# Numpy and Pandas in Python

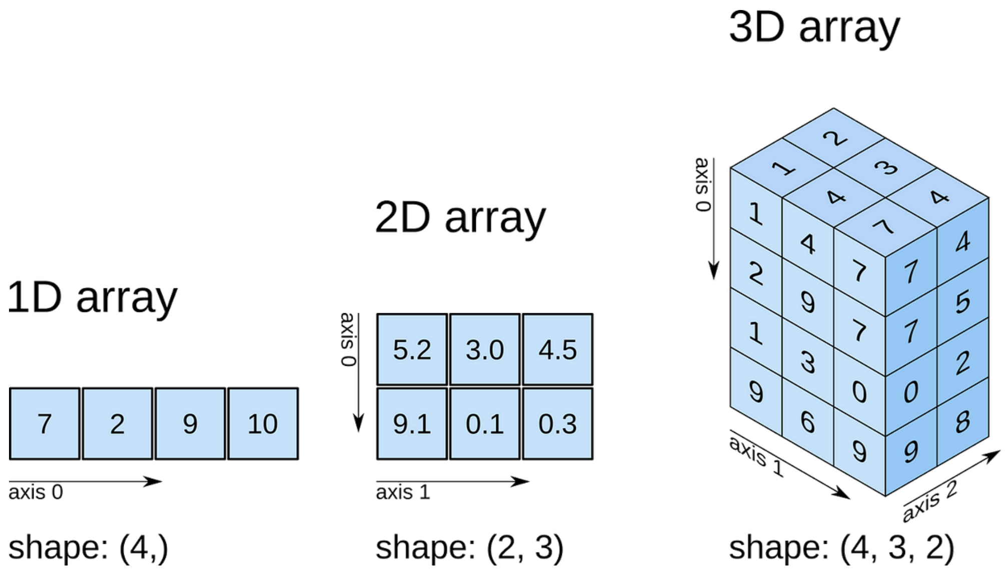
## What is numpy?

- NumPy, short for Numerical Python, is a fundamental library for scientific computing in Python. It acts as the backbone for many other data science and machine learning libraries.
- NumPy's core strength lies in its powerful multidimensional arrays (ndarrays), which can efficiently store and manipulate large sets of numerical data. It offers significant advantages over standard Python lists for numerical computations due to its optimized C-based implementation.
- NumPy provides a rich collection of mathematical functions that operate on entire arrays element-wise, significantly speeding up calculations compared to looping through individual elements.
- This makes NumPy a go-to tool for tasks like linear algebra operations, vectorized calculations, and array transformations, forming the foundation for various scientific and engineering applications in Python.

## Creating Numpy Arrays

**Use the np.array() function to convert a Python NumPy ndarray. The data type of the elements ndarray will be inferred from the data types in default. You can explicitly specify the data type dtype argument.**

```
# Example: Create an array of integers from a

data = [1, 2, 3, 4, 5]

array_int = np.array(data)  # Inferred data type: int

print(array_int.dtype)  # Output: int64
```



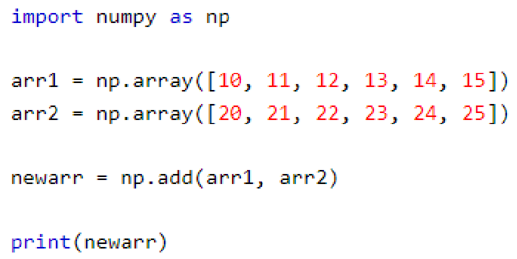
## Iterating Arrays

- Iterating over arrays is a fundamental task in data analysis and scientific computing.
- NumPy provides several methods for iterating over arrays, including for loops, enumerate, and vectorized operations.
- The choice of method depends on the size of the array, the complexity of the operation, and whether you need to access the elements by index.

