Name : Pawan Acharya

Email: ipawanacharya@gmail.com

Mobile: 9845634867

What is NumPy, and why is it important in the Python ecosystem? Provide a detailed explanation.

NumPy, which stands for Numerical Python, is an open-source library in Python that provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.

NumPy stands out for several reasons:

- **Speed**: Utilizes C code under the hood for quick computation on large data sets.
- **Foundation for Scientific Computing**: Serves as the backbone for many other data science and machine learning libraries like SciPy, Pandas, and Scikit-learn.
- **Mathematical Functions**: Offers a wide array of mathematical operations for complex numerical computations.
- **Memory Efficiency**: Uses less memory for data storage compared to standard Python lists, crucial for handling large data.
- **Broadcasting and Vectorization**: Facilitates cleaner and faster code by avoiding explicit loops in arithmetic operations and function applications.

Discuss the history and development of NumPy. How has it evolved over time?

NumPy, crucial for numerical computing in Python, originated from the need to unify Python's numerical array capabilities. It evolved from two earlier projects: Numeric, created by Jim Hugunin in 1995, and Numarray, developed later for handling larger arrays and offering more flexibility. Travis Oliphant merged the best features of both into NumPy in 2005, with NumPy 1.0 released in 2006. Since then, it has continuously improved in functionality and performance, thanks to contributions from a vibrant community. NumPy has become foundational for scientific computing in Python, underpinning many other libraries and adapting over time to include support for new computing environments and enhancing its core capabilities for a wide range of applications.

Describe the core features of NumPy. How do these features benefit scientific and mathematical computing efforts?

NumPy is a cornerstone of Python's scientific computing stack, offering:

1. **Multidimensional Arrays**: Efficient handling of large, multi-dimensional data sets.
2. **Broadcasting**: Simplifies arithmetic operations on arrays of different shapes.

3. **Mathematical Functions**: Comprehensive suite for linear algebra, statistics, and Fourier transforms.
4. **Linear Algebra Support**: Essential tools for complex mathematical computations.
5. **C and Fortran Integration**: Facilitates the use of legacy code for performance-critical tasks.
6. **Memory Efficiency**: More compact than Python lists, enabling savings on large datasets.
7. **Speed**: Executes operations much faster than pure Python, crucial for large-scale data processing.
8. **Random Number Generation**: Supports simulations and statistical modeling.

These features make NumPy indispensable for efficiently performing complex numerical computations, essential for scientific research, data analysis, and engineering applications.

Explain the concept of ndarrays. How do ndarrays differ from standard Python lists?

ndarrays are NumPy's multidimensional arrays that provide efficient storage and manipulation of homogeneous data. They differ from Python lists by requiring all elements to be of the same data type, enabling faster processing and less memory usage. Unlike lists, **ndarrays** support vectorized operations and broadcasting, allowing for mathematical operations on whole arrays without explicit loops, making them better suited for numerical computations.

What are universal functions (ufuncs) in NumPy? Give examples and explain their significance.

Universal functions (ufuncs) in NumPy are functions that operate element-wise on arrays, providing a highly efficient and fast way to perform mathematical operations. Ufuncs are implemented in C, ensuring performance is optimized, which is critical for numerical computations involving large data arrays.

Examples of ufuncs:

- **Arithmetic Operations**: **np.add**, **np.subtract**, **np.multiply**, **np.divide** for basic element-wise addition, subtraction, multiplication, and division.
- **Trigonometric Functions**: **np.sin**, **np.cos**, **np.tan** for computing the sine, cosine, and tangent of each element.
- **Exponential and Logarithmic Functions**: **np.exp**, **np.log**, **np.log10** for exponential and logarithmic operations.
- **Aggregation Functions**: **np.sum**, **np.min**, **np.max** for summing up elements, finding minimum or maximum values.

Describe various mathematical operations that can be performed using NumPy. Provide examples for each.

NumPy supports a wide range of mathematical operations, making it highly versatile for scientific computing. Here's an overview of various operations and examples for each:

1. Basic Arithmetic Operations

Perform element-wise operations like addition, subtraction, multiplication, and division.

