In [ ]:

```
Reversing a 2D Array
For 2D arrays, you can reverse along different axes. The syntax is similar, but you specify the axis to reverse.

Reversing along the rows (axis=0): This reverses the order of the rows.

Reversing along the columns (axis=1): This reverses the order of the columns.

Example:

# Create a 2D array
array_2d = np.array([[1, 2, 3],
                     [4, 5, 6],
                     [7, 8, 9]])

# Reverse along rows (axis=0)
reversed_rows = array_2d[::-1]
print("Original 2D array:\n", array_2d)
print("Reversed along rows:\n", reversed_rows)

# Reverse along columns (axis=1)
reversed_columns = array_2d[:, ::-1]
print("Reversed along columns:\n", reversed_columns)
```

In [ ]:

```
#Q1- . Explain the purpose and advantages of NumPy in scientific computing and data analysis. How does it enhance Python's capabilities for numerical operations?
Ans - NumPy, short for Numerical Python, is a powerful library in Python that provides extensive support for numerical operations and scientific computing. Its primary purpose is to facilita

Purpose of NumPy
Array Representation: NumPy introduces the ndarray object, which is a fast, flexible container for large datasets in Python. This allows users to work with data in multi-dimensional arrays m

Performance: NumPy is optimized for performance, leveraging contiguous memory allocation and efficient looping mechanisms. Operations on NumPy arrays can be significantly faster than those o

Mathematical Functions: It provides a rich set of mathematical functions to perform operations such as linear algebra, statistical analysis, Fourier transforms, and more, making it a core to

Interoperability: NumPy serves as a foundation for many other libraries in the scientific computing ecosystem, such as SciPy, Pandas, and Matplotlib, enhancing Python's capabilities for data

Advantages of NumPy
Speed: NumPy operations are implemented in C, allowing them to run faster than pure Python loops. This is especially important for large datasets where performance is crucial.

Memory Efficiency: NumPy arrays require less memory than Python lists due to their fixed data type and contiguous memory allocation, making them more efficient for large-scale data storage.

Convenient Syntax: The syntax for array operations is more concise and expressive compared to traditional Python lists, allowing for cleaner and more readable code.

Broadcasting: NumPy supports broadcasting, which allows operations to be performed on arrays of different shapes without explicit looping. This feature simplifies coding and enhances perform

Comprehensive Functionality: The library offers a wide range of functions for mathematical computations, random number generation, and linear algebra, making it versatile for various scienti

Community and Ecosystem: With a large user base and active community, NumPy benefits from continuous development and a wealth of resources, tutorials, and documentation, facilitating learnin

Enhancements to Python's Numerical Capabilities
N-dimensional Arrays: While Python lists can hold any data type, they are not optimized for numerical operations. NumPy’s ndarray allows for efficient storage and manipulation of numerical d

Element-wise Operations: NumPy allows for element-wise operations on arrays, making it easy to perform mathematical computations without explicit loops.

Advanced Indexing and Slicing: NumPy provides advanced indexing capabilities that simplify the extraction and manipulation of data from arrays, including slicing, masking, and more.

Integration with Other Libraries: NumPy’s array structure is the foundation for other libraries such as SciPy (for scientific computing), Pandas (for data manipulation and analysis), and Mat
```

In [ ]:

```
#Q2- Compare and contrast np.mean() and np.average() functions in NumPy. When would you use one over the other?

Ans - The np.mean() and np.average() functions in NumPy are both used to calculate the average of elements in an array, but they have some key differences in functionality and flexibility.

np.mean()
Purpose: Computes the arithmetic mean (average) of the elements along a specified axis.
Syntax: np.mean(a, axis=None, dtype=None, out=None, keepdims=False)
Default Behavior: By default, it calculates the mean of all elements in the input array.
Parameters:
a: Input array.
axis: Axis or axes along which to compute the mean. Default is None, meaning the mean is computed over the entire array.
dtype: Data type to use for the calculation.
out: A location into which the result is stored (optional).
keepdims: If True, the reduced axes are left in the result as dimensions with size one.
np.average()
Purpose: Computes the weighted average of the elements in an array.
Syntax: np.average(a, axis=None, weights=None, returned=False)
Default Behavior: Computes the mean of all elements if no weights are provided, similar to np.mean().
Parameters:
a: Input array.
axis: Axis or axes along which to compute the average.
weights: An array of weights, same shape as a. If provided, the average is weighted accordingly.
returned: If True, returns a tuple of the average and the sum of the weights.
Key Differences
Functionality:

np.mean() always calculates the arithmetic mean.
np.average() can calculate a weighted average if the weights parameter is provided, making it more versatile.
Use Cases:

Use np.mean() when you need a simple arithmetic mean of your data.
Use np.average() when you want to consider different contributions of values through weights.
When to Use One Over the Other
Use np.mean() when:

You need a straightforward average of values.
You don’t have weights to apply.
Use np.average() when:

You need to calculate a weighted average where some values contribute more to the average than others.
You need additional functionality, such as the option to return the sum of the weights along with the average.
```