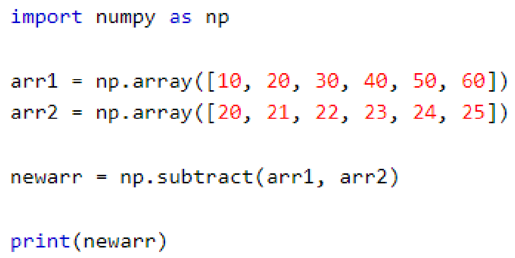
```
# Create a sample array

arr = np.array([1, 2, 3, 4, 5])

# Iterate over the elements using a for loop

for element in arr: print(element)
```

# Numpy for Numerical Operations

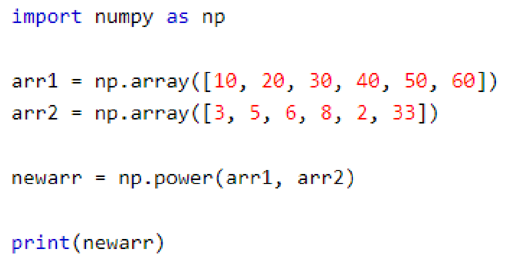
- Numpy can be used for mathematical operations and other complex computations
- Mathematical functions (np.add, np.subtract, np.sin, etc.)
- Broadcasting for compatible shapes
- Universal functions (ufuncs) for efficient computations



```
import numpy as np

arr1 = np.array([10, 11, 12, 13, 14, 15])
arr2 = np.array([20, 21, 22, 23, 24, 25])

newarr = np.add(arr1, arr2)

print(newarr)
```



```
import numpy as np

arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([20, 21, 22, 23, 24, 25])

newarr = np.subtract(arr1, arr2)

print(newarr)
```



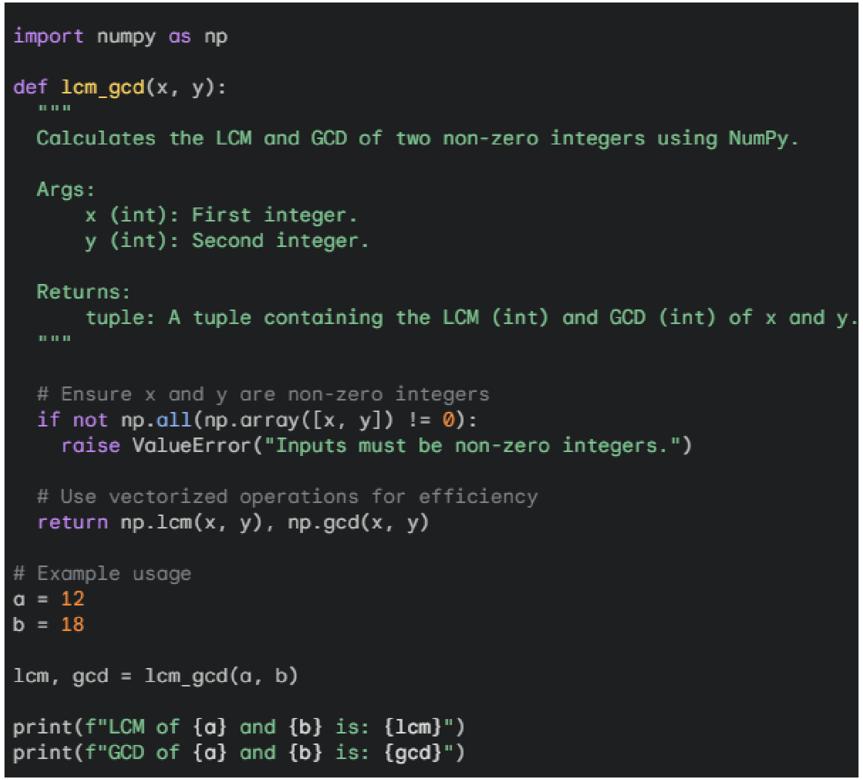
```
import numpy as np

arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([3, 5, 6, 8, 2, 33])

newarr = np.power(arr1, arr2)

print(newarr)
```

## Universal Function (ufunc)

- NumPy ufuncs (universal functions) are highly optimized functions that operate on NumPy arrays.
- Allow element-wise operations on arrays.
- Support various mathematical operations, logical operations,
- Efficient: Optimized for NumPy arrays, leading to faster execution compared to for loops.
- Concise: Offer a concise way to perform element-wise operations.
- Versatility: Support various mathematical, logical, and other operations.

```
import numpy as np
# Create sample arrays
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
# Add the elements using the add ufunc
added_arr = np.add(arr1, arr2)
# Print the added array
print(added_arr)
```

# LCM and GCD in python

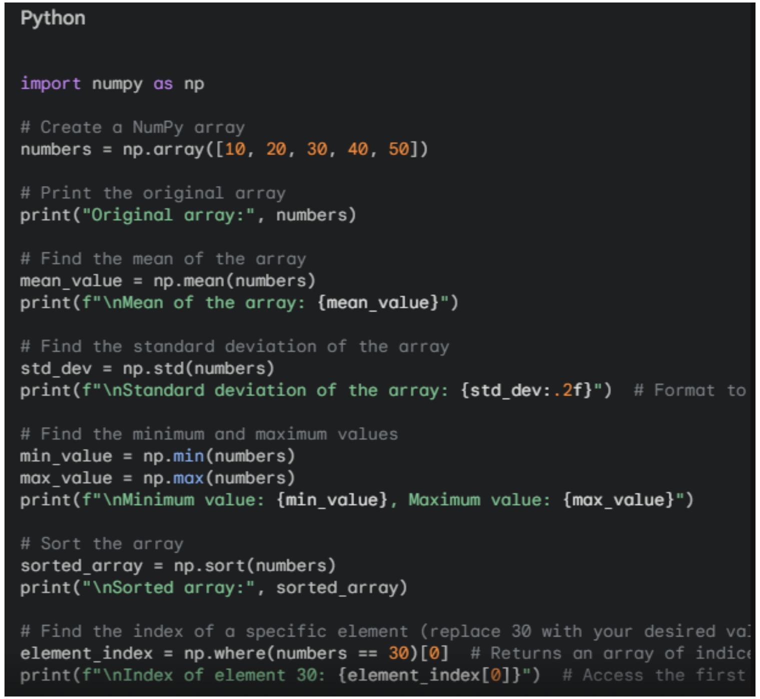


```python
import numpy as np

def lcm_gcd(x, y):
  """
  Calculates the LCM and GCD of two non-zero integers using NumPy.

  Args:
      x (int): First integer.
      y (int): Second integer.

  Returns:
      tuple: A tuple containing the LCM (int) and GCD (int) of x and y.
  """

  # Ensure x and y are non-zero integers
  if not np.all(np.array([x, y]) != 0):
    raise ValueError("Inputs must be non-zero integers.")

  # Use vectorized operations for efficiency
  return np.lcm(x, y), np.gcd(x, y)

# Example usage
a = 12
b = 18

lcm, gcd = lcm_gcd(a, b)

print(f"LCM of {a} and {b} is: {lcm}")
print(f"GCD of {a} and {b} is: {gcd}")
```

# Sample Program

Create a numpy program to:

1. Create a numpy array
2. Find the Mean
3. Find the standard deviation
4. Find min and max
5. Sort the array



Python

```python
import numpy as np

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

# Print the original array
print("Original array:", numbers)

# Find the mean of the array
mean_value = np.mean(numbers)
print(f"\nMean of the array: {mean_value}")

# Find the standard deviation of the array
std_dev = np.std(numbers)
print(f"\nStandard deviation of the array: {std_dev:.2f}")  # Format to

# Find the minimum and maximum values
min_value = np.min(numbers)
max_value = np.max(numbers)
print(f"\nMinimum value: {min_value}, Maximum value: {max_value}")

# Sort the array
sorted_array = np.sort(numbers)
print("\nSorted array:", sorted_array)

# Find the index of a specific element (replace 30 with your desired va
element_index = np.where(numbers == 30)[0]  # Returns an array of indic
print(f"\nIndex of element 30: {element_index[0]}")  # Access the first
```

# Introduction to Pandas

- Pandas is an incredibly useful library in Python for data scientists and analysts. It excels at working with tabular data, similar to spreadsheets but with much more power and flexibility.
- Pandas allows you to efficiently import and export data from various sources, clean and manipulate messy datasets, and perform exploratory analysis.
- It provides powerful tools for wrangling data, including selecting and filtering specific parts, transforming data types, handling missing values, and merging or joining multiple datasets. With its DataFrames, you can organize and structure your data for easy analysis and visualization, making it a cornerstone for data exploration and preparation before diving into machine learning or statistical modeling.

## Using Pandas

**Lists:** Similar to NumPy arrays, you can create a Series from a Python list. Pandas will attempt to infer data types for each element and use them for the Series.

Ex: import pandas as pd data = [1, 2, 3, 4, 5]

my_series = pd.Series(data)

print(my_series)

**Dictionaries**: Use a dictionary as input to the Series constructor. The keys will become the index labels for the Series, and the values will be the data.

Ex: data = {"apple": 10, "banana": 15, "orange": 20}

fruits_series = pd.Series(data)

print(fruits_series)

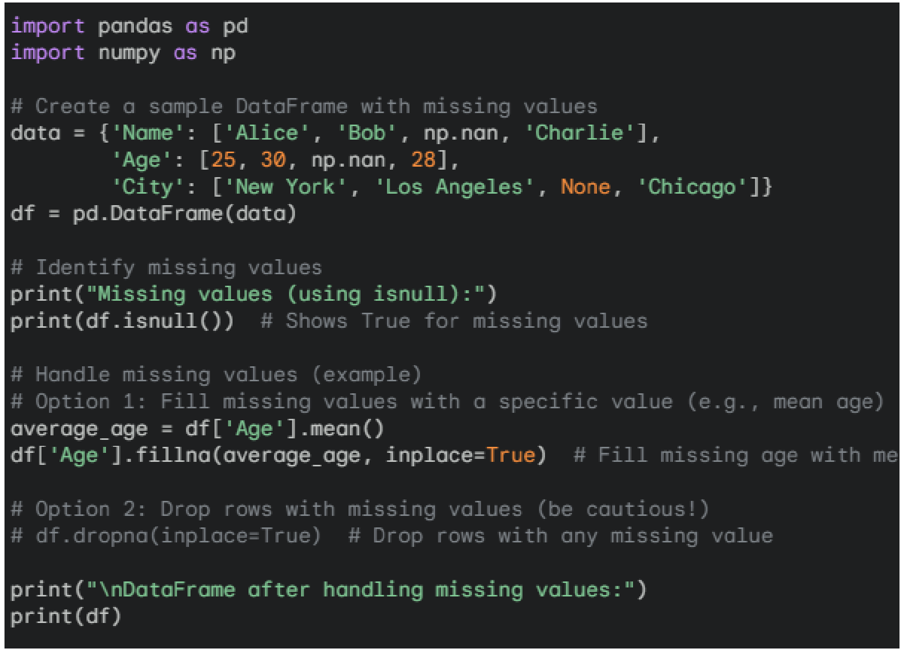
## Dataframes in Pandas

This example creates a list of dictionaries, where each dictionary represents a row in the DataFrame. The keys of the dictionaries become the column names, and the values become the corresponding cell values.

```
import pandas as pd

# Define a list of dictionaries

data = [

    {"Name": "Alice", "Age": 30, "City": "New York"},

    {"Name": "Bob", "Age": 25, "City": "Los Angeles"},

    {"Name": "Charlie", "Age": 32, "City": "Chicago"},

]

# Create a DataFrame from the list

df = pd.DataFrame(data)

# Print the DataFrame

print(df)
```

# Handling Missing values using Pandas

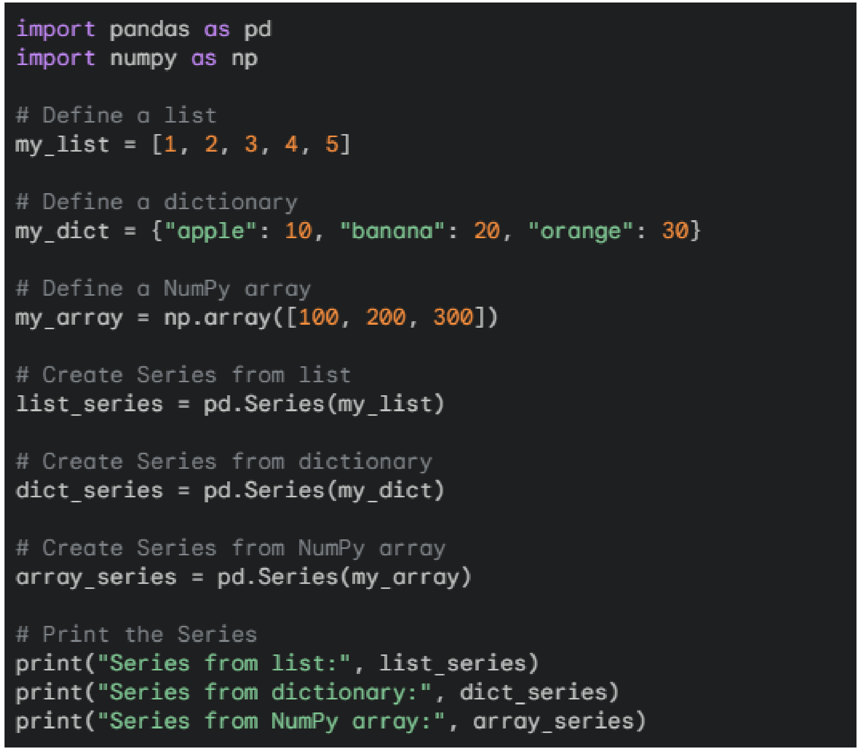
- Identifying missing data (using isnull and notnull)
- Methods for handling missing data:
    - Dropping rows/columns with missing values (use with caution)
    - Filling missing values with specific values (e.g., fillna(0))
    - Interpolation (filling with estimated values based on surrounding data)



```
import pandas as pd
import numpy as np

# Create a sample DataFrame with missing values
data = {'Name': ['Alice', 'Bob', np.nan, 'Charlie'],
        'Age': [25, 30, np.nan, 28],
        'City': ['New York', 'Los Angeles', None, 'Chicago']}
df = pd.DataFrame(data)

# Identify missing values
print("Missing values (using isnull):")
print(df.isnull())  # Shows True for missing values

# Handle missing values (example)
# Option 1: Fill missing values with a specific value (e.g., mean age)
average_age = df['Age'].mean()
df['Age'].fillna(average_age, inplace=True)  # Fill missing age with me

# Option 2: Drop rows with missing values (be cautious!)
# df.dropna(inplace=True)  # Drop rows with any missing value

print("\nDataFrame after handling missing values:")
print(df)
```

## Sample Problem

Develop a code to create a Series from: List, Dictionary and Numpy Array



```python
import pandas as pd
import numpy as np

# Define a list
my_list = [1, 2, 3, 4, 5]

# Define a dictionary
my_dict = {"apple": 10, "banana": 20, "orange": 30}

# Define a NumPy array
my_array = np.array([100, 200, 300])

# Create Series from list
list_series = pd.Series(my_list)

# Create Series from dictionary
dict_series = pd.Series(my_dict)

# Create Series from NumPy array
array_series = pd.Series(my_array)

# Print the Series
print("Series from list:", list_series)
print("Series from dictionary:", dict_series)
print("Series from NumPy array:", array_series)
```