conducting linear algebra operations.
* Working with Fourier transformations.
* Handling missing values efficiently in datasets.

**Why Learn NumPy?**

* NumPy speeds up math operations like addition and multiplication on large groups of numbers compared to regular Python..
* It’s good for handling large lists of numbers (arrays), so you don’t have to write complicated loops.
* It gives ready-to-use functions for statistics, algebra and random numbers.
* Libraries like Pandas, SciPy, TensorFlow and many others are built on top of NumPy.
* NumPy uses less memory and stores data more efficiently, which matters when working with lots of data.

**NumPy Basics:**

This section covers the fundamentals of NumPy, including installation, importing the library and understanding its core functionalities. You will learn about the advantages of NumPy over Python lists and how to set up your environment for efficient numerical computing.

**NumPy Arrays:**

NumPy arrays (ndarrays) are the backbone of the library. This section covers how to create and manipulate arrays effectively for data storage and processing

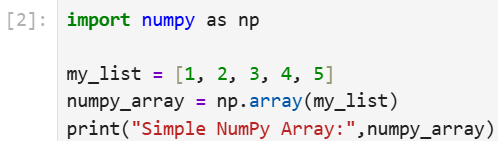
**Different Ways to Create Numpy Arrays in Python:**

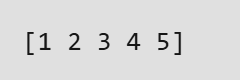
Creating NumPy arrays is a fundamental aspect of working with numerical data in Python. NumPy provides various methods to create arrays efficiently, catering to different needs and scenarios. In this article, we will see how we can create NumPy arrays using different ways and methods.

**Create Numpy Arrays Using Lists or Tuples**

The simplest way to create a NumPy array is by passing a Python list or tuple to the numpy.array() function. This method creates a one-dimensional array.

**EXAMPLE:**

****

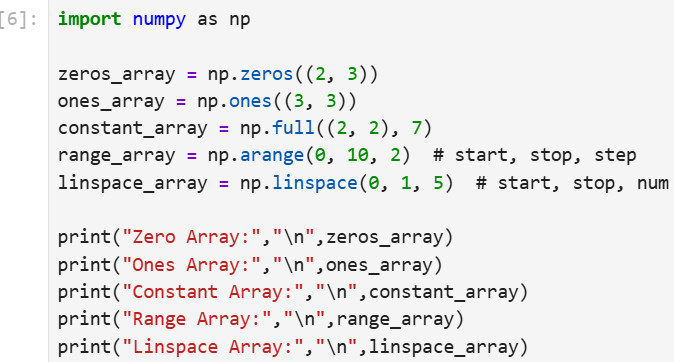
****

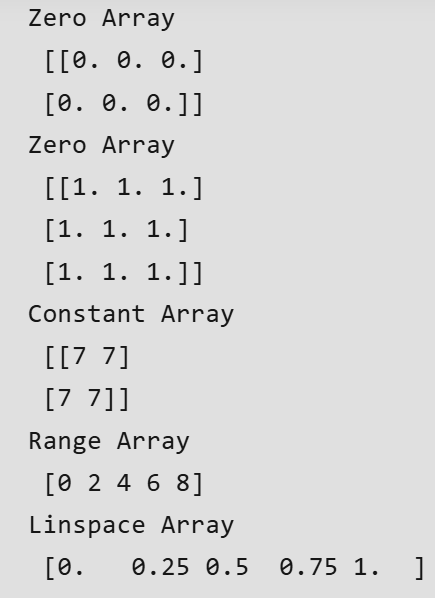
**Initialize a Python NumPy Array Using Special Functions:**

NumPy provides several built-in functions to generate arrays with specific properties.

* np.zeros(): Creates an array filled with zeros.
* np.ones(): Creates an array filled with ones.
* np.full(): Creates an array filled with a specified value.
* np.arange(): Creates an array with values that are evenly spaced within a given range.
* np.linspace(): Creates an array with values that are evenly spaced over a specified interval.

**EXAMPLE:**

****

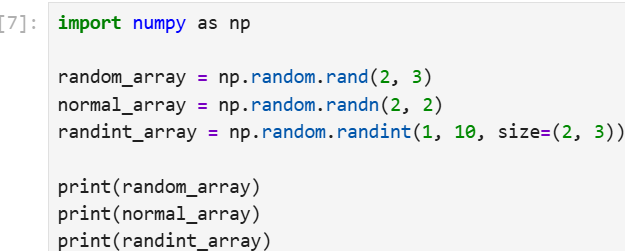
****

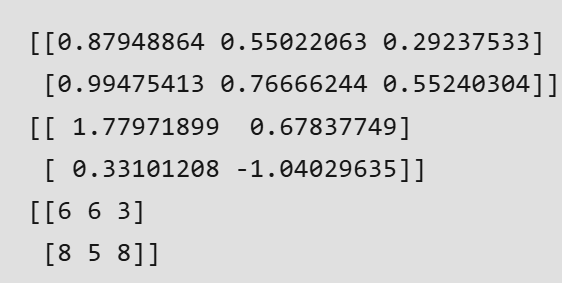
**Create Python Numpy Arrays Using Random Number Generation:**

NumPy provides functions to create arrays filled with random numbers.

* np.random.rand(): Creates an array of specified shape and fills it with random values sampled from a uniform distribution over [0, 1).
* np.random.randn(): Creates an array of specified shape and fills it with random values sampled from a standard normal distribution.
* np.random.randint(): Creates an array of specified shape and fills it with random integers within a given range.