In [11]:

```
# ex:-
```

In [ ]:

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

# Using np.mean()
mean_value = np.mean(data)
print("Mean:", mean_value)  # Output: Mean: 3.0

# Using np.average() without weights
average_value = np.average(data)
print("Average:", average_value)  # Output: Average: 3.0

# Using np.average() with weights
weights = np.array([1, 1, 1, 2, 2])  # Giving more weight to the last two elements
weighted_average = np.average(data, weights=weights)
print("Weighted Average:", weighted_average)  # Output: Weighted Average: 3.6
```

In [ ]:

```
#Q3- Describe the methods for reversing a NumPy array along different axes. Provide examples for 1D and 2D arrays.
Ans - Reversing a NumPy array can be done easily using slicing or specific functions. Here’s how you can reverse both 1D and 2D arrays along different axes.

Reversing a 1D Array
To reverse a 1D array, you can use slicing. The syntax array[::-1] effectively reverses the order of elements.

Example:


import numpy as np

# Create a 1D array
array_1d = np.array([1, 2, 3, 4, 5])

# Reverse the 1D array
reversed_1d = array_1d[::-1]
print("Original 1D array:", array_1d)  # Output: [1 2 3 4 5]
print("Reversed 1D array:", reversed_1d)  # Output: [5 4 3 2 1]
```

In [ ]:

```
#Q4- How can you determine the data type of elements in a NumPy array? Discuss the importance of data types in memory management and performance.
Ans - In NumPy, you can determine the data type of elements in an array using the .dtype attribute. This attribute returns the data type of the array's elements, which can be useful for unde

Determining the Data Type
Here’s how you can check the data type of a NumPy array:


import numpy as np

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

# Determine the data type
data_type = array.dtype
print("Data type of the array:", data_type)
```

In [ ]:

```
#Importance of Data Types
Memory Management:

Efficient Storage: Different data types consume different amounts of memory. For example, an int32 takes 4 bytes, while an int64 takes 8 bytes. Using the appropriate data type can significan
Data Type Casting: You can explicitly specify the data type when creating an array (e.g., np.array([1, 2, 3], dtype=np.float32)) to optimize memory usage. This can be crucial in environments
Performance:

Speed of Operations: Operations on arrays of smaller data types (e.g., float32 vs. float64) can be faster because they require less memory bandwidth and processing power. This can lead to be
Vectorized Operations: NumPy’s performance benefits from using contiguous memory blocks for uniform data types. This allows for optimized, vectorized operations, which are much faster than l
Data Integrity:

Type Consistency: Ensuring that all elements in an array are of the same type prevents errors that might arise from mixing data types (e.g., integer and string) and provides clearer semantic
Control Over Numerical Precision: The choice of data type allows control over precision and range. For example, using float16 can save memory but may introduce rounding errors that would not
```

In [ ]:

```
# Q5- Define ndarrays in NumPy and explain their key features. How do they differ from standard Python lists?
Ans - In NumPy, ndarrays (n-dimensional arrays) are the core data structure used for storing and manipulating numerical data. They are highly efficient and optimized for performance in scien

Key Features of ndarrays
Homogeneous Data Types:

All elements in a NumPy array must be of the same data type, which allows for optimized storage and faster computation. This is different from Python lists, which can contain mixed data type
Multidimensional:

Ndarrays can be one-dimensional, two-dimensional, or n-dimensional (hence the name). This flexibility allows for the representation of complex data structures, such as matrices or tensors.
Efficient Memory Usage:

Ndarrays are stored in contiguous memory locations, which improves cache performance and reduces memory overhead. This leads to lower memory usage compared to Python lists.
Vectorized Operations:

NumPy supports vectorized operations, which allow for element-wise operations without the need for explicit loops. This enhances performance and leads to more concise and readable code.
Broadcasting:

NumPy can automatically expand the dimensions of arrays to perform operations on arrays of different shapes. This feature simplifies coding when dealing with different array sizes.
Rich Functionality:

NumPy provides a wide range of built-in functions for mathematical operations, statistical analysis, linear algebra, Fourier transforms, and more. This extensive functionality makes it a pow
Shape and Reshape:

Ndarrays have a shape attribute that defines the size of each dimension. You can easily reshape arrays to different dimensions without altering their data.
Differences from Standard Python Lists
Feature NumPy Ndarrays  Python Lists
Data Type       Homogeneous (same type) Heterogeneous (mixed types)
Memory Layout   Contiguous memory allocation    Non-contiguous memory allocation
Performance     Faster for numerical operations Slower for numerical operations
Operations      Supports vectorized operations  Requires explicit loops
Dimensionality  Can be multi-dimensional        Primarily one-dimensional
Built-in Functions      Extensive mathematical functions        Limited to basic operations
Shape Management        Shape attribute for easy manipulation   No built-in shape management
Example
Here’s a simple example to illustrate the differences:


import numpy as np

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

# Accessing the shape and data type
print("Ndarray shape:", array.shape)  # Output: (2, 3)
print("Ndarray data type:", array.dtype)  # Output: int64 (or int32 depending on the system)

# Vectorized operation
result = array * 2
print("Vectorized operation result:\n", result)

# Creating a Python list
list_data = [[1, 2, 3], [4, 5, 'six']]  # Mixed types

# Accessing elements
print("Python list:", list_data)
```

In [ ]:

```
#Q6- Analyze the performance benefits of NumPy arrays over Python lists for large-scale numerical operations.
Ans - The performance benefits of NumPy arrays over Python lists for large-scale numerical operations are significant and stem from several key factors. Here’s a detailed analysis of why Num

1. Memory Efficiency
Contiguous Memory Allocation: NumPy arrays are stored in contiguous blocks of memory, which enhances cache performance. This allows for faster access times compared to Python lists, which st
Fixed Data Types: NumPy arrays require all elements to be of the same data type, allowing for more compact storage. In contrast, Python lists can contain mixed types, leading to additional m
2. Performance of Numerical Operations
Vectorization: NumPy allows for vectorized operations, meaning that operations can be performed on entire arrays without the need for explicit loops. This is not only syntactically simpler b

For example, adding two arrays element-wise can be done directly:


import numpy as np

# NumPy arrays
a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])
c = a + b  # Vectorized addition

Broadcasting: NumPy supports broadcasting, allowing operations on arrays of different shapes without the need for manual expansion. This feature enables efficient computation while minimizin
3. Use of Optimized Libraries
BLAS and LAPACK: NumPy leverages highly optimized libraries like BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra Package) for performing complex mathematical operations. T
4. Reduced Overhead
Less Overhead: Operations on NumPy arrays have less overhead than those on Python lists. For example, each operation on a Python list involves more checks (like type checking) and higher-lev
5. Parallelism and Optimization
NumPy’s Built-in Optimizations: Many operations in NumPy are optimized for performance through parallel processing and other techniques. This can result in substantial speedups, especially f

Compiled Code: Operations in NumPy are executed in compiled code (C/Fortran), while Python lists rely on interpreted Python code, which is inherently slower.

Performance Comparison Example
To illustrate the performance benefits, consider the following example where we compare the performance of NumPy arrays and Python lists for a large-scale numerical operation, such as elemen


import numpy as np
import time

# Large-scale data
size = 10**6
list_a = list(range(size))
list_b = list(range(size))

# Timing Python list addition
start_time = time.time()
list_result = [a + b for a, b in zip(list_a, list_b)]
print("Python list addition time:", time.time() - start_time)

# Creating NumPy arrays
array_a = np.array(list_a)
array_b = np.array(list_b)

# Timing NumPy array addition
start_time = time.time()
array_result = array_a + array_b
print("NumPy array addition time:", time.time() - start_time)
```

In [ ]:

```
#Q7- Compare vstack() and hstack() functions in NumPy. Provide examples demonstrating their usage and output.
Ans - In NumPy, the vstack() and hstack() functions are used to stack arrays vertically and horizontally, respectively. These functions are particularly useful when you want to combine multi

1. vstack()
Purpose: Stacks arrays in sequence vertically (row-wise). It is equivalent to concatenating along the first axis (axis=0).
Input Requirement: The input arrays must have the same shape along all but the first axis.
Example of vstack():


import numpy as np

# Create two 2D arrays
array1 = np.array([[1, 2, 3],
                   [4, 5, 6]])

array2 = np.array([[7, 8, 9],
                   [10, 11, 12]])

# Stack the arrays vertically
result_vstack = np.vstack((array1, array2))
print("Result of vstack:\n", result_vstack)
```

In [ ]:

```
2. hstack()
Purpose: Stacks arrays in sequence horizontally (column-wise). It is equivalent to concatenating along the second axis (axis=1).
Input Requirement: The input arrays must have the same shape along all but the second axis.
Example of hstack():


# Create two 2D arrays
array1 = np.array([[1, 2, 3],
                   [4, 5, 6]])

array2 = np.array([[7, 8, 9],
                   [10, 11, 12]])

# Stack the arrays horizontally
result_hstack = np.hstack((array1, array2))
print("Result of hstack:\n", result_hstack)
```

In [ ]:

```
#Q8- Explain the differences between fliplr() and flipud() methods in NumPy, including their effects on various array dimensions.
Ans - In NumPy, the fliplr() and flipud() functions are used to flip (reverse) the elements of arrays along specific axes. Here's a detailed explanation of the differences between these two

1. fliplr()
Purpose: Flips an array from left to right (horizontally). It is specifically used for 2D arrays.
Effect: For a 2D array, fliplr() reverses the order of columns.
Example of fliplr():


import numpy as np

# Create a 2D array
array_2d = np.array([[1, 2, 3],
                     [4, 5, 6]])

# Flip the array left to right
flipped_lr = np.fliplr(array_2d)
print("Original array:\n", array_2d)
print("Flipped left to right:\n", flipped_lr)



2. flipud()
Purpose: Flips an array from up to down (vertically). It is also primarily used for 2D arrays.
Effect: For a 2D array, flipud() reverses the order of rows.
Example of flipud():


# Flip the array up to down
flipped_ud = np.flipud(array_2d)
print("Flipped up to down:\n", flipped_ud)
```

In [ ]:

```
# Effects on Various Array Dimensions
2D Arrays:

fliplr(): Reverses columns.
flipud(): Reverses rows.
1D Arrays:

Both functions have the same effect because a 1D array has only one axis to flip. Using either function will reverse the order of elements in the array.
Example:


array_1d = np.array([1, 2, 3, 4])

# Flipping a 1D array
flipped_lr_1d = np.fliplr(array_1d.reshape(1, -1))  # Reshape to 2D for fliplr
flipped_ud_1d = np.flipud(array_1d.reshape(1, -1))  # Reshape to 2D for flipud

print("Original 1D array:", array_1d)
print("Flipped left to right (as 2D):", flipped_lr_1d.flatten())
print("Flipped up to down (as 2D):", flipped_ud_1d.flatten())
```

In [ ]:

```
#Q9- Discuss the functionality of the array_split() method in NumPy. How does it handle uneven splits?
Ans - The array_split() function in NumPy is a versatile method used to split an array into multiple sub-arrays. It is particularly useful when you want to divide data for analysis or proces

Functionality of array_split()

numpy.array_split(ary, indices_or_sections, axis=0)

Parameters:

ary: The input array to be split.
indices_or_sections: This can be either an integer or a sequence of indices. If it’s an integer, it specifies the number of equal sections to split the array into. If it’s a sequence, it spe
axis: The axis along which to split the array. The default is 0, meaning that the split occurs along the first dimension.
Returns: A list of sub-arrays created from the split.
```

In [ ]:

```
import numpy as np

# Create a 1D array
array_1d = np.array([1, 2, 3, 4, 5, 6])

# Split into 3 equal parts
split_equal = np.array_split(array_1d, 3)
print("Equal splits:", split_equal)
```

```
# Split into 4 parts
split_uneven = np.array_split(array_1d, 4)
print("Uneven splits:", split_uneven)
```

In [ ]:
```
#Q10- . Explain the concepts of vectorization and broadcasting in NumPy. How do they contribute to efficient array operations?
Ans - In NumPy, vectorization and broadcasting are two powerful concepts that significantly enhance the efficiency of array operations. They allow for high-performance numerical computations