- **Addition**: **np.add(arr1, arr2)** or **arr1 + arr2**
- **Subtraction**: **np.subtract(arr1, arr2)** or **arr1 - arr2**
- **Multiplication**: **np.multiply(arr1, arr2)** or **arr1 * arr2**
- **Division**: **np.divide(arr1, arr2)** or **arr1 / arr2**

2. Trigonometric Functions

Compute trigonometric ratios for array elements.

- **Sine**: **np.sin(arr)**
- **Cosine**: **np.cos(arr)**
- **Tangent**: **np.tan(arr)**

3. Exponential and Logarithmic Functions

Handle exponential and logarithmic calculations.

- **Exponential**: **np.exp(arr)**
- **Natural Logarithm**: **np.log(arr)**
- **Base-10 Logarithm**: **np.log10(arr)**

4. Statistical Operations

Calculate statistical measures directly on arrays.

- **Mean**: **np.mean(arr)**
- **Median**: **np.median(arr)**
- **Standard Deviation**: **np.std(arr)**

5. Linear Algebra

NumPy offers functions for matrix operations and linear algebra.

- **Dot Product**: **np.dot(arr1, arr2)**
- **Matrix Multiplication**: **np.matmul(arr1, arr2)** or **arr1 @ arr2**
- **Determinant**: **np.linalg.det(matrix)**
- **Inverse**: **np.linalg.inv(matrix)**

6. Aggregate Functions

Perform aggregation operations over arrays.

- **Sum**: **np.sum(arr)**
- **Minimum**: **np.min(arr)**
- **Maximum**: **np.max(arr)**

7. Comparison Operations

Compare elements of arrays to generate boolean arrays.

- **Greater Than**: **np.greater(arr1, arr2)** or **arr1 > arr2**
- **Equal To**: **np.equal(arr1, arr2)** or **arr1 == arr2**

Explain the concept of aggregation in NumPy. How does it differ from simple summation or multiplication?

Aggregation in NumPy refers to the process of summarizing or condensing large amounts of data into smaller, summary statistics or values. This includes operations like finding the sum, mean, median, minimum, maximum, standard deviation, etc., of data contained in NumPy arrays. Aggregation functions take a collection of values and return a single value that represents a summary of the collection.

How It Differs from Simple Summation or Multiplication:

- **Summation (np.sum)**: This is a form of aggregation that combines elements into a single sum value. While it is an aggregation operation, it's specifically focused on adding all elements.
- **Multiplication (np.prod)**: This multiplies all elements of the array together, another specific form of aggregation.

Aggregation differs from these operations in that it encompasses a wider range of functions designed to provide insights into the distribution, shape, and central tendency of data, beyond just adding or multiplying elements. For example:

- **Mean (np.mean)** provides the average value of the data, giving an insight into the central tendency.
- **Standard Deviation (np.std)** offers a measure of the spread of the data around the mean, indicating how much variation exists.
- **Minimum (np.min) and Maximum (np.max)** identify the smallest and largest values, highlighting the range of the data.

Aggregations are more about getting a holistic view of the data rather than performing a simple arithmetic operation. They can reveal underlying patterns or characteristics of the data set as a whole, which is invaluable for data analysis, statistics, and scientific research.

Provide examples of different aggregation functions available in NumPy and their practical applications.

Aggregation in NumPy refers to the process of summarizing or condensing large sets of data into smaller, summary statistics or values. Aggregation functions compute a single value from a collection of values, offering insights into the distribution, tendency, or overall characteristics of the data. While simple summation or multiplication operates on two arrays or an array and a scalar value to produce another array of values, aggregation functions reduce the dimensionality of the data, providing a single summary statistic.

Examples of Aggregation Functions and Their Applications:

1. **Sum (np.sum)**: Calculates the total sum of elements in an array. It's useful for finding the total magnitude of a dataset.
    - **Application**: Summing the total number of items sold across all stores.
2. **Mean (np.mean)**: Computes the average value of an array. It's helpful for identifying the central tendency of the data.
    - **Application**: Calculating the average temperature of a region over a month.
3. **Median (np.median)**: Finds the middle value in an array. It's used when you need a measure of central tendency that's less sensitive to outliers.
    - **Application**: Determining the median income in a survey to understand income distribution.
4. **Standard Deviation (np.std)**: Measures the spread of the data around the mean. It's crucial for assessing the variability in a dataset.
    - **Application**: Evaluating the consistency of an athlete's performance over time.
5. **Variance (np.var)**: Calculates the square of the standard deviation. It provides insight into the degree of spread in the data.
    - **Application**: Analyzing the variability in electricity consumption in households.
6. **Min (np.min)** and **Max (np.max)**: Identify the minimum and maximum values in an array, respectively. These are useful for understanding the range of the data.
    - **Application**: Finding the highest and lowest temperatures recorded in a city.
7. **Percentile (np.percentile)**: Computes the nth percentile of an array, giving a value below which a certain percentage of observations fall.
    - **Application**: Determining the 90th percentile of test scores to identify the top performers.
8. **Aggregate (np.aggregate)**: Allows for custom aggregation by applying a specified function to an array.
    - **Application**: Custom aggregations tailored to specific analytical needs, such as weighted averages or custom statistical models.

Importance of NumPy in Python

NumPy is critically important in the Python ecosystem for several key reasons, making it indispensable for data science, scientific computing, and more:

1. **Efficient Data Handling**: NumPy introduces powerful N-dimensional array objects which allow for efficient storage and manipulation of large datasets. These arrays are faster and more space-efficient than Python lists.
2. **Foundation for Scientific Computing**: It serves as the foundational library for a vast majority of the Python scientific stack, including libraries like SciPy (for advanced scientific computing), Pandas (for data manipulation and analysis), Matplotlib (for plotting and visualization), and machine learning libraries like TensorFlow and Scikit-learn.
3. **Performance Boost**: Written in C and Python, NumPy operations are executed much faster than standard Python code, particularly for complex mathematical operations and large data manipulation. This performance boost is crucial for research and applications requiring intensive computations.
4. **Comprehensive Mathematical Functions**: NumPy provides a wide array of mathematical functions including linear algebra routines, Fourier transforms, and statistics, making it a versatile tool for mathematical modeling and computation.
5. **Cross-discipline Utility**: Beyond data science, NumPy is used in a variety of fields that require numerical computing, such as finance, engineering, bioinformatics, and physics, demonstrating its versatility and broad applicability.
6. **Data Interoperability**: NumPy arrays can serve as a bridge between Python and libraries written in other languages like C, C++, and Fortran, allowing for efficient data exchange and integration of legacy code into Python applications.
7. **Community and Ecosystem**: With a large, active community, NumPy is continuously improved and updated. Its widespread adoption and extensive documentation make it accessible to newcomers, while its depth of functionality caters to the needs of advanced users.

Efficiency and Performance:
Discuss the importance of NumPy in the context of efficiency and performance in scientific computing.

NumPy is vital for efficiency and performance in scientific computing due to its optimized, low-level C implementations for mathematical operations. This allows NumPy to handle large data arrays much faster than Python's built-in types. By providing efficient storage and computation for large datasets, NumPy minimizes memory usage and speeds up processing times, making it indispensable for high-performance computing tasks in research, data analysis, and engineering.

How does NumPy improve computational performance in Python? Provide a detailed explanation with examples.

NumPy enhances computational performance in Python through several key mechanisms, making it highly efficient for numerical computations, data analysis, and scientific research. Here's a detailed explanation of these mechanisms along with examples:

1. Homogeneous Data Types and Contiguous Memory Allocation

NumPy arrays store elements of the same data type, enabling efficient data access and manipulation. This homogeneity allows NumPy to allocate memory contiguously, improving cache utilization and computational speed.

Example:

```
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='int32')
```

2. Vectorization

Vectorization refers to performing operations on entire arrays rather than their individual elements, eliminating the overhead of Python loops. This approach utilizes low-level optimizations and SIMD (Single Instruction, Multiple Data) capabilities of modern processors, significantly speeding up computations.

Example:

```
a = np.array([1, 2, 3])

b = np.array([4, 5, 6])

c = a + b  # Vectorized addition
```

3. Broadcasting

NumPy can perform arithmetic operations on arrays of different shapes and sizes by "broadcasting" smaller arrays over the larger ones. This feature enables more flexible code without the need for manually matching array sizes.

Example:

```
a = np.array([1, 2, 3])

b = 2

c = a + b  # Broadcasting adds 'b' to every element in 'a'
```

4. Efficient Memory Usage

NumPy's efficient handling of memory, such as reducing memory footprint and minimizing memory bandwidth usage, contributes to its high performance. It uses contiguous memory blocks for data storage, allowing for efficient caching and memory access patterns.