**EXAMPLE:**

****

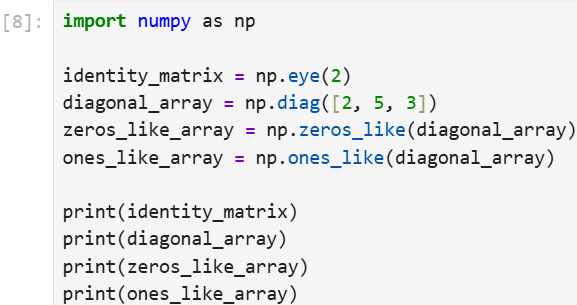
****

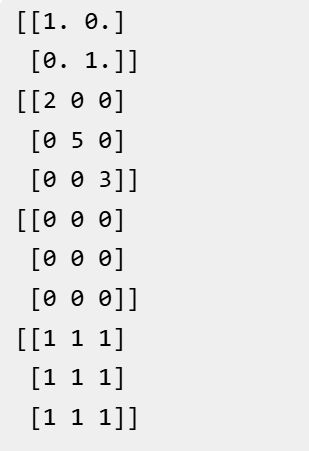
**Create Python Numpy Arrays Using Matrix Creation Routines:**

NumPy provides functions to create specific types of matrices.

* np.eye(): Creates an identity matrix of specified size.
* np.diag(): Constructs a diagonal array.
* np.zeros\_like(): Creates an array of zeros with the same shape and type as a given array.
* np.ones\_like(): Creates an array of ones with the same shape and type as a given array.

**EXAMPLE:**

****

****

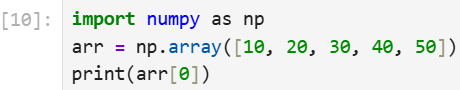
**Numpy Array Indexing:**

**Array indexing**in NumPy refers to the method of accessing specific elements or subsets of data within an array. This feature allows us to retrieve, modify and manipulate data at specific positions or ranges helps in making it easier to work with large datasets. In this article, we’ll see the different ways to index and slice NumPy arrays which helps us to work with our data more effectively.

**1. Accessing Elements in 1D Arrays**

A 1D NumPy array is a sequence of values with positions called indices which starts at 0. We access elements by using these indices in square brackets like arr[0] for the first element. Negative indices count from the end so arr[-1] gives the last element.

**EXAMPLE:**

****

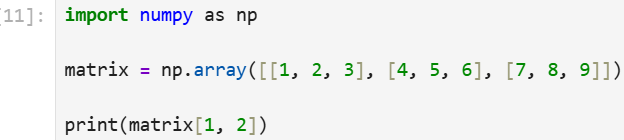
**OUTPUT: 10**

**Accessing Elements in Multidimensional Arrays**

In this we will see how to access elements in both 2D and 3D arrays using specific indices.

**2D Arrays**: We can access elements by specifying both row and column indices like**matrix[row, column]**.

**EXAMPLE**:



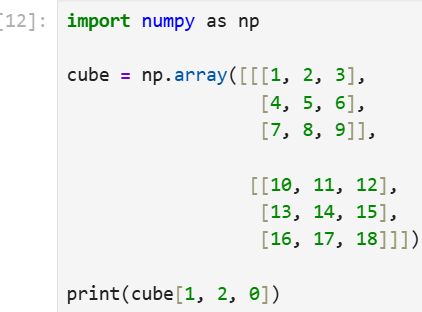
**OUTPUT: 6**

**3D Arrays**: It can be visualized as a stack of 2D arrays, we need three indices-

1. **Depth:** Specifies the 2D slice.
2. **Row:** Specifies the row within the slice.
3. **Column:** Specifies the column within the row.

We can access elements by specifying row, column and depth indices like **matrix[depth, row, column]**.

**EXAMPLE:**

****

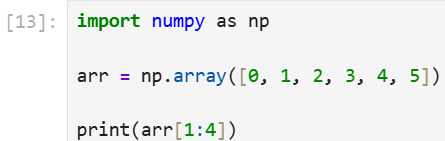
**OUTPUT: 16**

**Slicing Arrays**

It allows us to extract a range of elements using the format **start:stop:step**. This can be done for both 1D and multidimensional arrays which allows us to select ranges of elements or submatrices easily.

**Slicing 1D Arrays:** For a 1D array, slicing returns a subset of elements between the **start**and **stop**indices.

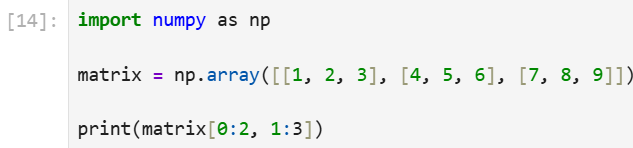
**EXAMPLE:**

****

**OUTPUT: [1, 2, 3]**

**Slicing Multidimensional Arrays:** In this slicing can be applied to each dimension separately which allows us to extract submatrices or smaller blocks of data.

**EXAMPLE:**

****

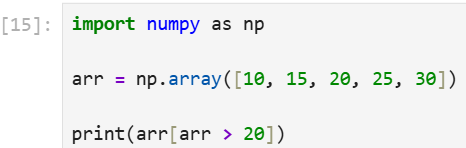
**OUTPUT: [[2, 3]**

**[5, 6]]**

**Boolean Indexing:**

It allows us to filter elements from an array based on a **condition**and returns only those that meet it. We create a boolean array from a condition and use it to select elements and can combine conditions with logical operators.

**EXAMPLE:**

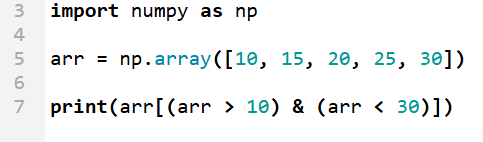
****

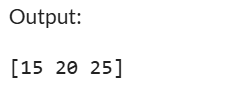
****

The condition **arr > 20** returns **True** for elements greater than 20 so only 25 and 30 are selected and printed.