1. Vectorization
Definition: Vectorization refers to the process of replacing explicit loops in code with array operations that operate on entire arrays (or large chunks of them) at once. This takes advantag

Benefits of Vectorization:
Performance: Vectorized operations are executed at a lower level, leading to faster execution compared to loops written in Python.
Code Clarity: Code becomes cleaner and more readable. Instead of writing loops, you can express operations succinctly.
Example of Vectorization:

Without vectorization, you might write:


import numpy as np

# Create two arrays
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Using a loop
result = np.empty_like(a)
for i in range(len(a)):
    result[i] = a[i] + b[i]
print(result)

With vectorization, you can do:


# Vectorized addition
result_vectorized = a + b
print(result_vectorized)

2. Broadcasting
Definition: Broadcasting is a technique that allows NumPy to perform operations on arrays of different shapes. When performing arithmetic operations on arrays, NumPy automatically expands th

Rules for Broadcasting:
If the arrays have a different number of dimensions, the shape of the smaller-dimensional array is padded with ones on the left side until both shapes are the same.
If the sizes of the dimensions are different, broadcasting occurs when one of the dimensions is 1. The array with size 1 in that dimension is expanded to match the size of the other array.
Benefits of Broadcasting:
Efficiency: Reduces memory usage and computational overhead since it avoids the creation of large temporary arrays.
Simplicity: Enables operations between arrays of different shapes without needing to manually replicate data.
Example of Broadcasting:


# Create a 1D array and a 2D array
a = np.array([1, 2, 3])        # Shape (3,)
b = np.array([[10], [20], [30]])  # Shape (3, 1)

# Broadcasting the operation
result_broadcasted = a + b
print(result_broadcasted)
```

In [ ]:
```
# practical part
```

In [ ]:
```
# Q1- Create a 3x3 NumPy array with random integers between 1 and 100. Then, interchange its rows and columns.

# Original Array:
 [[ 9, 45, 26],
 [32, 55, 37],
 [45,  8, 20]]
```

In [ ]:
```
# Q2. Generate a 1D NumPy array with 10 elements. Reshape it into a 2x5 array, then into a 5x2 array.

# Original 1D Array:
 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]


[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Reshaped 2x5 Array:
 [[0, 1, 2, 3, 4],
 [5, 6, 7, 8, 9]]


[[0, 1, 2, 3, 4], [5, 6, 7, 8, 9]]
```

In [ ]:
```
#Q3. Create a 4x4 NumPy array with random float values. Add a border of zeros around it, resulting in a 6x6 array.

# Original 4x4 Array:
[[0.8678196 , 0.15011539, 0.98441141, 0.51404075],
 [0.47281166, 0.15780332, 0.94808389, 0.47052897],
 [0.75081521, 0.99531155, 0.91992204, 0.17253079],
 [0.58034472, 0.62934717, 0.83240124, 0.12760527]]


[[0.8678196, 0.15011539, 0.98441141, 0.51404075],
 [0.47281166, 0.15780332, 0.94808389, 0.47052897],
 [0.75081521, 0.99531155, 0.91992204, 0.17253079],
 [0.58034472, 0.62934717, 0.83240124, 0.12760527]]

# 6x6 Array with Border of Zeros:
[[0.        , 0.        , 0.        , 0.        , 0.        , 0.        ],
 [0.        , 0.8678196 , 0.15011539, 0.98441141, 0.51404075, 0.        ],
 [0.        , 0.47281166, 0.15780332, 0.94808389, 0.47052897, 0.        ],
 [0.        , 0.75081521, 0.99531155, 0.91992204, 0.17253079, 0.        ],
 [0.        , 0.58034472, 0.62934717, 0.83240124, 0.12760527, 0.        ],
 [0.        , 0.        , 0.        , 0.        , 0.        , 0.        ]]