Example:

```
import sys
```

```
lst = list(range(1000))
print("Python list memory:", sys.getsizeof(lst), "bytes")


np_arr = np.arange(1000)
print("NumPy array memory:", np_arr.nbytes, "bytes")
```

5. Universal Functions (ufuncs)

NumPy provides universal functions (ufuncs), which are optimized, compiled functions that operate element-wise on arrays. Ufuncs are much faster than Python functions, especially for large datasets.

Example:

```
angles = np.array([0, np.pi/2, np.pi])
sin_values = np.sin(angles)  # Efficient element-wise sine operation
```

6. Compiled Code

At its core, NumPy is implemented in C, which means its operations run at compiled speed. This significantly reduces the execution time compared to Python's interpreted execution.

Example: NumPy's dot product vs. a Python loop implementation for matrix multiplication showcases significant speed differences due to compiled C code execution.

Interoperability:

Interoperability

A critical aspect of NumPy's design is its interoperability with other languages and libraries. It provides seamless integration with C, C++, and Fortran code, enabling researchers and developers to leverage existing legacy codebases and specialized libraries without significant overhead. This capability not only extends NumPy's utility but also ensures that performance-critical sections of an application can be written in a lower-level language for maximum efficiency.

In conclusion, NumPy's design—focused on homogeneous data types, contiguous memory allocation, vectorization, compiled C code, and efficient memory usage—coupled with its interoperability features, substantially improves computational performance in Python, particularly for tasks involving large datasets and complex mathematical operations common in scientific computing.

Explain how NumPy interacts with other Python libraries. Why is this interoperability important for data science and machine learning?

NumPy interacts with other Python libraries through its array object, serving as the fundamental data structure for numerical data across the Python data science and machine learning ecosystem. Libraries like Pandas, SciPy, Matplotlib, Scikit-learn, TensorFlow, and PyTorch either build upon NumPy arrays directly or support interoperability with them. This widespread compatibility is crucial for several reasons:

1. **Seamless Data Workflow**: Data can be easily transferred between different libraries without conversion or loss of efficiency, streamlining the data processing pipeline from preprocessing and analysis to visualization and machine learning modeling.
2. **Standardization**: NumPy arrays as a standard data format simplify learning and using different tools within the ecosystem, allowing practitioners to focus on solving problems rather than wrestling with data compatibility issues.
3. **Performance Optimization**: Many libraries leverage NumPy's efficient, C-optimized array operations under the hood, ensuring high performance across the board.
4. **Flexibility and Innovation**: Interoperability encourages the development of specialized libraries that complement each other, fostering innovation within the data science and machine learning fields.

This interoperability forms the backbone of Python's powerful and cohesive ecosystem for data science and machine learning, enabling efficient, flexible, and accessible workflows for practitioners at all levels of expertise.

Comparison: Python List vs NumPy :

When comparing Python lists to NumPy arrays, several key differences stand out, each impacting performance, usability, and functionality in significant ways. Here's a comparison across various dimensions:

1. Data Homogeneity

- **Python List**: Can contain elements of different data types.
- **NumPy Array**: Requires all elements to be of the same data type, ensuring efficient storage and computations.

2. Memory Usage

- **Python List**: Less memory efficient due to the flexibility in storing different data types and additional storage for type information.
- **NumPy Array**: More memory efficient as it directly stores data in a compact, contiguous block of memory.

3. Performance

- **Python List**: Slower operations, especially as the list size grows, because operations are implemented in Python.
- **NumPy Array**: Faster operations due to implementation in C and the use of homogeneous data types, allowing for vectorized operations and efficient caching.

4. Functionality

- **Python List**: Provides basic functionality for data storage and manipulation but lacks built-in support for mathematical operations beyond simple arithmetic.

- **NumPy Array**: Offers extensive mathematical functionality, including operations for linear algebra, statistics, and more. Supports vectorized operations, broadcasting, and more sophisticated array manipulations.

5. Flexibility vs. Rigidity

- **Python List**: More flexible in terms of data types and structures that can be stored, making it suitable for general-purpose programming.
- **NumPy Array**: More rigid in data type requirements but provides significant advantages in terms of computational efficiency and functionality for numerical data.

6. Size and Scalability

- **Python List**: Can become inefficient in terms of both speed and memory usage as the data size grows.
- **NumPy Array**: Designed to handle large data sets efficiently, making it well-suited for data-intensive computations.

Python List Example:

```
# Adding elements of two lists

list1 = [1, 2, 3]

list2 = [4, 5, 6]

sum_list = [a + b for a, b in zip(list1, list2)]
```

NumPy Array Example:

```
import numpy as np

# Element-wise addition of two arrays

array1 = np.array([1, 2, 3])

array2 = np.array([4, 5, 6])

sum_array = array1 + array2
```

Compare the memory consumption between Python lists and NumPy arrays. Why is there a difference?

Python Lists

- **Dynamic:** Python lists are dynamic arrays. They can store elements of different data types (e.g., integers, strings, objects) within the same list.
- **Memory Overhead:** Due to their flexibility and the need to handle elements of various types, each item in a list is a complete Python object that stores not only the value but also additional information, like type and reference count. This overhead makes lists memory-inefficient for numeric data storage.
- **Pointer Structure:** The list itself holds pointers to the objects it contains, rather than the raw data. This further adds to the memory overhead.

NumPy Arrays

- **Homogeneous:** NumPy arrays are designed for numerical calculations. They are homogeneous, meaning all elements are of the same data type. This restriction allows for more efficient data storage.
- **Less Memory Overhead:** Since NumPy knows the type of elements stored, it can allocate a contiguous block of memory ahead of time, significantly reducing the memory overhead per item. There's no need to store type information for each element, as the array's data type is uniform.
- **Direct Storage:** The array directly holds the raw data values, which eliminates the need for pointers to Python objects. This direct approach not only reduces memory usage but also speeds up access and computation.

Why the Difference?

The primary reason for the difference in memory consumption between Python lists and NumPy arrays is their design philosophy and intended use case. Python lists are designed for general-purpose use and flexibility, allowing for mixed-type elements at the cost of higher memory usage and slower performance for numerical operations. NumPy arrays, on the other hand, are specialized for numerical computations, favoring performance and lower memory usage over flexibility. This specialization includes using a contiguous memory block for storage and performing operations in compiled code.

Conduct a performance analysis between Python lists and NumPy arrays for basic operations (addition, multiplication, etc.).

```
import numpy as np

import time


# Size of the array and list

size = 1000000


# Creating a Python list and a NumPy array

python_list = list(range(size))

numpy_array = np.array(python_list)


# Performance for addition

# Python list

start_time = time.time()

python_list_added = [i + 1 for i in python_list]

python_list_add_time = time.time() - start_time
```

```
# NumPy array
start_time = time.time()
numpy_array_added = numpy_array + 1
numpy_array_add_time = time.time() - start_time

# Performance for multiplication
# Python list
start_time = time.time()
python_list_multiplied = [i * 2 for i in python_list]
python_list_multiply_time = time.time() - start_time

# NumPy array
start_time = time.time()
numpy_array_multiplied = numpy_array * 2
numpy_array_multiply_time = time.time() - start_time

python_list_add_time, numpy_array_add_time, python_list_multiply_time,
numpy_array_multiply_time
```

Output:

```
(0.22359609603881836,
 0.013400793075561523,
 0.14237356185913086,
 0.008232593536376953)
```

Why does NumPy outperform Python lists?

NumPy (Numerical Python) outperforms Python lists for several reasons, particularly in the context of numerical computations and data manipulation. These reasons include:

1. **Homogeneous Data Types**: NumPy arrays are homogeneous, meaning they contain elements of the same data type. This contrasts with Python lists, which can contain elements of different data types. The uniformity in NumPy arrays allows for more efficient storage and manipulation of data because the system can optimize operations knowing the exact type and size of each element.
2. **Contiguous Memory Allocation**: NumPy arrays are stored in contiguous blocks of memory, unlike Python lists which are arrays of pointers to objects scattered across

memory. This contiguous storage leads to better cache utilization and memory access patterns, significantly improving performance for numerical computations.

3. **Vectorization**: NumPy utilizes vectorized operations, meaning operations on NumPy arrays can be performed element-wise using highly optimized C and Fortran libraries without the need for explicit loops in Python. This not only makes code more concise but also significantly faster due to the reduction in overhead and the use of optimized low-level code for computations.
4. **Compiled Code**: Under the hood, NumPy operations run on pre-compiled C and Fortran code. This is much faster than Python's interpreted code. The efficiency of compiled languages comes from being closer to the machine language, thus reducing the execution time for operations.
5. **Parallel Processing**: Some operations in NumPy can automatically use multiple CPU cores due to its underlying libraries (like BLAS and LAPACK for linear algebra operations). While Python's global interpreter lock (GIL) limits the execution of multiple threads in a single process, NumPy can bypass these limitations for certain operations, leveraging multi-core processors more effectively.
6. **Specific Optimizations**: NumPy includes specific optimizations for numerical computations, including functions for linear algebra, statistical operations, and random number generation. These are highly optimized for performance, far exceeding what could be efficiently implemented in pure Python.
7. **Less Overhead**: Python lists carry a significant amount of overhead for type checking and other dynamic features. NumPy arrays reduce this overhead by having fixed types and sizes, making operations much quicker.

Discuss the speed difference between operations performed on Python lists and those on NumPy arrays. Provide reasons for any disparities observed.

The speed difference between operations on Python lists and NumPy arrays is mainly due to:

1. **Data Type**: NumPy arrays are homogeneous (same data type) and can be optimized, whereas Python lists are heterogeneous (different data types) and require type checking.
2. **Memory Storage**: NumPy arrays use contiguous memory, improving cache efficiency and speed, while Python lists have scattered memory allocation.
3. **Operations**: NumPy operations are vectorized and run on compiled C and Fortran code, making them much faster than Python's interpreted loops.
4. **Optimizations**: NumPy takes advantage of low-level optimizations and CPU features (like SIMD) that are not available for Python lists.
5. **Overhead**: Python lists carry more overhead due to dynamic type checking and interpreter overhead, whereas NumPy arrays minimize this with fixed types and sizes.

In essence, NumPy arrays are designed and optimized for numerical computations, making them significantly faster for such tasks than Python lists.

Practical Application: Reading and Manipulating CSV Data with NumPy

```
import numpy as np

# Step 1: Reading CSV Data
data = np.genfromtxt('data.csv', delimiter=',', skip_header=1, dtype=float)

# Step 2: Manipulating Data - Calculating the mean of the first column
mean_col1 = np.mean(data[:, 0])
print("Mean of column 1:", mean_col1)

# Step 3: Filtering Data - Selecting rows where the second column is greater than 50
filtered_data = data[data[:, 1] > 50]

# Step 4: Saving Modified Data
np.savetxt('modified_data.csv', filtered_data, delimiter=',', fmt='%f')
```

How can you read a CSV file in Python without using external libraries? Write a script that reads a CSV file using csv.reader and converts it into a list.

```
import csv

# Specify the path to your CSV file
csv_file_path = 'your_file.csv'

# Initialize an empty list to hold the rows of the CSV file
data = []
```

```
# Open the CSV file for reading
with open(csv_file_path, newline='', encoding='utf-8') as csvfile:
    # Create a csv.reader object to read from the opened file
    reader = csv.reader(csvfile)

    # Iterate over each row in the csv.reader object
    for row in reader:
        # Append each row to the data list
        data.append(row)

# At this point, 'data' is a list of lists, where each sublist represents a row in the CSV
# Now, you can print the data or work with it
print(data)
```

Once you have your data in a list format, convert this data into a NumPy array. Why might this conversion be beneficial for further operations?

```
import numpy as np

# Assuming 'data' is your list of lists
# Convert 'data' to a NumPy array
numpy_array = np.array(data)
print(numpy_array)
```

Benefits of Conversion to NumPy Array

Converting your data into a NumPy array is beneficial for several reasons:

1. **Performance**: NumPy arrays are stored in contiguous blocks of memory, making operations on them much faster than operations on lists, especially for large datasets.
2. **Efficient Storage**: Due to their homogeneous nature, NumPy arrays use storage more efficiently than Python lists, which can contain elements of different types and thus require more memory.
3. **Vectorized Operations**: NumPy enables vectorized operations, meaning you can apply functions and operations on entire arrays at once without needing to loop over

elements. This not only results in cleaner, more readable code but also significantly improves performance.

4. **Advanced Mathematical Functions**: NumPy provides a vast library of mathematical functions that can be applied directly to arrays, facilitating complex mathematical operations, statistical analyses, and more.
5. **Multidimensional Data Handling**: While Python lists can be nested to create structures like matrices, NumPy arrays are inherently designed to handle multidimensional data, making operations on such data more straightforward and efficient.
6. **Integration with Other Libraries**: Many Python data science and machine learning libraries, such as SciPy, Matplotlib, and Pandas, are built to work well with NumPy arrays, so converting your data into a NumPy array early on can facilitate interoperability with these tools.

Apply universal functions on the NumPy array derived from your CSV data. Explain what each function does and why it might be useful for data analysis.

```
import numpy as np

# Example CSV data (here as a string for demonstration purposes)
csv_data = """
Day,Temperature
Monday,22.5
Tuesday,24.0
Wednesday,19.5
Thursday,21.0
Friday,23.0
Saturday,25.5
Sunday,20.0
"""

# Simulating reading CSV data into a NumPy array (focusing only on the temperatures)
temperature_data = np.array([22.5, 24.0, 19.5, 21.0, 23.0, 25.5, 20.0])

# Applying various universal functions on the temperature data
# Calculate the mean temperature
mean_temperature = np.mean(temperature_data)

# Calculate the maximum temperature
max_temperature = np.max(temperature_data)

# Standard deviation to understand the variability
std_deviation = np.std(temperature_data)

# Convert temperatures to Fahrenheit
temperatures_fahrenheit = np.multiply(temperature_data, 9/5) + 32

mean_temperature, max_temperature, std_deviation, temperatures_fahrenheit
```

1. **Mean** (**np.mean**): Calculates the average temperature, useful for identifying the central trend.
2. **Maximum** (**np.max**): Finds the highest temperature, helpful for spotting extreme values.
3. **Standard Deviation** (**np.std**): Measures data variability, indicating the spread of temperature values around the mean.
4. **Conversion** (**np.multiply** and addition): Converts temperatures from Celsius to Fahrenheit, demonstrating element-wise operations for unit conversions.

Perform aggregation operations on your dataset. Use aggregation functions and explain the insights they could provide about your dataset.

1. **Average Salary by Department** reveals the compensation landscape, helping to identify which departments are more lucrative and might require more investment.
2. **Median Age and Years at Company by Department** provide a snapshot of workforce demographics and experience, indicating departments with more seasoned employees or newer, potentially more dynamic teams.
3. **Total Salary Expense by Department** shows where the bulk of payroll expenses are allocated, aiding in budget planning and identifying potential for financial adjustments.

4. **Count of Employees by Department** highlights department sizes, which is essential for resource distribution and organizational structure decisions.
5. **Max and Min Salaries within the Company** point out the range of compensation, shedding light on pay equity and informing salary structure discussions.

```
# Generating a sample dataset for the employee context
employee_data = {
    'EmployeeID': range(1, 101),  # 100 employees
    'Department': np.random.choice(['HR', 'Tech', 'Sales', 'Marketing', 'Finance'], 100),
    'Age': np.random.randint(22, 60, 100),
    'YearsAtCompany': np.random.randint(1, 20, 100),
    'Salary': np.random.randint(50000, 150000, 100),
}

employee_df = pd.DataFrame(employee_data)

# 1. Average Salary by Department
avg_salary_by_dept = employee_df.groupby('Department')['Salary'].mean()

# 2. Median Age and Years at Company by Department
median_age_by_dept = employee_df.groupby('Department')['Age'].median()
median_years_at_company_by_dept =
employee_df.groupby('Department')['YearsAtCompany'].median()

# 3. Total Salary Expense by Department
total_salary_expense_by_dept = employee_df.groupby('Department')['Salary'].sum()

# 4. Count of Employees by Department
count_employees_by_dept = employee_df.groupby('Department').size()

# 5. Max and Min Salaries within the Company
max_salary = employee_df['Salary'].max()
```

```
min_salary = employee_df['Salary'].min()


avg_salary_by_dept, median_age_by_dept, median_years_at_company_by_dept,
total_salary_expense_by_dept, count_employees_by_dept, max_salary, min_salary
```

Output:

```
(Department
 Finance      96113.941176
 HR           97209.111111
 Marketing    101362.526316
 Sales        105005.800000
 Tech          89597.904762
 Name: Salary, dtype: float64,
 Department
 Finance      43.0
 HR           43.0
 Marketing    36.0
 Sales        37.0
 Tech         42.0
 Name: Age, dtype: float64,
 Department
 Finance       8.0
 HR           12.0
 Marketing     6.0
 Sales         8.0
 Tech          8.0
 Name: YearsAtCompany, dtype: float64,
 Department
 Finance      1633937
 HR           1749764
 Marketing    1925888
 Sales        2625145
 Tech         1881556
 Name: Salary, dtype: int64,
 Department
 Finance      17
 HR           18
 Marketing    19
 Sales        25
 Tech         21
 dtype: int64,
 148855,
 51371)
```

Explain the following with appropriate examples:
np.empty
np.arange
np.i
np.linspace
Shape vs reshape
Broadcasting
Numpy stacking
np.block
np.hsplit
np.vsplit
np.dsplit
np.searchsorted(explain parameters)
np.sort and argsort
np.flatten vs np.ravel
np.shuffle
np.unique
np.resize
Transpose
Swapaxes
Inverse
Power
determinant

```
import numpy as np


# np.empty
```

```
empty_array = np.empty((2, 3))

# np.arange
arange_array = np.arange(1, 10, 2)

# np.linspace
linspace_array = np.linspace(0, 1, 5)

# Shape vs reshape
array_reshape_before = np.arange(6)
array_reshape_after = array_reshape_before.reshape((2, 3))

# Broadcasting example
broadcast_a = np.array([1, 2, 3])
broadcast_b = np.array([[0], [1], [2]])
broadcast_result = broadcast_a + broadcast_b

# Numpy stacking
stack_h = np.hstack((broadcast_a.reshape(3, 1), broadcast_b))
stack_v = np.vstack((broadcast_a, broadcast_a))

# np.block
block_example = np.block([[np.zeros((2, 2)), np.eye(2)], [np.ones((2, 2)), np.zeros((2, 2))]])

# np.hsplit
hsplit_example = np.hsplit(block_example, 2)

# np.vsplit
vsplit_example = np.vsplit(block_example, 2)

# np.dsplit example with 3D array
array_3d = np.random.rand(2, 2, 4)
```

```
dsplit_example = np.dsplit(array_3d, 2)

# np.searchsorted
searchsorted_array = np.array([1, 2, 4, 5])
searchsorted_example = np.searchsorted(searchsorted_array, [3, 6])

# np.sort and argsort
sort_array = np.array([3, 1, 2])
sorted_array = np.sort(sort_array)
argsorted_indices = np.argsort(sort_array)

# np.flatten vs np.ravel
flatten_example = block_example.flatten()
ravel_example = block_example.ravel()

# np.shuffle
shuffle_array = np.array([1, 2, 3, 4, 5])
np.random.shuffle(shuffle_array)

# np.unique
unique_array = np.array([1, 2, 2, 3, 3, 3, 4])
unique_values = np.unique(unique_array)

# np.resize
resize_example = np.resize(unique_array, (3, 3))

# Transpose
transpose_example = block_example.T

# Swapaxes
swapaxes_example = np.swapaxes(array_3d, 0, 2)
```

```
# Inverse
matrix_to_inverse = np.array([[1, 2], [3, 4]])
inverse_example = np.linalg.inv(matrix_to_inverse)

# Power
power_example = np.power(broadcast_a, 2)

# determinant
determinant_example = np.linalg.det(matrix_to_inverse)

{
    "empty_array": empty_array,
    "arange_array": arange_array,
    "linspace_array": linspace_array,
    "array_reshape_before": array_reshape_before,
    "array_reshape_after": array_reshape_after,
    "broadcast_result": broadcast_result,
    "stack_h": stack_h,
    "stack_v": stack_v,
    "block_example": block_example,
    "hsplit_example": hsplit_example,
    "vsplit_example": vsplit_example,
    "dsplit_example": dsplit_example,
    "searchsorted_example": searchsorted_example,
    "sorted_array": sorted_array,
    "argsorted_indices": argsorted_indices,
    "flatten_example": flatten_example,
    "ravel_example": ravel_example,
    "shuffle_array": shuffle_array,
    "unique_values": unique_values,
    "resize_example": resize_example,
    "transpose_example": transpose_example,
    "swapaxes_example": swapaxes_example,
    "inverse_example": inverse_example,
    "power_example": power_example,
    "determinant_example": determinant_example,
}
```