We can also use logical operators like **& (AND), | (OR) and ~ (NOT)**to combine conditions.

**EXAMPLE 2:**

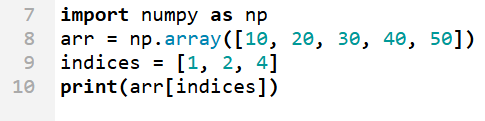
****

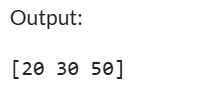
****

**Fancy Indexing:**

It is also known as**Advanced Indexing** which allows us access elements of an array by using another array or list of indices. This allows selecting multiple elements at once even if they are not next to each other which makes it easy to pick specific values from different positions in the array.

**EXAMPLE:**

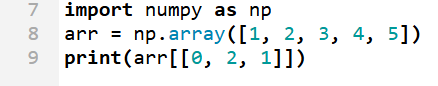
****

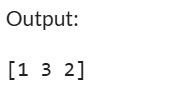
****

**Integer Array Indexing:**

It is similar to fancy indexing and uses an array of integers to select multiple elements from another array. This method allows us to access elements at specific, non-adjacent positions which makes it useful for extracting scattered data points.

**EXAMPLE:**

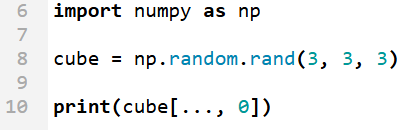
****

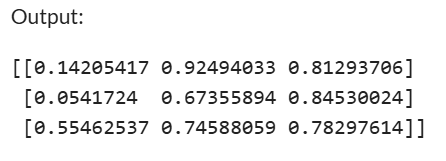
****

**Ellipsis (...) in Indexing:**

The **ellipsis (...)** can be used to select all dimensions which are not explicitly mentioned. This is helpful in multidimensional arrays when we don’t want to specify every dimension.

**EXAMPLE:**

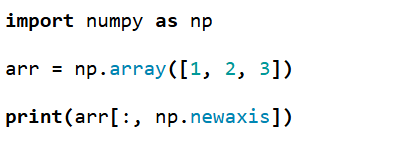
****

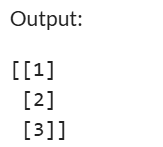
****

**Using np.newaxis to Add New Dimensions:**

The **np.newaxis keyword** adds a new axis to the array which helps in converting a 1D array into a row or column vector.

**EXAMPLE:**

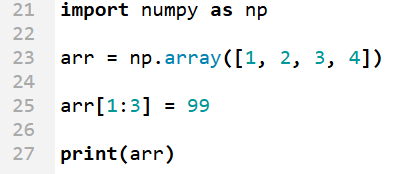
****

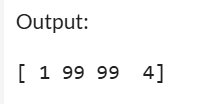
****

**Modifying Array Elements:**

We can modify array elements directly by using **indexing**or **slicing**. This makes it easy to update specific elements or ranges of elements in an array.

**EXAMPLE:**

****

****

**OPERATIONS IN NUMPY:**

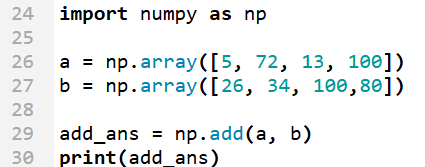
**NumPy - Arithmetic Operations**

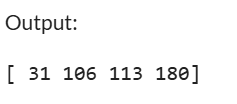
**Arithmetic operations** are used for numerical computation and we can perform them on arrays using NumPy. With NumPy we can quickly add, subtract, multiply, divide and get power of elements in an array. NumPy performs these operations even with large amounts of data. In this article, we’ll see at the basic arithmetic functions in NumPy and show how to use them for simple calculations.

**1. Addition of Arrays**

Addition is an arithmetic operation where the corresponding elements of two arrays are added together. In NumPy the addition of two arrays is done using the **np.add()**function.

**EXAMPLE:**

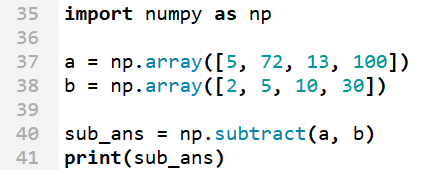
****

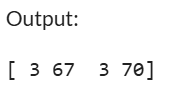
****

**Subtraction of Arrays**

We can subtract two arrays element-wise using the**np.subtract()** function. This function subtracts each element of the second array from the corresponding element in the first array.

**EXAMPLE:**

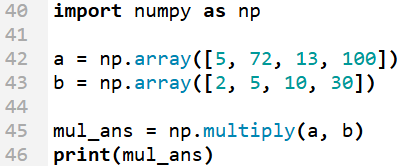
****

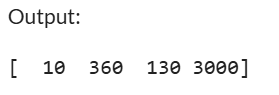
****

**Multiplication of Arrays**

Multiplication in NumPy can be done element-wise using the **np.multiply()** function. This multiplies corresponding elements of two arrays.

**EXAMPLE:**

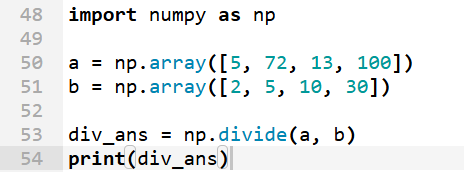
****

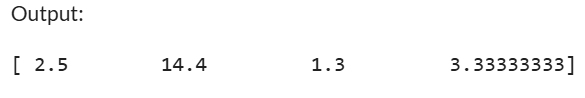
****

**Division of Arrays**

Division is another important operation that is performed element-wise using the **np.divide()** function. This divides each element of the first array by the corresponding element in the second array.

**EXAMPLE:**

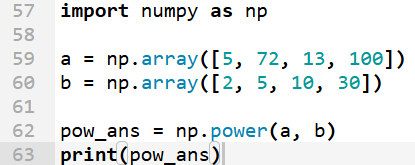
****

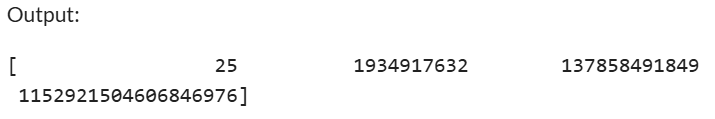
****

**Exponentiation (Power)**

It allows us to raise each element in an array to a specified power. In NumPy, this can be done using the **np.power()** function.

**EXAMPLE:**

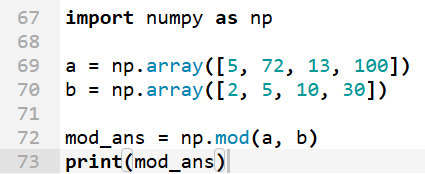
****

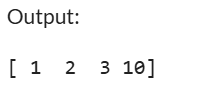
****

**Modulus Operation**

It finds the remainder when one number is divided by another. In NumPy, you can use the **np.mod()** function to calculate the modulus element-wise between two arrays.

**EXAMPLE:**

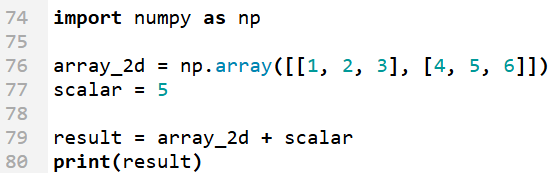
****

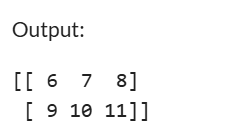
****

**NumPy Array Broadcasting:**

Broadcasting in NumPy allows us to perform arithmetic operations on arrays of different shapes without reshaping them. It automatically adjusts the smaller array to match the larger array's shape by replicating its values along the necessary dimensions. This makes element-wise operations more efficient by reducing memory usage and eliminating the need for loops. In this article, we will see how broadcasting works.

**EXAMPLE:**

****

****

**Working of Broadcasting in NumPy:**

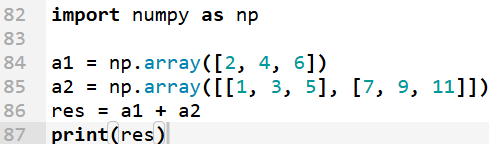
Broadcasting applies specific rules to find whether two arrays can be aligned for operations or not that are:

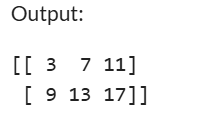
1. **Check Dimensions:** Ensure the arrays have the same number of dimensions or expandable dimensions.
2. **Dimension Padding:** If arrays have different numbers of dimensions the smaller array is left-padded with ones.
3. **Shape Compatibility:** Two dimensions are compatible if they are equal or one of them is

**NOTE:** If these conditions aren’t met NumPy will raise a Value Error.

**Example 1: Broadcasting a 1D Array to a 2D Array**

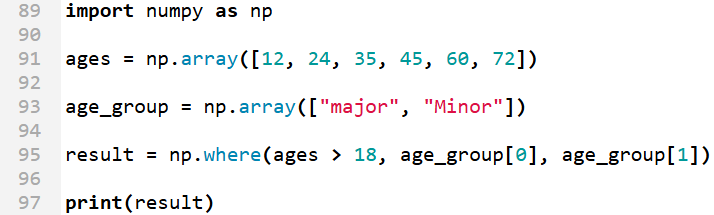
This example shows how a 1D array a1 is added to a 2D array a2. NumPy automatically expands the 1D array along the rows of the 2D array to perform element-wise addition.

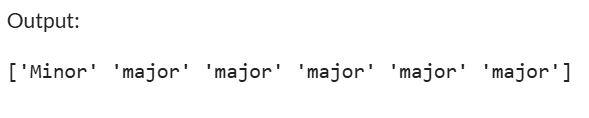




**Example 2: Broadcasting in Conditional Operations**

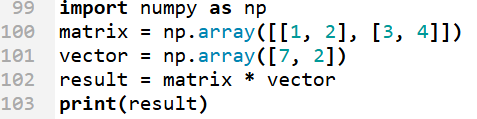
We may need to apply a condition to an entire array or a subset of it. Broadcasting can help to perform these operations efficiently without needing loops.

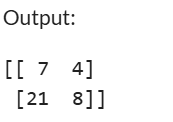




**Example 3: Using Broadcasting for Matrix Multiplication**

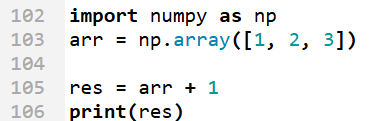
In this example, each element of a 2D matrix is multiplied by the corresponding element in a broadcasted vector.

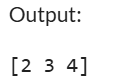




**Example 4: Broadcasting a Scalar to a 1D Array**

It creates a NumPy array arr with values [1, 2, 3]. It adds a scalar value 1 to each element of the array using broadcasting.

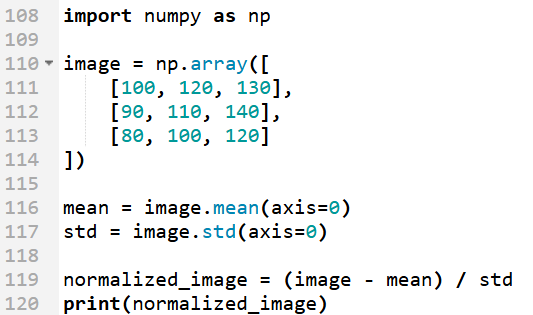


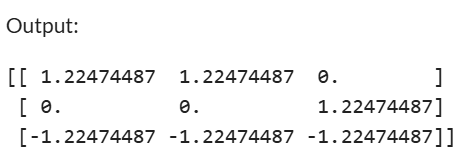


**Example 5: Normalizing Image Data**

Normalization is important in many real-world scenarios like image processing and machine learning because it:

1. Centers data by subtracting the mean by ensuring features have zero mean.
2. Scales data by dividing by the standard deviation by ensuring features have unit variance.
3. Improves numerical stability and performance of algorithms like gradient descent.





**NumPy ufuncs | Universal functions**

**NumPy Universal functions (ufuncs**in short**)** are simple mathematical functions that operate on **ndarray** (**N-dimensional array**) in an element-wise fashion.

It supports **array broadcasting, type casting**, and several other standard features. NumPy provides various universal functions like **standard trigonometric functions**, **functions for arithmetic operations**, handling complex numbers, statistical functions, etc.

**Characteristics of NumPy ufuncs**

* These functions operate on **ndarray** (N-dimensional array) i.e. NumPy's array class.
* It performs fast element-wise array operations.
* It supports various features like array broadcasting, type casting, etc.
* Numpy universal functions are objects that belong to **numpy.ufunc**class.
* Python functions can also be created as a universal function using the **frompyfunc library function**.
* Some **ufuncs** are called automatically when the corresponding arithmetic operator is used on arrays. For example, when the addition of two arrays is performed element-wise using the '+' operator then np.add() is called internally.

**Why use ufuncs?**

ufunc, or universal functions offer various advantages in NumPy. Some benefits of using ufuncs are:

**1. Vectorized Operations**

* ufuncs are applied element-wise to all the elements in the ndarray.
* ufuncs are more efficient than loops as they are applied simultaneously to all elements. Vectorization is very useful on large data sets.

**2. Type Casting**

* Type casting means converting the data type of a variable to perform the necessary operation.
* ufuncs automatically handle type casting and ensure compatible datatypes for calculations.
* This allows code to be concise and reduces the chances of error.

**3. Broadcasting**

* Broadcasting means to perform arithmetic operations on arrays of different size.
* ufuncs automatically handle broadcasting and avoids the need for manual array shape manipulation.

**Basic Universal Functions (ufunc) in NumPy**

Here are some of the universal functions (ufunc) in the NumPy Python library:

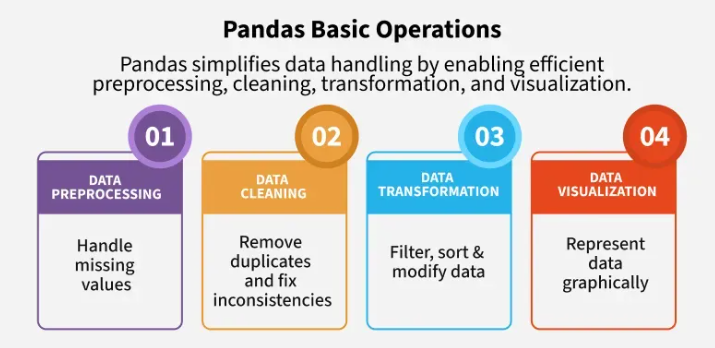
**Trigonometric functions**

These functions work on **radians**, so angles need to be converted to radians by multiplying by **pi/180**. Only then we can call trigonometric functions. They take an array as input arguments.

**PANDAS IN PYTHON:**

Pandas (stands for Python Data Analysis) is an open-source software library designed for **data manipulation** and **analysis**.

* Revolves around two primary Data structures: Series (1D) and DataFrame (2D)
* Built on top of NumPy, efficiently manages large datasets, offering tools for data cleaning, transformation, and analysis.
* Tools for working with time series data, including date range generation and frequency conversion. For example, we can convert date or time columns into pandas’ datetime type using pd.to\_datetime(), or specify parse\_dates=True during CSV loading.
* Seamlessly integrates with other Python libraries like NumPy, Matplotlib, and scikit-learn.
* Provides methods like .dropna() and .fillna() to handle missing values seamlessly



**NOTE:**

* **DataFrames:** It is a two-dimensional data structure constructed with rows and columns, which is more similar to Excel spreadsheet.
* **pandas:** This name is derived for the term "panel data" which is econometrics terms of data sets.

**What is Pandas Used for?**

With pandas, you can perform a wide range of data operations, including

* Reading and writing data from various file formats like CSV, Excel and SQL databases.
* Cleaning and preparing data by handling missing values and filtering entries.
* Merging and joining multiple datasets seamlessly.
* Reshaping data through pivoting and stacking operations.
* Conducting statistical analysis and generating descriptive statistics.
* Visualizing data with integrated plotting capabilities.

**Why Learn Pandas?**

Here’s why it’s worth learning:

* It offers a simple and intuitive way to work with structured data, especially using DataFrames.
* Makes data exploration easy, so you can quickly understand patterns or spot issues.
* Saves time by reducing the need for complex code.
* It's widely used in industries like finance, healthcare, marketing and research.
* A must-have skill for data science, analytics and machine learning roles.

Pandas is open-source Python library which is used for data manipulation and analysis. It consist of data structures and functions to perform efficient operations on data. It is well-suited for working with **tabular data** such as **spreadsheets** or **SQL tables**. It is used in data science because it works well with other important libraries. **It is built on top of the NumPy library**as it makes easier to manipulate and analyze. Pandas is used in other libraries such as:

* **Matplotlib** for plotting graphs
* **SciPy** for statistical analysis
* **Scikit-learn** for machine learning algorithms.
* It uses many functionalities provided by **NumPy library**.

Here is a various tasks that we can do using Pandas:

* **Data Cleaning, Merging and Joining**: Clean and combine data from multiple sources, handling inconsistencies and duplicates.
* **Handling Missing Data**: Manage missing values (NaN) in both floating and non-floating point data.
* **Column Insertion and Deletion**: Easily add, remove or modify columns in a DataFrame.
* **Group By Operations**: Use "split-apply-combine" to group and analyze data.
* **Data Visualization**: Create visualizations with Matplotlib and Seaborn, integrated with Pandas.

**PANDAS DATAFRAME**

A DataFrame is a two-dimensional, size-mutable and potentially heterogeneous tabular data structure with labeled axes (rows and columns).

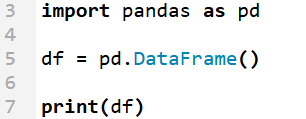
**CREATING A PANDAS DATAFRAME**

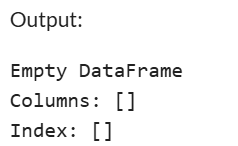
Pandas DataFrame comes is a powerful tool that allows us to store and manipulate data in a structured way, similar to an Excel spreadsheet or a SQL table. A DataFrame is similar to a table with rows and columns. It helps in handling large amounts of data, performing calculations, filtering information with ease.

**Creating an Empty DataFrame**

An empty DataFrame in pandas is a table with no data but can have defined column names and indexes. It is useful for setting up a structure before adding data dynamically. An empty DataFrame can be created just by calling a dataframe constructor.

**EXAMPLE:**

****

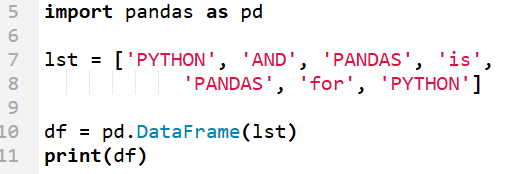
****

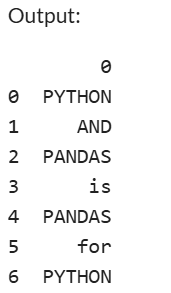
**Creating a DataFrame from a List**

A simple way to create a DataFrame is by using a single list. Pandas automatically assigns index values to the rows when you pass a list.

* Each item in the list becomes a row.
* The DataFrame consists of a single unnamed column.

**EXAMPLE:**

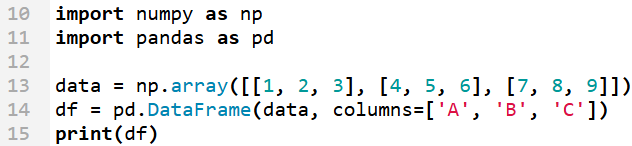
****

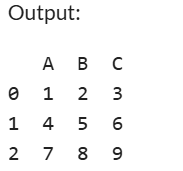
****

**Creating DataFrame from dict of Numpy Array**

We can create a Pandas DataFrame using a dictionary of NumPy arrays. Each key in the dictionary represents a column name and the corresponding NumPy array provides the values for that column.

**EXAMPLE:**

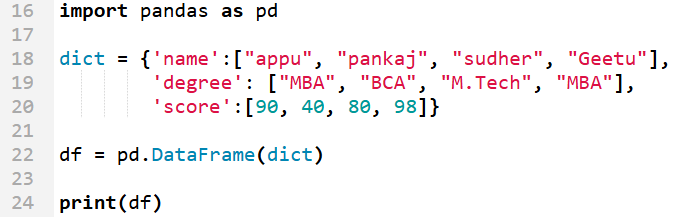
****

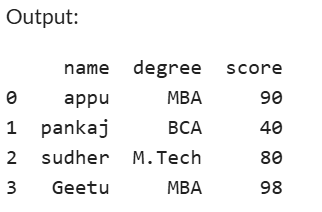
****

**Creating a DataFrame from a List of Dictionaries**

We can also create dataframe using List of Dictionaries. It represents data where each dictionary corresponds to a row. This method is useful for handling structured data from APIs or JSON files. It is commonly used in web scraping and API data processing since JSON responses often contain lists of dictionaries.

**EXAMPLE:**

****

****

**Pandas Series:**

A Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floating-point numbers, Python objects, etc.). It’s similar to a column in a spreadsheet or a database table.

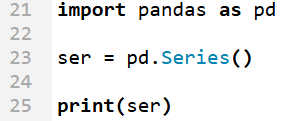
**CREATING A PANDA SERIES:**

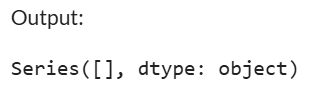
A Pandas Series is like a single column of data in a spreadsheet. It is a one-dimensional array that can hold many types of data such as numbers, words or even other Python objects. Each value in a Series is associated with an index, which makes data retrieval and manipulation easy. This article explores multiple ways to create a Pandas Series with step-by-step explanations and examples.

**Creating an Empty Pandas Series**

An empty Series contains no data and can be useful when we plan to add values later. we can create an empty Series using the pd.Series() function. By default an empty Series has a float64 data type. If we need a different data type specify it using the dtype parameter

**EXAMPLE:**

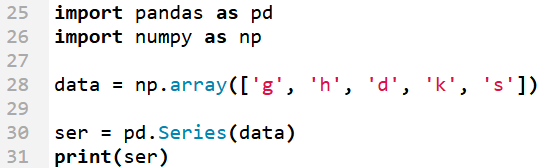
****

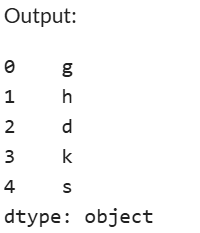
****

**Creating a Series from a NumPy Array**

If we already have data stored in a **NumPy array**we can easily convert it into a Pandas Series. This is helpful when working with numerical data.

**EXAMPLE:**

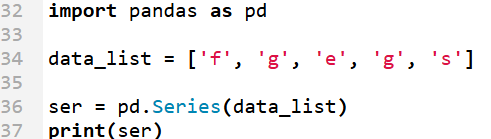
****

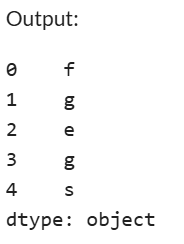
****

**Creating a Series from a List**

we can create a Series by passing a Python **list** to the pd.Series() function. Pandas automatically assigns an index to each element starting from 0. This is a simple way to store and manipulate data.

**EXAPMLE:**

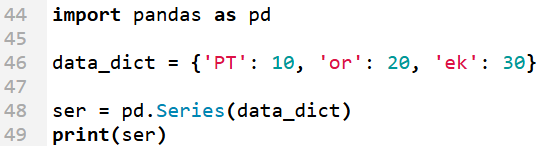
****

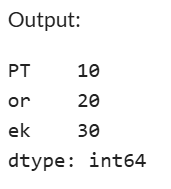
****

**Creating a Series from a Dictionary**

A dictionary in Python stores data as key-value pairs. When we convert Dictionary into a Pandas Series the keys become index labels and the values become the data. This method is useful for labeled data preserving structure and enabling quick access. Below is an example.

**EXAMPLE:**

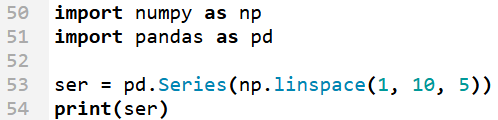
****

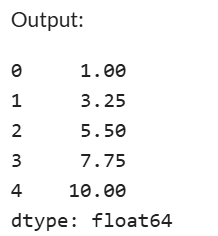
****

**Creating a Series Using NumPy Functions**

In order to create a series using numpy function. Some commonly used NumPy functions for generating sequences include **numpy.linspace()** for creating evenly spaced numbers over a specified range and **numpy.random.randn()** for generating random numbers from a normal distribution. This is particularly useful when working with scientific computations, statistical modeling or large datasets

**EXAMPLE:**

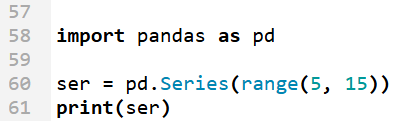
****

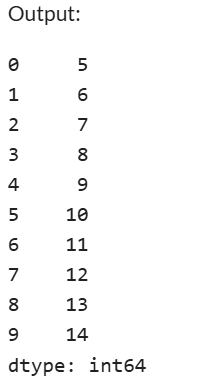
****

**Creating a Series Using range()**

The **range**() function in Python is commonly used to generate sequences of numbers and it can be easily converted into a Pandas Series. This is particularly useful for creating a sequence of values in a structured format without need of manually specify each element. Below is an how range() can be used to create a Series.

**EXAMPLE:**

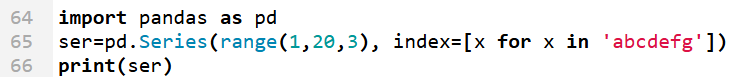
****

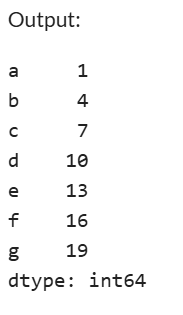
****

**Creating a Series Using List Comprehension**

**List** **comprehension** is a concise way to generate sequences and apply transformations in a single line of code. This method is useful when we need to create structured sequences dynamically. Below is an example demonstrating how list comprehension is used to create a Series with a custom index.

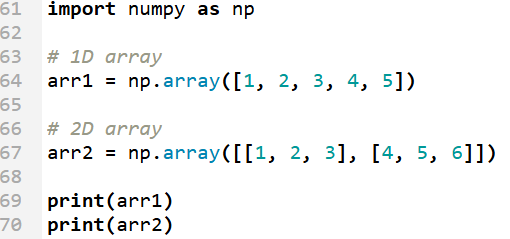
**EXAMPLE:**

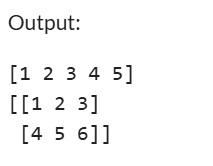
****

****

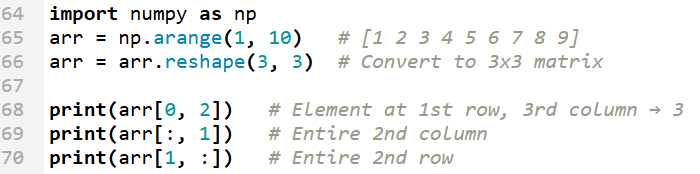
**NUMPY OPERATIONS:**

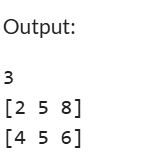
1. **CREATE ARRAYS**

****

****

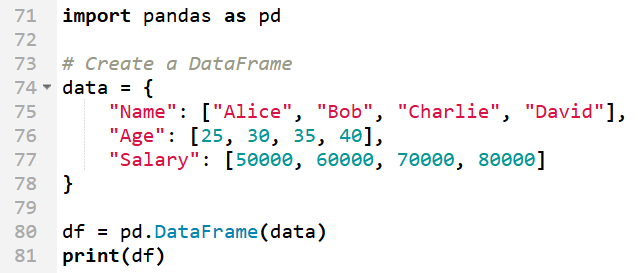
1. **RESHAPING AND INDEXING**

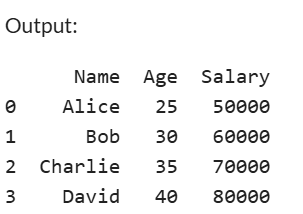
****

****

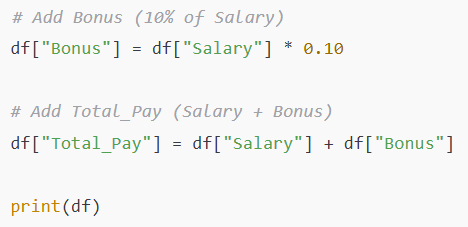
**PANDAS DATASET MANIPULATION:**

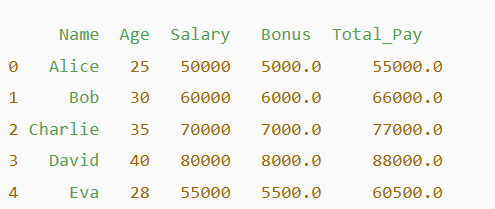
1. **CREATING A DATAFRAME**



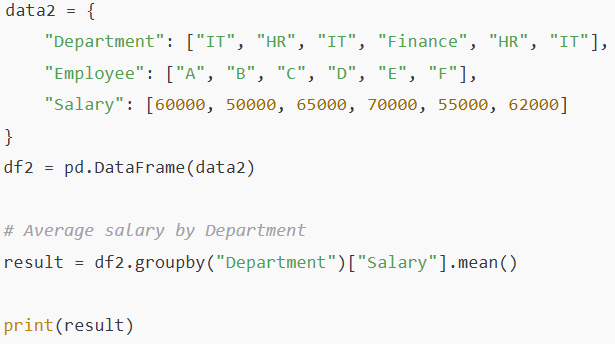


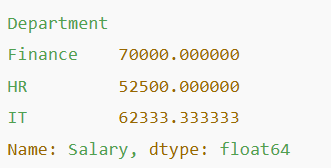
1. **ADDING NEW COLUMNS**



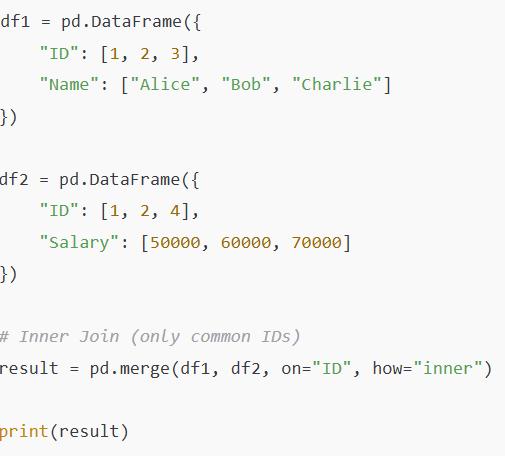


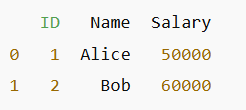
1. **GROUPING AND AGGREGATION**



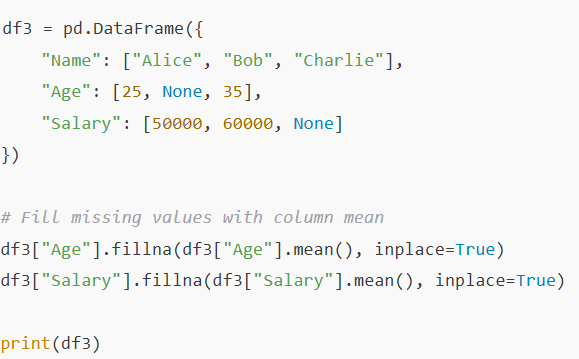


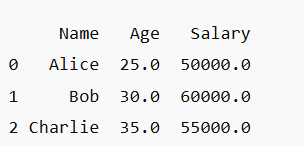
1. **MERGING TWO DATA FRAMES**





1. **HANDLING MISSING DATA**

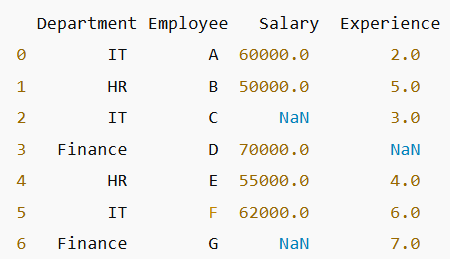




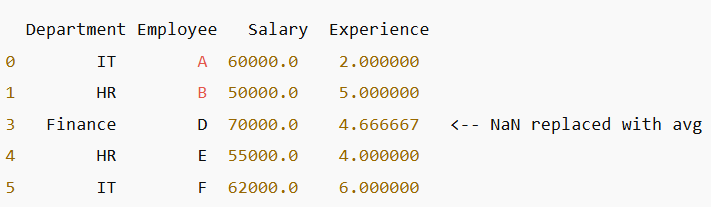
**PROJECT:**

****

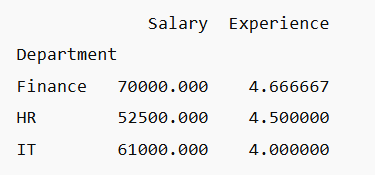
**Original data frame**

****

**Cleaned data frame**

****

**Aggregated data**

****