[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
 [0.0, 0.8678196, 0.15011539, 0.98441141, 0.51404075, 0.0],
 [0.0, 0.47281166, 0.15780332, 0.94808389, 0.47052897, 0.0],
 [0.0, 0.75081521, 0.99531155, 0.91992204, 0.17253079, 0.0],
 [0.0, 0.58034472, 0.62934717, 0.83240124, 0.12760527, 0.0],
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]
```

In [ ]:
```
#Q4. Using NumPy, create an array of integers from 10 to 60 with a step of 5.

import numpy as np

# Create an array of integers from 10 to 60 with a step of 5
arr = np.arange(10, 61, 5)

# Print the result
print(arr)
```

In [ ]:
```
#Q5. Create a NumPy array of strings ['python', 'numpy', 'pandas']. Apply different case transformations (uppercase, lowercase, title case, etc.) to each element.

import numpy as np

# Create a NumPy array of strings
arr = np.array(['python', 'numpy', 'pandas'])

# Apply different case transformations
upper_case = np.char.upper(arr)           # Uppercase
lower_case = np.char.lower(arr)           # Lowercase
title_case = np.char.title(arr)           # Title Case
capitalize_case = np.char.capitalize(arr) # Capitalize

# Print the results
print("Uppercase:", upper_case)
print("Lowercase:", lower_case)
print("Title Case:", title_case)
print("Capitalize:", capitalize_case)
```

In [ ]:
```
#Q6. Generate a NumPy array of words. Insert a space between each character of every word in the array.

import numpy as np

# Create a NumPy array of words
words = np.array(['adnan', 'qazi', 'ayan', 'sadaf'])

# Insert a space between each character of every word
spaced_words = np.char.join(' ', words)

# Print the result
print(spaced_words)
```

In [ ]:
```
#Q7. Create two 2D NumPy arrays and perform element-wise addition, subtraction, multiplication, and division.

import numpy as np

# Create two 2D NumPy arrays
array1 = np.array([[1, 2, 3], [4, 5, 6]])
array2 = np.array([[7, 8, 9], [10, 11, 12]])

# Perform element-wise addition
addition = array1 + array2

# Perform element-wise subtraction
subtraction = array1 - array2

# Perform element-wise multiplication
multiplication = array1 * array2

# Perform element-wise division
division = array1 / array2

# Print the results
print("Addition:\n", addition)
print("Subtraction:\n", subtraction)
print("Multiplication:\n", multiplication)
print("Division:\n", division)
```

In [ ]:
```
#Q8. Use NumPy to create a 5x5 identity matrix, then extract its diagonal elements.

import numpy as np

# Create a 5x5 identity matrix
identity_matrix = np.eye(5)

# Extract the diagonal elements
diagonal_elements = np.diagonal(identity_matrix)

# Print the results
print("Identity Matrix:\n", identity_matrix)
print("Diagonal Elements:", diagonal_elements)
```

In [ ]:
```
#Q9. Generate a NumPy array of 100 random integers between 0 and 1000. Find and display all prime numbers in this array.

import numpy as np

# Function to check if a number is prime
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

# Generate a NumPy array of 100 random integers between 0 and 1000
random_integers = np.random.randint(0, 1000, size=100)

# Find and display all prime numbers in the array
prime_numbers = [num for num in random_integers if is_prime(num)]

# Print the results
print("Random Integers:", random_integers)
print("Prime Numbers:", prime_numbers)
```

In [ ]:
```
#Q10. Create a NumPy array representing daily temperatures for a month. Calculate and display the weekly averages.

import numpy as np

# Create a NumPy array representing daily temperatures for a month (30 days)
np.random.seed(0)  # For reproducibility
daily_temperatures = np.random.uniform(32, 90, 30)  # Fahrenheit

# Reshape array to 5 weeks, with the last week having fewer days if needed
# The total number of elements in the reshaped array must match the original array
num_weeks = 5  # Change this to the desired number of weeks
weekly_temperatures = daily_temperatures.reshape(num_weeks, -1) #-1 is inferred from the length of the array and remaining dimension which is 5
# Calculate weekly averages
weekly_averages = np.mean(weekly_temperatures, axis=1)

# Display daily temperatures and weekly averages
print("Daily Temperatures:")
print(daily_temperatures)

print("\nWeekly Temperatures:")
print(weekly_temperatures)

print("\nWeekly Averages:")
for i, avg in enumerate(weekly_averages):
    print(f"Week {i + 1}: {avg:.2f}°F")
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: