Question 1

What is NumPy, and why is it important in Machine Learning?

Theory

NumPy (Numerical Python) is the foundational open-source library for numerical computing in Python. Its core feature is the powerful N-dimensional array object, the **ndarray**, which provides a fast and memory-efficient way to store and manipulate homogeneous numerical data.

Why is it Important in Machine Learning?

NumPy is the bedrock upon which the entire Python data science and machine learning ecosystem is built. Its importance stems from two key areas: **performance** and **functionality**.

1. **Performance and Efficiency**:
    - **Vectorization**: NumPy allows you to perform mathematical operations on entire arrays at once, a concept known as vectorization. This avoids slow, explicit for loops in Python.
    - **C Backend**: NumPy ndarrays are dense data structures stored in contiguous blocks of memory. The operations on them are implemented in highly optimized, pre-compiled C or Fortran code.
    - **Result**: These factors make NumPy operations orders of magnitude faster than equivalent operations on standard Python lists. This speed is non-negotiable for the computationally intensive tasks common in machine learning.
2.
3. **The Standard for Data Representation**:
    - The NumPy ndarray is the universal language for numerical data in Python. All major machine learning libraries, including **Pandas, Scikit-learn, TensorFlow, and PyTorch**, are designed to operate on or seamlessly integrate with NumPy arrays.
    - A **dataset** is represented as a 2D array (matrix).
    - An **image** is a 3D array (height x width x channels).
    - **Model parameters** (weights and biases) are stored in arrays.
4.
5. **Comprehensive Mathematical Functionality**:
    - **Linear Algebra**: NumPy provides a robust linalg module for essential linear algebra operations like dot products, matrix multiplication, determinants, inverses, and decompositions (SVD, QR), which are the mathematical foundation of many ML algorithms.
    - **Statistical Operations**: Functions for calculating mean, median, standard deviation, variance, etc., are built-in and highly optimized.

    - **Random Number Generation**: A sophisticated random module for creating arrays of random numbers from various distributions, crucial for weight initialization, data augmentation, and stochastic algorithms.
6.

In summary, NumPy provides the high-performance array object and the mathematical machinery required to implement and execute ML algorithms efficiently. Without it, the Python ML ecosystem would be drastically slower and less capable.

Question 2

Explain how NumPy arrays are different from Python lists.

Theory

While both NumPy arrays and Python lists are used to store collections of items, they are fundamentally different in their design, performance, and functionality. These differences make NumPy arrays the superior choice for numerical computation.

Key Differences

1. **Data Type (Homogeneity vs. Heterogeneity)**:
    - **Python List**: Can store elements of **different data types** (heterogeneous). A single list can contain integers, strings, and other objects.
    - **NumPy Array**: Must contain elements of the **same data type** (homogeneous). All elements must be, for example, an int32 or a float64. This homogeneity is the key to its performance.
2.
3. **Memory Usage**:
    - **Python List**: Stores a collection of **pointers** to objects scattered around in memory. This introduces significant memory overhead for each element.
    - **NumPy Array**: Stores data in a **contiguous block of memory**. There are no pointers for individual elements. This makes NumPy arrays much more memory-efficient.
4.
5. **Performance**:
    - **Python List**: Operations on lists often require slow, explicit Python for loops.
    - **NumPy Array**: Supports **vectorized operations**, which are executed in fast, compiled C code. This avoids the overhead of the Python interpreter, making NumPy operations orders of magnitude faster.
6.
7. **Functionality**:

   - **Python List**: Provides basic operations like appending, inserting, and removing elements. Mathematical operations are not built-in.
   - **NumPy Array**: Designed specifically for numerical computation. It comes with a vast library of mathematical functions, from basic arithmetic to complex linear algebra, that can be applied to entire arrays.
8.

Summary Table

Feature	Python List	NumPy Array (ndarray)
Data Types	Heterogeneous (mixed types)	Homogeneous (fixed type)
Memory Layout	Pointers to objects (non-contiguous)	Contiguous block of memory
Performance	Slow (interpreted Python loops)	Fast (vectorized C/Fortran code)
Functionality	General-purpose collection	Optimized for numerical computation
Example	[1, "two", 3.0]	np.array([1, 2, 3], dtype=np.int32)

Code Example

Generated python
```
    import numpy as np
import sys

# Memory Usage Comparison
py_list = list(range(1000))
np_array = np.arange(1000)

print(f"Memory size of Python list: {sys.getsizeof(py_list)} bytes")
print(f"Memory size of NumPy array: {np_array.nbytes} bytes")

# Performance Comparison
# %timeit [i*2 for i in py_list] # Run this in a Jupyter Notebook
# %timeit np_array * 2           # Run this in a Jupyter Notebook
# The NumPy version will be significantly faster.
```

Question 3

What are the main attributes of a NumPy ndarray?

Theory

A NumPy ndarray object is more than just an array of data; it contains important metadata that describes the array's structure and how to interpret its data. These attributes can be accessed directly from the array object.

Main Attributes

Let's consider the following array for demonstration:
arr = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.int16)

1. **ndarray.ndim**:
    - **Description**: The number of dimensions (or axes) of the array.
    - **Example**: For arr, arr.ndim would be 2. A vector would have ndim of 1.
2.
3. **ndarray.shape**:
    - **Description**: A tuple of integers indicating the size of the array in each dimension.
    - **Example**: For arr, arr.shape would be (2, 3), meaning it has 2 rows and 3 columns.
4.
5. **ndarray.size**:
    - **Description**: The total number of elements in the array. This is equal to the product of the elements of shape.
    - **Example**: For arr, arr.size would be 2 * 3 = 6.
6.
7. **ndarray.dtype**:
    - **Description**: An object describing the data type of the elements in the array.
    - **Example**: For arr, arr.dtype would be int16 (a 16-bit integer). This is a crucial attribute for memory optimization.
8.
9. **ndarray.itemsize**:
    - **Description**: The size in bytes of each element of the array.
    - **Example**: For arr with dtype=int16 (which is 2 bytes), arr.itemsize would be 2.
10.
11. **ndarray.nbytes**:
    - **Description**: The total size in bytes of the array's data. It is equal to itemsize * size.
    - **Example**: For arr, arr.nbytes would be 2 * 6 = 12.
12.
13. **ndarray.T**:
    - **Description**: An attribute that returns the transposed version of the array. It's a shortcut for the transpose() method.
    - **Example**: arr.T would have a shape of (3, 2).
14.

Code Example

Generated python

```
    import numpy as np

arr = np.array([[1, 2, 3, 4],
                [5, 6, 7, 8]], dtype=np.int32)

print(f"The array:\n{arr}\n")
print(f"arr.ndim: {arr.ndim}")
print(f"arr.shape: {arr.shape}")
print(f"arr.size: {arr.size}")
print(f"arr.dtype: {arr.dtype}")
print(f"arr.itemsize: {arr.itemsize} bytes")
print(f"arr.nbytes: {arr.nbytes} bytes")
print(f"arr.T (transpose):\n{arr.T}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code

```
    The array:
[[1 2 3 4]
 [5 6 7 8]]

arr.ndim: 2
arr.shape: (2, 4)
arr.size: 8
arr.dtype: int32
arr.itemsize: 4 bytes
arr.nbytes: 32 bytes
arr.T (transpose):
[[1 5]
 [2 6]
 [3 7]
 [4 8]]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Question 4

Explain the concept of broadcasting in NumPy.

Theory

Broadcasting is a powerful mechanism in NumPy that allows for performing arithmetic operations on arrays of **different shapes**. When certain constraints are met, NumPy can implicitly "broadcast" the smaller array across the larger array so that they have compatible shapes for element-wise operations.

This is a crucial feature that avoids the need to create explicit copies of data, making code more concise and memory-efficient.

The Broadcasting Rules

Broadcasting works by comparing the shapes of the two arrays element-wise, starting from the trailing (rightmost) dimensions. Two dimensions are compatible if:

1. They are equal, OR
2. One of them is 1.

If these conditions are not met, a ValueError is raised. If the number of dimensions of the two arrays is different, the smaller array is padded with 1s on its left side.

Example Walkthrough

Let's add a 1D array b to a 2D array A.

- A has shape (4, 3)
- b has shape (3,)

Generated code
```
   A:     4 x 3
b:        3
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

1. **Rule 1**: Compare the trailing dimensions. 3 == 3. They are compatible.

2. **Rule 2**: Move to the next dimension. A has a dimension of 4, but b has no more dimensions. NumPy "stretches" or "broadcasts" b's shape to match A's. The b array is conceptually duplicated along the new axis to match the shape of A.

Effectively, the operation A + b is treated as:

Generated code
```
    [[1, 2, 3],      [[10, 20, 30],
 [4, 5, 6],   +   [10, 20, 30],
 [7, 8, 9],       [10, 20, 30],
 [10, 11, 12]]    [10, 20, 30]]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

This duplication does not actually happen in memory; it's a virtual process that makes the computation efficient.

Code Example

Generated python
```
    import numpy as np

# Example 1: Adding a 1D array to a 2D array
A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])  # Shape (3, 3)

b = np.array([10, 20, 30]) # Shape (3,)

# b is broadcast across each row of A
result1 = A + b
print(f"A (3,3) + b (3,):\n{result1}\n")

# Example 2: Adding a 2D column vector to a 2D array
c = np.array([[100], [200], [300]]) # Shape (3, 1)

# c is broadcast across each column of A
result2 = A + c
print(f"A (3,3) + c (3,1):\n{result2}\n")

# Example 3: Incompatible shapes
d = np.array([1, 2]) # Shape (2,)
try:
    A + d # Will raise a ValueError
except ValueError as e:
    print(f"Error adding A (3,3) and d (2,): {e}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code

```
     A (3,3) + b (3,):
[[11 22 33]
 [14 25 36]
 [17 28 39]]

A (3,3) + c (3,1):
[[101 102 103]
 [204 205 206]
 [307 308 309]]

Error adding A (3,3) and d (2,): operands could not be broadcast together with shapes (3,3) (2,)
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Question 5

What are the data types supported by NumPy arrays?

Theory

NumPy provides a rich set of numerical data types, called **dtypes**, which are much more specific and memory-efficient than Python's standard types. These dtypes are crucial for optimizing performance and memory usage, as they allow you to specify the exact size and type of the data being stored.

Each dtype is essentially a wrapper around a C data type.

Main Categories of Data Types

1. **Integers**:
    - **Signed Integers**: int8, int16, int32, int64. The number indicates the number of bits used to store the value. An int8 can store values from -128 to 127.
    - **Unsigned Integers**: uint8, uint16, uint32, uint64. These can only store non-negative values. A uint8 can store values from 0 to 255 (often used for image data).
2.
3. **Floating-Point Numbers**:
    - float16 (half precision), float32 (single precision), float64 (double precision).
    - float64 is the default floating-point type in Python and NumPy, but in machine learning, float32 is often used to save memory and speed up computations on GPUs.
4.
5. **Complex Numbers**:
    - complex64, complex128. These are composed of two floats (real and imaginary parts).
6.
7. **Booleans**:
    - bool_. Stores True or False values, typically using one byte of memory.
8.
9. **Strings**:
    - string_: Fixed-length ASCII string. You must specify the length, e.g., S5 for a 5-character string.
    - unicode_: Fixed-length Unicode string.
10.
11. **Objects**:
    - object_. A generic type that can store any Python object. Using this dtype essentially removes the performance and memory benefits of NumPy, as the array just stores pointers to objects, similar to a Python list. It should be avoided if possible.
12.

Code Example

Generated python

```
   import numpy as np

# Specifying dtypes at creation
arr_int8 = np.array([1, 2, 3], dtype=np.int8)
arr_float32 = np.array([1.0, 2.5, 3.8], dtype=np.float32)
arr_bool = np.array([True, False, True], dtype=np.bool_)

print(f"Array: {arr_int8}, dtype: {arr_int8.dtype}, itemsize: {arr_int8.itemsize} byte")
print(f"Array: {arr_float32}, dtype: {arr_float32.dtype}, itemsize: {arr_float32.itemsize} bytes")
print(f"Array: {arr_bool}, dtype: {arr_bool.dtype}, itemsize: {arr_bool.itemsize} byte")
```

```
# Type casting
arr_float64 = np.array([1, 2, 3]) # Defaults to int64 or int32 depending on system
print(f"\nOriginal array dtype: {arr_float64.dtype}")

# Cast to a different type
arr_as_float = arr_float64.astype(np.float64)
print(f"Casted array dtype: {arr_as_float.dtype}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Question 6

What is the difference between a deep copy and a shallow copy in NumPy?

Theory

The concepts of deep copy and shallow copy relate to how NumPy arrays are duplicated. The distinction is crucial for understanding how modifications to one array might affect another and for managing memory.

In NumPy, these concepts are often referred to as **copy** and **view**.

Shallow Copy (View)

- **Concept**: A shallow copy, or a **view**, creates a new array object but **does not create a copy of the underlying data**. The new array object "looks at" or "points to" the same block of memory as the original array.
- **How to Create**: Slicing an array creates a view.
- **Key Characteristic**: Any modifications made to the view **will affect** the original array, and vice versa.
- **Benefit**: It is extremely fast and memory-efficient because no data is copied.

Deep Copy (Copy)

- **Concept**: A deep copy, or simply a **copy**, creates a completely new array object **and a new copy of the underlying data**.
- **How to Create**: Using the .copy() method.
- **Key Characteristic**: The new array and the original array are completely independent. Modifications made to the copy **will not affect** the original array.

- **Benefit**: It provides true isolation, which is safer if you need to modify an array without changing the original data source.

Assignment (No Copy)

- It's also important to distinguish these from a simple assignment (b = a). This does **not** create a new array object at all. It simply makes the new variable name b point to the *exact same* array object as a.

Code Example

Generated python

```
    import numpy as np

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

# --- 1. No Copy (Assignment) ---
b = a
print(f"--- Assignment (b = a) ---")
print(f"a before change: {a}")
b[0] = 99
print(f"b was changed. a is now: {a}") # a is affected
print(f"id(a) == id(b): {id(a) == id(b)}\n") # They are the same object

# --- 2. Shallow Copy (View) ---
a = np.array([1, 2, 3, 4, 5]) # Reset a
c = a[1:4] # Slicing creates a view
print(f"--- Shallow Copy / View (c = a[1:4]) ---")
print(f"a before change: {a}")
print(f"c initial: {c}")
c[0] = 88
print(f"c was changed. a is now: {a}") # a is affected
print(f"id(a) == id(c): {id(a) == id(c)}\n") # They are different objects...
print(f"but c.base is a: {c.base is a}") # ...but c shares data with a

# --- 3. Deep Copy (Copy) ---
a = np.array([1, 2, 3, 4, 5]) # Reset a
d = a.copy()
print(f"--- Deep Copy (d = a.copy()) ---")
print(f"a before change: {a}")
d[0] = 77
print(f"d was changed. a is now: {a}") # a is NOT affected
print(f"id(a) == id(d): {id(a) == id(d)}") # They are different objects
print(f"d.base is None: {d.base is None}") # d does not share data
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Question 7

What are universal functions (ufuncs) in NumPy?

Theory

Universal functions, or **ufuncs**, are functions in NumPy that operate on ndarrays in an **element-by-element** fashion. They are the core of NumPy's power, providing a way to perform fast, vectorized operations on arrays.

A ufunc is a "vectorized" wrapper for a simple function that takes a fixed number of scalar inputs and produces a fixed number of scalar outputs.

Key Characteristics of Ufuncs

1. **Vectorization**: They operate on entire arrays without needing an explicit Python for loop. This is why they are so fast.
2. **Broadcasting**: They fully support broadcasting, allowing you to perform operations on arrays of different but compatible shapes.
3. **C Implementation**: The underlying loops are implemented in compiled C code, which is much faster than interpreted Python.
4. **Type Casting**: They can automatically handle casting between different dtypes when necessary.

Types of Ufuncs

Ufuncs can be categorized based on the number of inputs they take:

- **Unary Ufuncs**: Operate on a single array.
  - Examples: np.sqrt(), np.exp(), np.sin(), np.log(), np.abs().
-
- **Binary Ufuncs**: Operate on two arrays.
  - Examples: np.add(), np.subtract(), np.multiply(), np.maximum(). These are also available through standard arithmetic operators like +, -, *.
-

Code Example

Generated python

```
import numpy as np
```

```
# --- Unary Ufunc Example ---
a = np.array([1, 4, 9, 16])
print(f"Original array a: {a}")
print(f"np.sqrt(a): {np.sqrt(a)}\n")

# --- Binary Ufunc Example with Broadcasting ---
b = np.arange(9).reshape(3, 3)
c = np.array([10, 20, 30])

print(f"Array b:\n{b}")
print(f"Array c: {c}")

# np.add is a ufunc. The '+' operator is a convenient shortcut.
result = np.add(b, c) # Equivalent to b + c
print(f"np.add(b, c):\n{result}\n")

# --- Creating a custom ufunc from a Python function ---
def my_custom_logic(x):
    """A simple Python function."""
    if x > 5:
        return x * 2
    else:
        return x / 2

# Vectorize the Python function to create a ufunc
custom_ufunc = np.vectorize(my_custom_logic)

d = np.array([1, 6, 3, 8])
print(f"Original array d: {d}")
print(f"Applying custom ufunc: {custom_ufunc(d)}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Note: np.vectorize is provided for convenience and is not as fast as native C-implemented ufuncs, as it still involves a Python loop internally. However, it's a great tool for applying custom logic to arrays in a clean way.

Question 8

What is the use of the axis parameter in NumPy functions?

Theory

The axis parameter is a fundamental concept in NumPy that allows you to perform operations along a **specific dimension** of a multi-dimensional array. It specifies the axis along which an aggregate function (like sum, mean, max) should be computed.

Understanding Axes

In a multi-dimensional array, each dimension is an axis.

- For a **2D array** (a matrix):
    - axis=0 refers to the **vertical** dimension (down the rows).
    - axis=1 refers to the **horizontal** dimension (across the columns).
-
- For a **3D array**:
    - axis=0 is the first dimension (depth).
    - axis=1 is the second dimension (rows).
    - axis=2 is the third dimension (columns).
-

When you use an aggregate function with the axis parameter, that dimension is **collapsed**, and the result is an array with one fewer dimension.

Code Example

Generated python

```
    import numpy as np

# A 2D array (3 rows, 4 columns)
arr = np.array([[1, 2, 3, 4],
             [5, 6, 7, 8],
             [9, 10, 11, 12]])

print(f"Original array (shape {arr.shape}):\n{arr}\n")

# --- No axis parameter ---
# The operation is performed on all elements of the array, returning a single scalar value.
total_sum = arr.sum()
print(f"arr.sum() (no axis): {total_sum}")
print("-" * 30)

# --- axis=0 ---
# The operation is performed down each COLUMN.
# The result is a 1D array with a length equal to the number of columns.
sum_axis_0 = arr.sum(axis=0)
print(f"arr.sum(axis=0): {sum_axis_0}")
print(f"Result shape: {sum_axis_0.shape}")
# Interpretation: [1+5+9, 2+6+10, 3+7+11, 4+8+12]
print("-" * 30)

# --- axis=1 ---
# The operation is performed across each ROW.
# The result is a 1D array with a length equal to the number of rows.
sum_axis_1 = arr.sum(axis=1)
print(f"arr.sum(axis=1): {sum_axis_1}")
print(f"Result shape: {sum_axis_1.shape}")
# Interpretation: [1+2+3+4, 5+6+7+8, 9+10+11+12]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code

```
    Original array (shape (3, 4)):
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

arr.sum() (no axis): 78
------------------------------
arr.sum(axis=0): [15 18 21 24]
Result shape: (4,)
------------------------------
arr.sum(axis=1): [10 26 42]
Result shape: (3,)
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Use Case in Machine Learning

The axis parameter is used constantly. For example, to **standardize the features** of a dataset X (where rows are samples and columns are features):

1. Calculate the mean of each feature: mean = X.mean(axis=0).
2. Calculate the standard deviation of each feature: std = X.std(axis=0).

3. Standardize the data: X_scaled = (X - mean) / std. (Broadcasting handles the subtraction and division here).

Question 9

Explain the use of slicing and indexing with NumPy arrays.

Theory

Slicing and indexing are the primary ways to access and modify subsets of data within a NumPy array. NumPy provides a rich and powerful set of indexing techniques that go far beyond what is possible with standard Python lists.

1. Basic Slicing

- **Concept**: Similar to Python lists, you can slice arrays using the start:stop:step notation.
- **Key Difference**: Unlike Python lists, array slices are **views** (shallow copies) of the original array, not new copies. Modifying a slice will modify the original array.
- **Multi-dimensional Slicing**: You can slice along multiple dimensions by separating the slice notation with a comma.

2. Advanced Indexing

NumPy's advanced indexing is triggered when the selection object is a non-tuple sequence object, an ndarray of integers or booleans. Advanced indexing always returns a **copy** of the data, not a view.

- **Integer Array Indexing**:
    - **Concept**: You can use arrays of integers to select specific elements.
    - **Use Case**: Allows you to select arbitrary rows or columns in any order, or to pick specific elements from an array.
-
- **Boolean Array Indexing**:
    - **Concept**: You can use a boolean array (of the same shape as the original array) to select elements. The elements corresponding to a True value in the boolean array are selected.
    - **Use Case**: This is extremely powerful for filtering data based on a condition.
-

Code Example

Generated python

```
    import numpy as np

arr = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])

# --- Basic Slicing ---
print("--- Basic Slicing ---")
# Get the first two rows and columns 1 through 2
slice_view = arr[:2, 1:3]
print(f"Slice (arr[:2, 1:3]):\n{slice_view}\n")

# Modifying the slice affects the original array
slice_view[0, 0] = 99
print(f"Original array after modifying slice:\n{arr}\n")

# --- Integer Array Indexing ---
print("--- Integer Indexing ---")
arr = np.array([10, 20, 30, 40, 50])
# Select elements at indices 0, 2, and 4
int_indexed = arr[[0, 2, 4]]
print(f"arr[[0, 2, 4]]: {int_indexed}\n")

arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# Select rows 0 and 2
print(f"arr_2d[[0, 2]]:\n{arr_2d[[0, 2]]}\n")

# --- Boolean Array Indexing ---
print("--- Boolean Indexing ---")
arr = np.array([1, 2, 3, 4, 5, 6])
# Create a boolean mask for even numbers
mask = arr % 2 == 0
print(f"Original array: {arr}")
print(f"Boolean mask (arr % 2 == 0): {mask}")

# Use the mask to select only the even numbers
even_numbers = arr[mask]
print(f"arr[mask]: {even_numbers}\n")

# This is extremely common in data preprocessing
# e.g., Replace all values greater than 50 with 50
data = np.array([10, 60, 20, 75, 40])
data[data > 50] = 50
print(f"Capping values > 50: {data}")
```

IGNORE_WHEN_COPYING_START
content_copy download

Use code with caution. Python
IGNORE_WHEN_COPYING_END

Question 10

What is the purpose of the NumPy histogram function?

Theory

The numpy.histogram function is a powerful tool for understanding the **distribution** of a set of data. It does not draw a plot; instead, it is a **computational function** that calculates the data needed to create a histogram.

A histogram works by:

1. Dividing the entire range of data values into a series of intervals, called **bins**.
2. Counting how many data points fall into each bin.

The np.histogram function returns two things:

- An array containing the **count** of values in each bin.
- An array representing the **edges** of the bins.

This raw data can then be used by a plotting library like Matplotlib to visualize the histogram.

Code Example

Generated python
```
    import numpy as np
import matplotlib.pyplot as plt

# Generate some sample data from a normal distribution
np.random.seed(42)
data = np.random.randn(1000)

# --- 1. Use np.histogram to compute the histogram data ---
# We specify 20 bins for our data
counts, bin_edges = np.histogram(data, bins=20)

print("--- np.histogram Output ---")
print(f"Number of counts: {len(counts)}")
print(f"Counts per bin: {counts}\n")
print(f"Number of bin edges: {len(bin_edges)}")
print(f"Bin edges: {np.round(bin_edges, 2)}")
```

```
# Note: There is always one more bin edge than there are counts.

# --- 2. Use Matplotlib to visualize the histogram ---
# We can use the computed counts and bin edges with plt.bar
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
# The width of the bars is the difference between the bin edges
plt.bar(bin_edges[:-1], counts, width=np.diff(bin_edges), edgecolor='black')
plt.title('Histogram from np.histogram Data')

# For comparison, here is Matplotlib's built-in histogram function,
# which uses np.histogram under the hood.
plt.subplot(1, 2, 2)
plt.hist(data, bins=20, edgecolor='black')
plt.title('Histogram using plt.hist()')

plt.tight_layout()
plt.show()
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Purpose and Use Cases

- **Exploratory Data Analysis (EDA)**: Its primary purpose is to help you understand the underlying frequency distribution of a dataset. You can quickly see if the data is symmetric, skewed, bimodal, etc.
- **Probability Density Estimation**: The histogram is a simple form of a non-parametric density estimator. By normalizing it (dividing the counts by the total number of samples and the bin width), it can approximate the probability density function (PDF) of the data.
- **Feature Engineering**: The shape of a feature's histogram can inform preprocessing steps. For example, if a feature is highly skewed, you might apply a log transformation to make it more symmetric.
- **Foundation for Plotting Libraries**: As shown in the example, high-level plotting functions like matplotlib.pyplot.hist use numpy.histogram as their computational engine to do the actual binning and counting before any plotting happens.

Question 11

What is the difference between np.var() and np.std()?

Theory

np.var() and np.std() are both fundamental statistical functions in NumPy used to measure the **dispersion** or **spread** of a set of data points around their mean. They are very closely related.

Variance (np.var())

- **Definition**: Variance measures the **average of the squared differences** from the mean.
- **Formula**: $Var(X) = (1/n) * \Sigma(x_i - \mu)^2$
    - n: number of data points
    - x_i: each data point
    - μ: the mean of the data
-
- **Interpretation**:
    - A small variance indicates that the data points are clustered closely around the mean.
    - A large variance indicates that the data points are spread out.
    - **The units of variance are the square of the original data's units**. For example, if your data is in meters, the variance will be in meters squared. This makes it less intuitive to interpret directly.
-

Standard Deviation (np.std())

- **Definition**: The standard deviation is simply the **square root of the variance**.
- **Formula**: $Std(X) = \sqrt{Var(X)} = \sqrt{[(1/n) * \Sigma(x_i - \mu)^2]}$
- **Interpretation**:
    - The standard deviation is the most common and intuitive measure of spread.
    - Crucially, **the units of the standard deviation are the same as the original data's units**. For example, if your data is in meters, the standard deviation will also be in meters.
    - This makes it much easier to interpret. You can say things like "the typical deviation from the mean is 5 meters."
-

Key Difference and Relationship

- **Relationship**: The standard deviation is the square root of the variance. std = sqrt(var).
- **Key Difference**: **Units**. The standard deviation is in the same units as the original data, making it directly interpretable, while the variance is in squared units.

Because of its interpretability, the **standard deviation is almost always the preferred metric** to report when describing the spread of data.

Code Example

Generated python

```
    import numpy as np

data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
mean = np.mean(data) # The mean is 5

# Calculate variance
variance = np.var(data)

# Calculate standard deviation
std_dev = np.std(data)

print(f"Data: {data}")
print(f"Mean: {mean}")
print(f"Variance (np.var): {variance:.4f}")
print(f"Standard Deviation (np.std): {std_dev:.4f}")

# Verify the relationship
print(f"\nIs std_dev the square root of variance? {np.isclose(std_dev, np.sqrt(variance))}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code
```
    Data: [1 2 3 4 5 6 7 8 9]
Mean: 5.0
Variance (np.var): 6.6667
Standard Deviation (np.std): 2.5820

Is std_dev the square root of variance? True
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Note on ddof: Both functions have a ddof (Delta Degrees of Freedom) parameter. By default, ddof=0, which calculates the population variance/std. In statistics, when estimating the variance from a sample, you often use ddof=1 (dividing by n-1 instead of n) to get an unbiased estimate. Pandas' .std() method uses ddof=1 by default, which is a common point of confusion.

Question 12

What is the concept of vectorization in NumPy?

Theory

Vectorization is the process of rewriting code that uses explicit for loops to iterate over elements of an array into equivalent expressions that use array operations. In the context of NumPy, this means replacing slow Python loops with highly optimized operations that are implemented in pre-compiled C or Fortran code.

It is the core concept that makes NumPy so fast and is a fundamental paradigm for numerical computing in Python.

Why is Vectorization Important?

1. **Performance**: The primary reason is speed.
   - **Python Loops**: Are slow because the Python interpreter has to process each step of the loop, perform type checks, and handle other overhead for every single element.
   - **Vectorized Operations**: A single statement like c = a + b on NumPy arrays is handed off to a highly optimized, compiled C function. This function executes a single loop in C over the contiguous memory of the arrays, which is orders of magnitude faster than the equivalent Python loop.
2.
3. **Code Readability and Conciseness**:
   - Vectorized code is much cleaner, more compact, and easier to read. It often looks closer to the original mathematical notation.
   - c = a * b is much more intuitive than a nested loop structure.
4.

Code Example

Let's compare the performance of adding two large vectors using a Python loop versus a vectorized NumPy operation.

Generated python
```
    import numpy as np
import time

# Create two large arrays
size = 1_000_000
a = np.random.rand(size)
b = np.random.rand(size)

# --- Method 1: Using a Python for loop ---
```

```
start_time = time.time()
c_loop = np.zeros(size)
for i in range(size):
    c_loop[i] = a[i] + b[i]
end_time = time.time()
loop_duration = end_time - start_time
print(f"Time taken with Python for loop: {loop_duration:.6f} seconds")

# --- Method 2: Vectorized NumPy operation ---
start_time = time.time()
c_vectorized = a + b
end_time = time.time()
vectorized_duration = end_time - start_time
print(f"Time taken with vectorized NumPy: {vectorized_duration:.6f} seconds")

# --- Verification and Performance Comparison ---
print(f"\nAre the results the same? {np.allclose(c_loop, c_vectorized)}")
if vectorized_duration > 0:
    print(f"Vectorized version is approximately {loop_duration / vectorized_duration:.2f} times
faster.")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Example Output:

Generated code
```
     Time taken with Python for loop: 0.178523 seconds
Time taken with vectorized NumPy: 0.002000 seconds

Are the results the same? True
Vectorized version is approximately 89.26 times faster.
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

This simple example demonstrates the massive performance gain from vectorization. In machine learning, where you are constantly performing operations on large matrices and vectors, writing vectorized code is not just a best practice—it is a necessity.

Question 13

Explain the term “stride” in the context of NumPy arrays.

Theory

A **stride** in NumPy is a fundamental concept related to how an array's data is laid out in memory. The **strides** of an array are a tuple of integers that specify the number of **bytes** to step in each dimension when moving from one element to the next.

Understanding strides is key to understanding how NumPy can perform operations like transposing and slicing so efficiently without copying data.

How Strides Work

Let's consider an array:
arr = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.int32)

- The dtype is int32, so each element takes up **4 bytes** in memory.
- The data is stored in a contiguous block of memory like this: [1, 2, 3, 4, 5, 6]

The strides tell us how to navigate this block of memory:

- **arr.strides** will be (12, 4).
- **Stride for axis 1 (columns)**: The last number in the strides tuple is 4. This means to move from one column to the next (e.g., from element 1 to 2), we need to jump **4 bytes** forward in memory.
- **Stride for axis 0 (rows)**: The first number is 12. This means to move from one row to the next (e.g., from element 1 to 4), we need to jump **12 bytes** forward in memory (3 elements * 4 bytes/element).

The Power of Strides: Views vs. Copies

Strides explain why slicing creates a view and not a copy.

- If we take a slice s = arr[:, ::2], we are selecting all rows but every second column.
- NumPy doesn't create a new data block. Instead, it creates a new ndarray object s that points to the *same* data as arr but has different strides.
- The new strides for s would be (12, 8). The row stride is the same, but the column stride is now 8 bytes (2 elements * 4 bytes/element), telling NumPy to skip every other element when traversing the columns.

This ability to create a new "view" of the data just by changing the strides metadata is what makes many NumPy operations incredibly fast and memory-efficient. Transposing an array (arr.T) is another zero-copy operation that just swaps the values in the strides tuple.

Code Example

Generated python

```
    import numpy as np

# A 3x4 array with 8-byte elements (int64)
arr = np.arange(12, dtype=np.int64).reshape(3, 4)

print(f"Array:\n{arr}")
print(f"Itemsize: {arr.itemsize} bytes")
print(f"Strides: {arr.strides} bytes\n")
# Expected strides: (4 * 8, 8) -> (32, 8)

# Transposing the array
arr_t = arr.T
print(f"Transposed Array:\n{arr_t}")
print(f"Transposed Strides: {arr_t.strides} bytes\n")
# Transposing just swaps the strides: (8, 32)

# Slicing the array
arr_slice = arr[::2, ::3] # Every 2nd row, every 3rd column
print(f"Sliced Array:\n{arr_slice}")
print(f"Original Strides: {arr.strides}")
# New stride = original stride * step size
# Row stride = 32 * 2 = 64
# Col stride = 8 * 3 = 24
print(f"Sliced Strides: {arr_slice.strides} bytes")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code

```
    Array:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
Itemsize: 8 bytes
Strides: (32, 8) bytes

Transposed Array:
[[ 0  4  8]
```

```
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]
Transposed Strides: (8, 32) bytes

Sliced Array:
[[ 0  3]
 [ 8 11]]
Original Strides: (32, 8)
Sliced Strides: (64, 24) bytes
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Question 14

How does NumPy handle data types to optimize memory use?

Theory

NumPy's ability to handle data types with high precision is a key feature for optimizing memory usage. Unlike standard Python lists, which store pointers to full-fledged Python objects, a NumPy array is a contiguous block of memory where each element is of the **same, fixed-size data type (dtype)**.

This design allows for significant memory optimization through several mechanisms:

1. Fixed-Size, Specific Data Types

- **Concept**: NumPy provides a wide range of specific dtypes that correspond to fixed-size C data types. This includes integers and floats of different bit depths (int8, int16, int32, int64, float16, float32, float64).
- **Optimization**: By choosing the smallest dtype that can accurately represent your data, you can drastically reduce the memory footprint of your arrays.
    - If you know your data consists of integers between 0 and 255, storing them as np.uint8 (1 byte per element) uses **8 times less memory** than the default np.int64 (8 bytes per element).
    - In deep learning, it's common practice to use float32 instead of the default float64, which **halves the memory usage** of model weights and data batches with often negligible impact on accuracy.
-

2. Contiguous Memory Layout

- **Concept**: The elements of a NumPy array are stored next to each other in a continuous block of memory.
- **Optimization**: This eliminates the overhead of storing pointers for each element, which is what Python lists do. For an array of a million 64-bit integers, a Python list would require memory for one million pointers *plus* memory for one million integer objects. A NumPy array just needs the memory for the one million integers.

3. Structured Arrays

- **Concept**: For more complex, heterogeneous data (like data from a CSV file), NumPy provides **structured arrays**. A structured array allows you to define a dtype that is a compound of other dtypes, similar to a struct in C.
- **Optimization**: This allows you to store heterogeneous data in a single, contiguous memory block, which is much more efficient than using an array with dtype=object, which would fall back to storing pointers.

Code Example

This example shows the memory savings from choosing an appropriate dtype.

Generated python

```
    import numpy as np

# Let's create an array of one million integers that are all between 0 and 100.
num_elements = 1_000_000
data = np.random.randint(0, 101, size=num_elements)

# --- Case 1: Default dtype (likely int64 on a 64-bit system) ---
arr_default = np.array(data)
default_memory = arr_default.nbytes / (1024**2) # Convert bytes to megabytes
print(f"Default dtype: {arr_default.dtype}")
print(f"Memory usage with default dtype: {default_memory:.2f} MB\n")

# --- Case 2: Optimized dtype ---
# Since all values are between 0 and 255, we can safely use uint8 (1 byte).
arr_optimized = np.array(data, dtype=np.uint8)
optimized_memory = arr_optimized.nbytes / (1024**2)
print(f"Optimized dtype: {arr_optimized.dtype}")
print(f"Memory usage with optimized dtype: {optimized_memory:.2f} MB\n")

# --- Calculate Savings ---
if optimized_memory > 0:
    print(f"By choosing the correct dtype, we used {default_memory / optimized_memory:.0f}x
less memory.")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code
```
    Default dtype: int64
Memory usage with default dtype: 7.63 MB

Optimized dtype: uint8
Memory usage with optimized dtype: 0.95 MB

By choosing the correct dtype, we used 8x less memory.
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

This demonstrates that a conscious choice of dtype is a powerful and simple technique for dramatically reducing memory consumption when working with large numerical datasets.

Question 15

What are NumPy strides, and how do they affect array manipulation?

Theory

NumPy strides are a crucial piece of metadata for an ndarray that dictates how to navigate the array's data in memory. The strides are a tuple of integers where each integer specifies the number of **bytes** to jump in memory to get to the next element along a specific dimension (axis).

Understanding strides is key to understanding NumPy's efficiency. They are the mechanism that allows NumPy to perform many array manipulations, like **slicing, transposing, and reshaping**, without actually copying the underlying data. These operations often just create a new **view** of the data with different strides.

How They Affect Array Manipulation

Let's use a 3x4 array of int32 (4 bytes per element) as an example.
arr = np.arange(12, dtype=np.int32).reshape(3, 4)
The data in memory is a flat block: [0, 1, 2, ..., 11]
The strides for this array are (16, 4).

- To move to the next column (axis 1), jump 4 bytes.
- To move to the next row (axis 0), jump 16 bytes.

1. Slicing:

- **Manipulation**: slice_arr = arr[:, ::2] (all rows, every second column).
- **Effect on Strides**: NumPy creates a new view. The data is not copied. The new strides become (16, 8). The row stride is unchanged, but the column stride is now 4 bytes * 2 = 8 bytes. The new array object knows how to "skip" over elements in the original data buffer.

2. Transposing:

- **Manipulation**: transposed_arr = arr.T.
- **Effect on Strides**: This is a zero-copy operation. The new view transposed_arr has a shape of (4, 3) and its strides are simply the reverse of the original: (4, 16). It reinterprets the same memory block with different stepping rules.

3. Reshaping:

- **Manipulation**: reshaped_arr = arr.reshape(4, 3).
- **Effect on Strides**: If the reshape is compatible with the memory layout, NumPy can often create a view with new strides. For example, reshaping a C-contiguous (2, 6) array to (3, 4) would require a copy, but reshaping (2, 6) to (6, 2) can be done with strides if the array is Fortran-contiguous.

4. C-style vs. Fortran-style Contiguity:

- The strides also define the memory layout.
- **C-style (row-major order)**: The last dimension's stride is the smallest. This is the default in NumPy. strides=(16, 4).
- **Fortran-style (column-major order)**: The first dimension's stride is the smallest. strides=(4, 12).
- Knowing the contiguity can be important for performance, as accessing elements in their stored order is faster due to CPU caching.

In summary, strides are the internal bookkeeping mechanism that allows NumPy to be so flexible and efficient. They allow many array manipulations to be nearly instantaneous, memory-free operations that simply create a new metadata "header" (a new ndarray object) that points to the same underlying data but with different rules for traversing it.

Question 16

Explain the concept and use of masked arrays in NumPy.

Theory

A **masked array** is a special type of NumPy array that allows you to "mask" or ignore certain elements during computations. It is a combination of a standard ndarray with a **boolean mask** of the same shape.

- **The Array**: Contains the data, which may include invalid or missing values.
- **The Mask**: A boolean array where a value of True indicates that the corresponding element in the data array should be **masked** (i.e., treated as invalid or missing).

When you perform computations on a masked array (e.g., calculating the mean or sum), the masked elements are automatically excluded from the calculation.

Use Cases

Masked arrays are particularly useful in scientific and data analysis contexts where datasets often contain missing, invalid, or unreliable data points that should not be included in calculations.

1. **Handling Missing Data**: This is the primary use case. If your dataset uses a specific value (like -999) to denote a missing entry, you can create a mask to ignore these values without having to physically remove or change them.
2. **Excluding Outliers**: You can programmatically identify outliers and create a mask to exclude them from statistical calculations like mean and standard deviation, giving you a more robust estimate.
3. **Working with Sensor Data**: Sensor readings may sometimes be invalid. A masked array allows you to flag and ignore these readings in your analysis.
4. **Plotting**: When plotting data with gaps, a masked array can be used to prevent plotting libraries from connecting lines across the missing data points.

Code Example

Generated python

```
    import numpy as np
import numpy.ma as ma # The masked array module

# --- 1. Create a masked array ---
# Imagine a dataset with -999 as a missing value indicator
data = np.array([1, 2, -999, 4, 5, -999, 7])
```

```
# Create a masked array where -999 is masked
masked_arr = ma.masked_where(data == -999, data)

print(f"Original data: {data}")
print(f"Masked array: {masked_arr}")
print(f"Mask: {masked_arr.mask}")
print(f"Data part: {masked_arr.data}")

# --- 2. Perform computations ---
# Compare the mean of the original array vs. the masked array
mean_original = data.mean()
mean_masked = masked_arr.mean()

print(f"\nMean of original data (incorrect): {mean_original:.2f}")
print(f"Mean of masked array (correct): {mean_masked:.2f}")

# You can also set a fill value for display or conversion
masked_arr.set_fill_value(0)
print(f"\nMasked array with fill value: {masked_arr.filled()}")

# --- 3. Modifying the mask ---
# You can modify the mask directly
masked_arr.mask[1] = True # Mask the second element
print(f"\nArray after masking index 1: {masked_arr}")
print(f"New mean: {masked_arr.mean():.2f}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code
```
    Original data: [   1    2 -999    4    5 -999    7]
Masked array: [1 2 -- 4 5 -- 7]
Mask: [False False  True False False  True False]
Data part: [   1    2 -999    4    5 -999    7]

Mean of original data (incorrect): -282.86
Mean of masked array (correct): 3.80

Masked array with fill value: [1 2 0 4 5 0 7]

Array after masking index 1: [1 -- -- 4 5 -- 7]
```

New mean: 4.25

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

This example clearly shows how the masked array correctly calculates the mean by ignoring the masked -999 values, whereas the standard NumPy array produces a nonsensical result.

Question 17

What are the functions available for padding arrays in NumPy?

Theory

Padding is the process of adding extra values (padding) around the borders of a NumPy array. This is a very common operation in several areas, especially in **image processing**, **convolutional neural networks (CNNs)**, and **signal processing**.

The primary function in NumPy for this task is numpy.pad().

numpy.pad()

This is a highly versatile function that allows for a wide variety of padding methods.

Key Arguments:

- array: The input array to be padded.
- pad_width: A sequence of tuples specifying the number of values to pad before and after each axis. For a 2D array, ((top, bottom), (left, right)).
- mode: A string specifying the padding method. This is the most important argument.

Common Padding Modes

1. **constant**: Pads with a constant value.
   - **Use Case**: The most common mode. Often used to pad with zeros.
   - Requires an additional constant_values argument.
2.
3. **edge**: Pads with the edge values of the array.
   - **Use Case**: Useful when you want to extend the boundary of an image or signal without introducing artifacts from a constant value.
4.

5. **reflect**: Pads with the reflection of the vector mirrored on the first and last values of the vector along each axis.
    - **Use Case**: Often used in signal processing and image filtering to reduce boundary effects.
6.
7. **symmetric**: Pads with the reflection of the vector mirrored along the edge of the array. (Similar to reflect but the edge value is not repeated).
8. **wrap**: Pads with the wrap-around of the vector. The values from the beginning of the array are used to pad the end, and vice versa.
    - **Use Case**: For data that is periodic, like a signal representing a full rotation.
9.

Code Example

Generated python

```
    import numpy as np

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

print(f"Original array:\n{arr}\n")

# --- 1. Constant Padding ---
# Pad with 1 value on top, 2 on bottom, 1 on left, 2 on right
padded_constant = np.pad(arr, pad_width=((1, 2), (1, 2)), mode='constant', constant_values=0)
print(f"--- Constant Padding (with 0) ---\n{padded_constant}\n")

# --- 2. Edge Padding ---
padded_edge = np.pad(arr, pad_width=1, mode='edge') # A single value pads all sides
print(f"--- Edge Padding ---\n{padded_edge}\n")

# --- 3. Reflect Padding ---
padded_reflect = np.pad(arr, pad_width=1, mode='reflect')
print(f"--- Reflect Padding ---\n{padded_reflect}\n")

# --- 4. Symmetric Padding ---
padded_symmetric = np.pad(arr, pad_width=1, mode='symmetric')
print(f"--- Symmetric Padding ---\n{padded_symmetric}\n")

# --- 5. Wrap Padding ---
padded_wrap = np.pad(arr, pad_width=1, mode='wrap')
print(f"--- Wrap Padding ---\n{padded_wrap}\n")
```

IGNORE_WHEN_COPYING_START
content_copy download

Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code
```
    Original array:
[[1 2]
 [3 4]]

--- Constant Padding (with 0) ---
[[0 0 0 0 0]
 [0 1 2 0 0]
 [0 3 4 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]]

--- Edge Padding ---
[[1 1 2 2]
 [1 1 2 2]
 [3 3 4 4]
 [3 3 4 4]]

--- Reflect Padding ---
[[4 3 4 3]
 [2 1 2 1]
 [4 3 4 3]
 [2 1 2 1]]

--- Symmetric Padding ---
[[1 1 2 2]
 [1 1 2 2]
 [3 3 4 4]
 [3 3 4 4]]

--- Wrap Padding ---
[[4 3 4 3]
 [2 1 2 1]
 [4 3 4 3]
 [2 1 2 1]]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Question 18

Describe how you can use NumPy for simulating Monte Carlo experiments.

Theory

Monte Carlo simulation is a computational technique that uses repeated **random sampling** to obtain numerical results. It is used to model the probability of different outcomes in a process that cannot easily be predicted due to the intervention of random variables.

NumPy is the perfect tool for Monte Carlo simulations in Python because:

1. It has a powerful and fast **random number generation** module (numpy.random).
2. Its ability to perform **vectorized operations** allows you to run hundreds of thousands or millions of simulation trials very quickly.

A Classic Example: Estimating Pi (π)

A simple and intuitive Monte Carlo experiment is to estimate the value of Pi.

- **The Idea**:
    1. Imagine a square with a side length of 2, centered at the origin. Its area is 2 * 2 = 4.
    2. Inscribe a circle inside this square. The circle has a radius of 1, and its area is π * r^2 = π.
    3. The ratio of the circle's area to the square's area is π / 4.
    4. If we generate a large number of random points uniformly inside the square, the proportion of points that fall *inside the circle* should be approximately equal to this ratio of the areas.
    5. (Points in circle) / (Total points) ≈ π / 4
    6. Therefore, π ≈ 4 * (Points in circle) / (Total points)
-

Code Example

Generated python

```
    import numpy as np
import matplotlib.pyplot as plt

def estimate_pi(num_simulations=1000000):
   """
   Estimates the value of Pi using a Monte Carlo simulation.

   Args:
       num_simulations (int): The number of random points to generate.

   Returns:
       float: The estimated value of Pi.
   """
   # 1. Generate random points
   # Generate random x and y coordinates between -1 and 1
   # This is much faster than a Python for loop
   x_coords = np.random.uniform(-1, 1, num_simulations)
   y_coords = np.random.uniform(-1, 1, num_simulations)

   # 2. Calculate the distance from the origin for each point
   # The distance from origin is sqrt(x^2 + y^2)
   # A point is inside the circle if its distance <= 1,
   # or more simply, if x^2 + y^2 <= 1.
   distances = x_coords**2 + y_coords**2

   # 3. Count the points inside the circle
   # Use boolean indexing to find points where the condition is met
   points_in_circle = np.sum(distances <= 1)

   # 4. Calculate Pi
   pi_estimate = 4 * points_in_circle / num_simulations

   return pi_estimate, x_coords, y_coords, distances

# --- Run the simulation ---
num_trials = 500000
pi_value, x, y, dist = estimate_pi(num_trials)

print(f"Number of trials: {num_trials}")
print(f"Estimated value of Pi: {pi_value}")
print(f"Actual value of Pi: {np.pi}")

# --- Visualize the results (for a smaller number of points) ---
num_viz_points = 5000
in_circle = dist[:num_viz_points] <= 1
out_circle = dist[:num_viz_points] > 1

plt.figure(figsize=(7, 7))
plt.scatter(x[:num_viz_points][in_circle], y[:num_viz_points][in_circle], color='blue', s=1,
label='Inside Circle')
plt.scatter(x[:num_viz_points][out_circle], y[:num_viz_points][out_circle], color='red', s=1,
label='Outside Circle')
```

```
plt.title('Monte Carlo Estimation of Pi')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.gca().set_aspect('equal', adjustable='box')
plt.show()
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Explanation of NumPy's Role

- **np.random.uniform**: This function generates a large array of random numbers in a single, fast operation. This is the "random sampling" part of the simulation.
- **Vectorized Arithmetic**: The calculation x_coords**2 + y_coords**2 is a fully vectorized operation that computes the squared distance for all points simultaneously.
- **Boolean Indexing and np.sum**: The line np.sum(distances <= 1) is a powerful and concise way to count the number of True values in the boolean mask distances <= 1. This is much faster than iterating with a counter.

This example shows how NumPy's features enable the efficient execution of millions of simulation trials, which is the essence of the Monte Carlo method.

Question 19

Explain how to resolve the MemoryError when working with very large arrays in NumPy.

Theory

A MemoryError in NumPy occurs when you try to create an array that is larger than the available RAM on your machine. This is a common problem when working with "big data" in machine learning.

Resolving this requires moving beyond in-memory processing and adopting strategies that either reduce the memory footprint of the data or process the data in smaller chunks.

Strategies to Resolve MemoryError

1. **Check and Optimize Data Types (dtype)**:
    - **Strategy**: This should always be the first step. The default dtype for integers is often int64 (8 bytes) and for floats is float64 (8 bytes). Your data may not require this much precision.

- **Action**: Explicitly downcast the dtype to the smallest possible type that can still represent your data accurately.
  - Use np.uint8 (1 byte) for values between 0-255.
  - Use np.float32 (4 bytes) instead of np.float64 for most machine learning tasks.
-
- **Impact**: This can reduce memory usage by 2x to 8x with a single line of code.

2.
3. **Process Data in Batches or Chunks**:
   - **Strategy**: Instead of loading the entire dataset into a single giant NumPy array, process it in smaller, manageable chunks that can fit into memory.
   - **Action**: When reading a large file (e.g., a CSV), use the chunksize parameter in pandas.read_csv. This returns an iterator that yields DataFrames of the specified chunk size. You can then process each chunk individually.

Generated python

```
    import pandas as pd
chunk_iter = pd.read_csv('very_large_file.csv', chunksize=100000)
for chunk in chunk_iter:
   # Process this chunk (which is a smaller DataFrame)
   process(chunk)
```

4.
   IGNORE_WHEN_COPYING_START
   content_copy download
   Use code with caution. Python
   IGNORE_WHEN_COPYING_END
5. **Use Memory-Mapped Arrays (np.memmap)**:
   - **Strategy**: A memory-mapped file is an array that is stored on disk but can be accessed and sliced as if it were an in-memory NumPy array.
   - **Action**: Use np.memmap to create an array on disk. The operating system handles the magic of loading only the necessary parts of the file into RAM when they are accessed.
   - **Use Case**: This is ideal when you need to perform random access (slicing) on an array that is too large for RAM.
6.
7. **Use Out-of-Core Computing Libraries (Dask)**:
   - **Strategy**: For more complex operations that need to be performed on the entire large dataset, use a library designed for out-of-core computing like **Dask**.
   - **Action**: Dask provides dask.array, which mimics the NumPy API. Dask breaks the large array into many smaller NumPy arrays (chunks) and creates a task graph of the computations. It can then execute these tasks in parallel, intelligently loading only the necessary chunks into memory at any given time.
   - **Benefit**: This allows you to perform complex computations on datasets that are orders of magnitude larger than your available RAM.

8.
9. **Use a More Powerful Machine or a Distributed System**:
   - **Strategy**: If the problem is simply too large for a single machine, the solution is to scale up or scale out.
   - **Action**:
     - **Scale Up**: Run the code on a machine with more RAM (e.g., a high-memory instance on a cloud provider).
     - **Scale Out**: Use a distributed computing framework like **Apache Spark (PySpark)**. Spark distributes the array (as an RDD or DataFrame) across a cluster of machines, allowing you to leverage the combined memory and CPU power of the entire cluster.
   -
10.

The best strategy depends on the nature of the data and the required computations, but starting with dtype optimization and moving to chunking or Dask are the most common and effective approaches.

Question 20

What are NumPy “polynomial” objects and how are they used?

Theory

The numpy.polynomial module provides a powerful and convenient way to work with polynomial functions in a stable and efficient manner. It allows you to create polynomial objects that can be manipulated much like any other numerical value.

Instead of representing a polynomial as a simple array of coefficients, this module provides classes that encapsulate the polynomial and its properties, offering methods for common operations like evaluation, differentiation, integration, and finding roots.

Key Classes in numpy.polynomial

The module provides classes for different polynomial bases, which have different numerical properties. The most common is:

- **Polynomial**: Represents a polynomial in the standard power series basis: $P(x) = c_0 + c_1x + c_2x^2 + ...$

How They Are Used

1. **Creating a Polynomial**:

   - You create a polynomial object by passing it a list or array of its coefficients, typically starting with the constant term.
2.
3. **Evaluation**:
   - Once created, you can evaluate the polynomial at a single point or on an entire NumPy array of points just by calling the object like a function.
4.
5. **Arithmetic**:
   - You can perform standard arithmetic operations (+, -, *, //, **) directly on the polynomial objects.
6.
7. **Calculus**:
   - The objects have built-in methods for calculus:
     - .deriv(): Returns a new polynomial object representing the derivative.
     - .integ(): Returns the integral.
   -
8.
9. **Root Finding**:
   - The .roots() method calculates all the roots (real and complex) of the polynomial.
10.
11. **Curve Fitting**:
    - The class method .fit(x, y, deg) is a powerful feature. It finds the polynomial of a specified degree (deg) that is the best least-squares fit for a given set of data points (x, y).
12.

Code Example

Generated python

```
    import numpy as np
import matplotlib.pyplot as plt
from numpy.polynomial import Polynomial

# --- 1. Creating and Evaluating a Polynomial ---
# Represents P(x) = 1 + 2x + 3x^2
p = Polynomial([1, 2, 3])

print(f"Polynomial p(x): \n{p}\n")
print(f"p(2) = {p(2)}") # Evaluate at x=2 -> 1 + 2*2 + 3*(2^2) = 17
print(f"p([0, 1, 2]) = {p([0, 1, 2])}") # Evaluate on an array

# --- 2. Calculus and Roots ---
p_deriv = p.deriv()
print(f"\nDerivative p'(x): \n{p_deriv}")
print(f"Roots of p(x): {p.roots()}")
```

```
# --- 3. Polynomial Fitting ---
# Create some sample data with a quadratic relationship + noise
np.random.seed(42)
x_data = np.linspace(-1, 1, 50)
y_data = 1 + 2*x_data + 3*x_data**2 + np.random.randn(50) * 0.2

# Fit a 2nd degree polynomial to the data
p_fit = Polynomial.fit(x_data, y_data, deg=2)
print(f"\nFitted polynomial:\n{p_fit}")
print(f"Fitted coefficients are close to [1, 2, 3]: {np.round(p_fit.convert().coef, 2)}")

# --- Visualization of the fit ---
plt.figure(figsize=(8, 6))
plt.scatter(x_data, y_data, label='Noisy Data')
plt.plot(x_data, p_fit(x_data), color='red', label='Polynomial Fit')
plt.title('Polynomial Curve Fitting')
plt.legend()
plt.grid(True)
plt.show()
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Why Use This Module?

Using the numpy.polynomial module is generally preferred over using a simple coefficient array (like with np.polyfit or np.poly1d) because the classes are designed to be more **numerically stable**, especially when dealing with high-degree polynomials. They provide a cleaner, object-oriented interface for a wider range of polynomial operations.

Question 21

How does the internal C-API contribute to NumPy's performance?

Theory

The high performance of NumPy is not due to clever Python code; it is due to the fact that the core of the library, including the ndarray object and its operations, is implemented in **C and Fortran**. The **NumPy C Application Programming Interface (API)** is the bridge that allows this powerful, low-level C code to be controlled by high-level, easy-to-use Python code.

The C-API plays a crucial role in several aspects of NumPy's performance.

Key Contributions of the C-API

1. **Direct Memory Manipulation**:
    - The C-API allows for the direct creation and manipulation of ndarrays in C. The data is stored in a contiguous C-level array in memory.
    - This avoids the overhead of Python objects. A Python integer is a complex object with a reference count and type information. A C integer is just a raw, 4-byte block of memory. By working with these raw types in C, operations can be performed much more efficiently.
2.
3. **Enabling Vectorized Operations (Ufuncs)**:
    - When you call a universal function (ufunc) in Python like np.add(a, b), the Python interpreter hands off the ndarray objects to the underlying C implementation of the ufunc.
    - The C function then performs a highly optimized, low-level loop directly on the raw data buffers of the arrays. This loop is compiled and can take advantage of low-level CPU optimizations (like SIMD instructions) that are inaccessible from pure Python.
    - This is the essence of **vectorization**: a single Python call triggers a fast, compiled C loop.
4.
5. **Releasing the Global Interpreter Lock (GIL)**:
    - The GIL is a mechanism in CPython that prevents multiple native threads from executing Python bytecode at the same time.
    - The NumPy C-API allows its C functions to **release the GIL** before executing a long-running, computationally intensive operation (like a large matrix multiplication).
    - **Impact**: While the C code is busy computing, other Python threads can run. Furthermore, the underlying numerical libraries that NumPy links to (like OpenBLAS or MKL) can be multi-threaded themselves, allowing a single NumPy operation to use all the cores of a CPU. This is a critical feature for parallelism.
6.
7. **Interoperability with Other C/Fortran Libraries**:
    - The C-API defines a standard way to represent multi-dimensional arrays in memory. This allows NumPy to act as a **lingua franca** for numerical data.
    - Other scientific libraries written in C, C++, or Fortran (like SciPy, Scikit-learn, TensorFlow) can use the C-API to directly access and operate on NumPy array data without any need for slow, intermediate data conversion steps. This seamless, zero-copy interoperability is fundamental to the performance of the entire scientific Python ecosystem.
8.

In summary, the C-API is the essential layer that connects the user-friendly Python interface of NumPy to its high-performance C/Fortran engine. It enables direct memory access, fast vectorized loops, parallelism by releasing the GIL, and seamless integration with other compiled libraries.

Question 22

Explain the concept of a stride trick in NumPy.

Theory

The **stride trick** is an advanced NumPy technique that involves manually manipulating the **strides** of an array to create a new array view without copying any data. It is a powerful but potentially dangerous method for creating complex array views, such as a rolling window of an array.

The "trick" is to create a new ndarray object that points to the same underlying data as an existing array but has a custom shape and stride configuration. This can create the illusion of a much larger array while using no additional memory.

The Core Function: as_strided

The main tool for this is numpy.lib.stride_tricks.as_strided.

as_strided(x, shape=None, strides=None)

- x: The input array.
- shape: The desired shape of the new array view.
- strides: The desired strides for the new array view.

Use Case: Creating a Rolling Window

This is the classic example. A rolling window view of a 1D array is a 2D array where each row is a "window" of consecutive elements from the original array.

Let's say we have an array a = [0, 1, 2, 3, 4, 5] and we want a rolling window of size 3. The desired output would be:

Generated code
```
  [[0, 1, 2],
 [1, 2, 3],
 [2, 3, 4],
 [3, 4, 5]]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Notice that the elements overlap. A naive implementation would copy the data for each row, using extra memory. The stride trick can create this 2D view with zero memory copy.

Code Example

Generated python

```
    import numpy as np
from numpy.lib.stride_tricks import as_strided

# Our original 1D array
a = np.arange(10, dtype=np.int32)
itemsize = a.itemsize # 4 bytes

print(f"Original array: {a}")
print(f"Original shape: {a.shape}")
print(f"Original strides: {a.strides}\n")

# --- Create a rolling window view ---
window_size = 4
num_windows = len(a) - window_size + 1

# The new shape will be (number_of_windows, window_size)
new_shape = (num_windows, window_size)

# The new strides:
# To move to the next row (e.g., from window [0,1,2,3] to [1,2,3,4]),
# we just need to move forward by one element's size in memory.
# To move to the next column within a window, we also move by one element's size.
new_strides = (itemsize, itemsize)

rolling_view = as_strided(a, shape=new_shape, strides=new_strides)

print(f"Rolling window view (shape {rolling_view.shape}):\n{rolling_view}\n")
print(f"Strides of the view: {rolling_view.strides}")

# --- Demonstrate that it's a view ---
print("--- Modifying the view affects the original ---")
rolling_view[0, 1] = 99
print(f"View after modification:\n{rolling_view}")
print(f"Original array after modification: {a}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code
```
    Original array: [0 1 2 3 4 5 6 7 8 9]
Original shape: (10,)
Original strides: (4,)

Rolling window view (shape (7, 4)):
[[0 1 2 3]
 [1 2 3 4]
 [2 3 4 5]
 [3 4 5 6]
 [4 5 6 7]
 [5 6 7 8]
 [6 7 8 9]]

Strides of the view: (4, 4)
--- Modifying the view affects the original ---
View after modification:
[[ 0 99  2  3]
 [99  2  3  4]
 [ 2  3  4  5]
 [ 3  4  5  6]
 [ 4  5  6  7]
 [ 5  6  7  8]
 [ 6  7  8  9]]
Original array after modification: [ 0 99  2  3  4  5  6  7  8  9]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Notice in the output that changing element 1 in the first window (rolling_view[0, 1]) also changed element 0 in the second window (rolling_view[1, 0]), because they are both pointing to the exact same location in the original array's memory.

Dangers and Warnings

- The stride trick is powerful but dangerous. It is very easy to create an invalid view that points to memory outside the original array's bounds, which can corrupt memory or cause the program to crash.
- The resulting array must not be written to if any of its elements overlap in memory, as this can lead to unexpected behavior (as shown in the example). It's best to treat strided views as read-only.
- In modern NumPy, there are often safer, higher-level functions available. For rolling windows, np.lib.stride_tricks.sliding_window_view is now the recommended and safer alternative.

Question 23

What is the role of the NumPy nditer object?

Theory

The nditer object, short for "n-dimensional iterator," is an efficient, multi-dimensional iterator object provided by NumPy. It is designed to provide a systematic and flexible way to iterate over the elements of one or more arrays.

While standard Python loops (for x in my_array) or multi-dimensional loops (for row in my_array: for x in row: ...) are easy to write, nditer provides a more powerful and optimized interface that is closer to how NumPy's internal C loops work.

Key Roles and Features

1. **Efficient Multi-Dimensional Iteration**:
    - nditer provides a way to visit every element of an N-dimensional array using a single loop, abstracting away the complexity of nested loops. The iteration order is chosen to be as efficient as possible, respecting the memory layout of the array (C-order vs. Fortran-order) to improve cache performance.
2.
3. **Iterating Over Multiple Arrays Simultaneously**:
    - This is one of its most powerful features. nditer can iterate over several arrays at once. It automatically handles **broadcasting** rules, making it easy to write custom functions that operate on multiple arrays in a vectorized fashion.
4.
5. **Modifying Arrays In-Place**:
    - By default, the iterator provides read-only access to the arrays. However, you can configure it with flags like 'readwrite' to allow for in-place modification of the array elements during iteration.
6.
7. **Controlling Iteration Order and Data Types**:

    - You can specify the iteration order (e.g., 'C', 'F') to optimize for memory access patterns.
    - You can specify a casting rule and an op_dtypes argument to control how the data is buffered and converted to a specific data type during iteration.
8.

Code Example

Generated python

```
    import numpy as np

# --- 1. Basic Iteration ---
a = np.arange(6).reshape(2, 3)

print("--- Basic Iteration ---")
with np.nditer(a) as it:
    for x in it:
        print(x, end=" ") # Iterates in C-order by default
print("\n")


# --- 2. Iterating over Multiple Arrays (with Broadcasting) ---
a = np.arange(6).reshape(2, 3) # Shape (2, 3)
b = np.array([10, 20, 30])    # Shape (3,)

print("--- Multi-Array Iteration ---")
# b will be broadcast to match a's shape
with np.nditer([a, b]) as it:
    for x, y in it:
        print(f"({x}, {y})", end=" ")
print("\n")


# --- 3. Modifying an Array In-Place ---
a = np.arange(6).reshape(2, 3)
print("\n--- In-place Modification ---")
print(f"Original a:\n{a}")

# The op_flags=['readwrite'] is required for modification
with np.nditer(a, op_flags=['readwrite']) as it:
    for x in it:
        x[...] = x * 2 # The [...] is required to modify the element in place

print(f"Modified a:\n{a}\n")
```

```
# --- 4. Controlling Iteration Order ---
a_f = np.arange(6).reshape(2, 3).T # A Fortran-contiguous array
print("--- Iteration Order ---")
print("Iterating a C-style array:")
with np.nditer(a) as it:
    for x in it:
        print(x, end=" ")
print("\nIterating a Fortran-style array:")
with np.nditer(a_f) as it:
    for x in it:
        print(x, end=" ")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

When to Use It

- You should use nditer when you need to write a custom, complex function in Python that needs to iterate over one or more arrays and the standard vectorized ufuncs are not sufficient.
- It provides a bridge between high-level Python and the low-level iteration logic of NumPy's C backend.
- It is particularly useful for authors of new ufuncs or other low-level numerical algorithms. For most everyday tasks, standard vectorized operations are preferred for their simplicity and performance.

Question 24

Explain how NumPy integrates with other Python libraries like Pandas and Matplotlib.

Theory

NumPy's ndarray is the **lingua franca** (common language) for numerical data in the Python scientific computing ecosystem. Its efficiency and standardized representation of data make it the foundational layer upon which many other libraries are built. This seamless integration is a key reason for Python's dominance in data science.

Integration with Pandas

- **Foundation**: Pandas is built directly on top of NumPy. A Pandas DataFrame is essentially a collection of Series objects, and each Series is internally backed by a NumPy ndarray.
- **How it Works**:
  - When you create a DataFrame with numerical data, the data for each column is stored in a NumPy array. This is why numerical operations on Pandas columns are so fast—they are actually NumPy vectorized operations.
  - You can access the underlying NumPy array of a DataFrame or Series at any time using the .values attribute (or .to_numpy() in modern Pandas).
  - You can easily create a DataFrame from a NumPy array.
-
- **Benefit**: This tight integration allows you to seamlessly switch between the two libraries. You can use Pandas for its powerful data manipulation, cleaning, and indexing capabilities, and then easily extract the data into a NumPy array to feed into a machine learning model from Scikit-learn, which expects NumPy arrays as input.

Integration with Matplotlib

- **Foundation**: Matplotlib, the primary plotting library, is designed to accept NumPy arrays as its main input format.
- **How it Works**:
  - Most plotting functions in Matplotlib, such as plt.plot(), plt.scatter(), and plt.imshow(), are optimized to work directly with NumPy arrays.
  - When you pass a NumPy array to a plotting function, Matplotlib can efficiently process the data and render the visualization.
-
- **Benefit**: This allows for a very natural workflow: perform your numerical computations and analysis in NumPy, and then directly pass the resulting arrays to Matplotlib to visualize your findings without any need for data conversion.

Code Example

Generated python

```
    import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# --- 1. Create data with NumPy ---
# NumPy is used for the core numerical data
X_np = np.linspace(0, 10, 100).reshape(-1, 1)
y_np = 2 * X_np.flatten() + 1 + np.random.randn(100) * 2

# --- 2. Manipulate data with Pandas ---
# We can easily convert the NumPy arrays into a Pandas DataFrame for analysis
df = pd.DataFrame({'feature': X_np.flatten(), 'target': y_np})
```

```
# Use Pandas for convenient data inspection
print("--- Data in Pandas DataFrame ---")
print(df.head())
print(df.describe())

# --- 3. Use with Scikit-learn ---
# Scikit-learn models expect NumPy arrays (or pandas DataFrames, which are converted
internally)
model = LinearRegression()
# We can pass the NumPy arrays directly
model.fit(X_np, y_np)
predictions = model.predict(X_np)

# --- 4. Visualize with Matplotlib ---
# Matplotlib works directly with the NumPy arrays
plt.figure(figsize=(8, 6))
plt.scatter(X_np, y_np, label='Original Data')
plt.plot(X_np, predictions, color='red', label='Linear Regression Fit')
plt.title('NumPy -> Pandas -> Scikit-learn -> Matplotlib Workflow')
plt.xlabel('Feature')
plt.ylabel('Target')
plt.legend()
plt.grid(True)
plt.show()
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

This example demonstrates the seamless workflow: NumPy creates the numerical data, Pandas provides a convenient structure for its analysis, Scikit-learn trains a model on it, and Matplotlib visualizes the results. This interoperability is possible because the ndarray serves as the common, underlying data format.

Question 25

Describe how NumPy can be used with JAX for accelerated machine learning computation.

Theory

JAX is a high-performance numerical computing library from Google that is rapidly gaining popularity in the machine learning research community. It combines a familiar, NumPy-like API with a powerful set of transformations that enable modern machine learning capabilities.

The integration between NumPy and JAX is incredibly tight. In fact, JAX's main interface, jax.numpy, is a **re-implementation of the NumPy API**.

Key JAX Transformations

JAX provides several key function transformations that can be applied to standard Python and jax.numpy code:

1. **jit (Just-In-Time Compilation)**:
    - **Concept**: You can use jax.jit as a decorator or a function to compile your Python code using XLA (Accelerated Linear Algebra), a domain-specific compiler for linear algebra.
    - **Benefit**: This can dramatically speed up your code by fusing multiple operations into a single optimized kernel that can run efficiently on accelerators like **GPUs and TPUs**.
2.
3. **grad (Automatic Differentiation)**:
    - **Concept**: jax.grad can automatically compute the gradient of any Python function.
    - **Benefit**: This is the foundation of modern deep learning. It allows you to define a loss function and then automatically get the function that calculates its gradients with respect to the model parameters, which is exactly what is needed for gradient descent.
4.
5. **vmap (Automatic Vectorization)**:
    - **Concept**: jax.vmap (vectorizing map) is a powerful transformation that can automatically convert a function designed to work on a single data point into a function that works on a whole batch of data points in a parallel, efficient manner.
    - **Benefit**: It allows you to write clean, simple code for a single example, and vmap handles the complexity of batching it efficiently.
6.

How NumPy and JAX Work Together

- **Familiar API**: A developer who knows NumPy can be productive with JAX almost immediately. The API is intentionally designed to be a near drop-in replacement. You can often take existing NumPy code and, with minimal changes (like changing import numpy as np to import jax.numpy as jnp), make it runnable with JAX.
- **Accelerator-Ready**: Code written with jax.numpy can be run seamlessly on CPUs, GPUs, or TPUs. JAX handles the details of moving the data and computations to the accelerator.

Code Example

This example shows how to take a simple prediction function written with the NumPy API, and then use JAX's transformations to compile it, get its gradient, and vectorize it.

Generated python
```
    import jax
import jax.numpy as jnp
import numpy as np

# Use JAX to set the default device (e.g., a GPU if available)
# jax.config.update('jax_platform_name', 'gpu')

# A simple linear regression prediction function using the jax.numpy API
def predict(params, inputs):
    return jnp.dot(inputs, params['w']) + params['b']

# --- 1. Use `jit` for compilation ---
# Create a compiled version of the predict function
fast_predict = jax.jit(predict)

# Some dummy data
key = jax.random.PRNGKey(0)
params = {'w': jax.random.normal(key, (3,)), 'b': 1.0}
inputs = jax.random.normal(key, (10, 3))

# Running the fast_predict function will be much faster on subsequent calls
prediction = fast_predict(params, inputs)
print(f"--- JIT Compilation ---")
print(f"Prediction result shape: {prediction.shape}\n")


# --- 2. Use `grad` for automatic differentiation ---
# Let's define a loss function
def mse_loss(params, inputs, targets):
    preds = predict(params, inputs)
    return jnp.mean((preds - targets)**2)

# Get the function that calculates the gradient of the loss
# argnums=0 means we want the gradient with respect to the first argument (params)
gradient_fn = jax.grad(mse_loss, argnums=0)

# Calculate the gradients
targets = jax.random.normal(key, (10,))
gradients = gradient_fn(params, inputs, targets)
```

```
print(f"--- Automatic Differentiation ---")
print(f"Gradients for weights 'w': {gradients['w']}")
print(f"Gradient for bias 'b': {gradients['b']}\n")


# --- 3. Use `vmap` for automatic vectorization ---
# Let's say we wrote our predict function to only work on a single input
def predict_single(params, single_input):
    return jnp.dot(single_input, params['w']) + params['b']

# Use vmap to create a new function that can handle a batch of inputs
# in_axes=(None, 0) means: don't map over the `params`, but map over the first axis of `inputs`
batched_predict = jax.vmap(predict_single, in_axes=(None, 0))

# Now we can call it on our batch of inputs
batch_predictions = batched_predict(params, inputs)
print(f"--- Automatic Vectorization ---")
print(f"Batched prediction result shape: {batch_predictions.shape}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

This example shows how JAX takes the familiar NumPy paradigm and enhances it with powerful, composable function transformations (jit, grad, vmap) that are essential for modern, high-performance machine learning research and development.

Question 1

How do you inspect the shape and size of a NumPy array?

Theory

Inspecting the shape and size of a NumPy array is a fundamental operation, crucial for debugging, understanding your data's dimensions, and ensuring compatibility between arrays for operations like matrix multiplication. NumPy provides several straightforward attributes for this purpose.

Key Attributes

Let arr be a NumPy ndarray.

1. **arr.shape**:

    - **What it is**: A **tuple** of integers representing the dimensions of the array. The length of the tuple is the number of dimensions (ndim).
    - **Example**: For a 2D array (matrix) with 3 rows and 5 columns, arr.shape will be (3, 5).
2.
3. **arr.size**:
    - **What it is**: An **integer** representing the total number of elements in the array.
    - **Example**: For the (3, 5) array, arr.size will be 3 * 5 = 15.
4.
5. **arr.ndim**:
    - **What it is**: An **integer** representing the number of axes or dimensions of the array.
    - **Example**: For the (3, 5) array, arr.ndim will be 2. For a vector, it would be 1.
6.

Code Example

Generated python

```
    import numpy as np

# Create a 3D array
# (2 layers, 3 rows, 4 columns)
arr_3d = np.arange(24).reshape(2, 3, 4)

print("--- Array Information ---")
print(f"The array:\n{arr_3d}\n")

# --- Inspecting the attributes ---

# 1. Shape
shape_tuple = arr_3d.shape
print(f"Shape of the array (arr.shape): {shape_tuple}")
print(f" - Number of layers: {shape_tuple[0]}")
print(f" - Number of rows: {shape_tuple[1]}")
print(f" - Number of columns: {shape_tuple[2]}")

# 2. Size
total_elements = arr_3d.size
print(f"\nTotal number of elements (arr.size): {total_elements}")

# 3. Number of dimensions
num_dimensions = arr_3d.ndim
print(f"\nNumber of dimensions (arr.ndim): {num_dimensions}")

# Verification
# size should be the product of the shape elements
assert total_elements == np.prod(shape_tuple)
# ndim should be the length of the shape tuple
assert num_dimensions == len(shape_tuple)
```

Output:

```
Generated code
    --- Array Information ---
The array:
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]

Shape of the array (arr.shape): (2, 3, 4)
 - Number of layers: 2
 - Number of rows: 3
 - Number of columns: 4

Total number of elements (arr.size): 24

Number of dimensions (arr.ndim): 3
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Question 2

How do you perform element-wise operations in NumPy?

Theory

Element-wise operations are operations that are applied independently to each element in an array or between corresponding elements of multiple arrays. This is a core concept of **vectorization** in NumPy and is the reason for its high performance.

Instead of writing a Python for loop, you can use standard arithmetic operators or NumPy's universal functions (ufuncs) to perform these operations on entire arrays at once.

Key Methods

1. **Using Standard Arithmetic Operators**:
    - The standard Python arithmetic operators (+, -, *, /, **, etc.) are overloaded in NumPy to perform element-wise operations.
    - These operations support **broadcasting**, allowing you to combine arrays of different but compatible shapes.
2.
3. **Using NumPy Universal Functions (ufuncs)**:
    - NumPy provides a rich library of ufuncs that perform element-wise operations. For every arithmetic operator, there is a corresponding ufunc (e.g., + corresponds to np.add()).
    - This category also includes a vast range of other functions like np.sqrt(), np.exp(), np.sin(), and np.log().
4.

Code Example

Generated python

```
    import numpy as np

# Create two arrays of the same shape
a = np.array([[1, 2],
              [3, 4]])
b = np.array([[10, 20],
              [30, 40]])

print(f"Array a:\n{a}")
print(f"Array b:\n{b}\n")

# --- Element-wise operations with operators ---
print("--- Using Operators ---")
# Element-wise addition
add_result = a + b
print(f"a + b:\n{add_result}\n")

# Element-wise multiplication
mul_result = a * b
print(f"a * b:\n{mul_result}\n")

# --- Element-wise operations with ufuncs ---
print("--- Using Ufuncs ---")
# Element-wise subtraction using the ufunc
```

```
sub_result = np.subtract(b, a)
print(f"np.subtract(b, a):\n{sub_result}\n")

# Element-wise square root on a single array
sqrt_result = np.sqrt(a)
print(f"np.sqrt(a):\n{sqrt_result}\n")

# --- Operations with a scalar ---
# The scalar is broadcast to every element of the array
scalar_result = a * 10
print("--- Operation with a Scalar ---")
print(f"a * 10:\n{scalar_result}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code [with caution](). Python
IGNORE_WHEN_COPYING_END

Output:

Generated code

```
    Array a:
[[1 2]
 [3 4]]
Array b:
[[10 20]
 [30 40]]

--- Using Operators ---
a + b:
[[11 22]
 [33 44]]

a * b:
[[ 10  40]
 [ 90 160]]

--- Using Ufuncs ---
np.subtract(b, a):
[[ 9 18]
 [27 36]]

np.sqrt(a):
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

```
--- Operation with a Scalar ---
a * 10:
[[10 20]
 [30 40]]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

This demonstrates how simple and readable it is to perform complex mathematical operations on entire datasets without writing a single explicit loop in Python.

Question 3

How do you compute the mean, median, and standard deviation with NumPy?

Theory

NumPy provides simple, highly optimized functions for computing the most common descriptive statistics of an array. These functions can be called either as top-level NumPy functions (e.g., np.mean(arr)) or as methods of the ndarray object itself (e.g., arr.mean()).

These functions can also be applied along a specific **axis** of a multi-dimensional array to compute the statistic for each row or column.

Key Functions

1. **Mean (np.mean() or .mean())**:
    - **Description**: Calculates the arithmetic average of the elements.
2.
3. **Median (np.median())**:
    - **Description**: Calculates the median (the middle value) of the elements. The median is a more robust measure of central tendency than the mean, as it is not affected by outliers.
4.
5. **Standard Deviation (np.std() or .std())**:
    - **Description**: Calculates the standard deviation, which is a measure of the amount of variation or dispersion of the data. It is the square root of the variance.
6.

Code Example

Generated python
```
    import numpy as np

# Create a 2D array
data = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 50]]) # Added an outlier (50)

print(f"Data:\n{data}\n")

# --- 1. Compute statistics for the entire array (flattened) ---
print("--- Overall Statistics ---")
mean_total = np.mean(data)
median_total = np.median(data)
std_total = np.std(data)

print(f"Mean: {mean_total:.2f}")
print(f"Median: {median_total:.2f}  <-- Note: Median is less affected by the outlier 50")
print(f"Standard Deviation: {std_total:.2f}\n")

# --- 2. Compute statistics along a specific axis ---
print("--- Axis-wise Statistics ---")
# axis=0 -> compute down the columns
mean_cols = data.mean(axis=0)
print(f"Mean of each column (axis=0): {np.round(mean_cols, 2)}")

# axis=1 -> compute across the rows
median_rows = np.median(data, axis=1)
print(f"Median of each row (axis=1): {median_rows}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code
```
    Data:
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 50]]

--- Overall Statistics ---
Mean: 9.67
```

```
Median: 6.50  <-- Note: Median is less affected by the outlier 50
Standard Deviation: 12.37

--- Axis-wise Statistics ---
Mean of each column (axis=0): [ 5.   6.   7.  20.67]
Median of each row (axis=1): [ 2.5  6.5 10.5]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Interpretation of the Axis-wise Results

- **mean(axis=0)**: This collapses the rows and calculates the mean for each of the 4 columns independently, resulting in a 1D array of 4 values. This is a very common operation in machine learning for calculating the mean of each feature.
- **median(axis=1)**: This collapses the columns and calculates the median for each of the 3 rows independently, resulting in a 1D array of 3 values.

Question 4

Why is NumPy more efficient for numerical computations than pure Python?

Theory

NumPy's efficiency for numerical computations stems from a combination of its low-level implementation, its memory layout, and its support for vectorized operations. It effectively bridges the gap between Python's ease of use and the raw speed of compiled languages like C and Fortran.

Key Reasons for Efficiency

1. **Implementation in C and Fortran**:
    - **Python**: Python is an interpreted language, which means there is a lot of overhead in executing code line by line.
    - **NumPy**: The core of NumPy, including the ndarray object and its operations (ufuncs), is written in highly optimized, compiled C and Fortran code. When you perform a NumPy operation, the heavy computation is handed off to this fast, low-level code.
2.
3. **Vectorization**:

    - **Python**: To perform an operation on a collection of numbers, you typically need to use an explicit for loop. Each iteration of this loop involves overhead from the Python interpreter.
    - **NumPy**: NumPy allows you to perform operations on entire arrays at once. A single Python statement like a + b triggers a single, highly optimized loop in C that iterates over the data. This is known as **vectorization**.
4.
5. **Contiguous Memory Layout**:
    - **Python Lists**: A list stores a collection of pointers to Python objects that can be scattered all over memory. Accessing the data requires an extra step of following these pointers.
    - **NumPy Arrays**: An ndarray is a homogeneous data structure stored in a **contiguous block of memory**. This is crucial for performance. When the data is contiguous, the computer's CPU can make effective use of its **memory cache**, which dramatically speeds up data access. This layout is also ideal for passing data to low-level C or Fortran libraries.
6.
7. **Homogeneous Data Type**:
    - **Python Lists**: Can contain objects of different types. This flexibility comes at a cost: for every operation, Python has to perform a type check.
    - **NumPy Arrays**: Are homogeneous (all elements are of the same dtype). Since the type is known for the entire array, NumPy can use optimized, specialized C functions for that specific data type without any need for type checking inside the main loop.
8.

In summary, NumPy is more efficient because it moves the computationally intensive work from the slow, flexible Python world to the fast, specialized C/Fortran world. It achieves this through **vectorization**, which leverages its **contiguous and homogeneous memory layout** to execute operations in highly optimized, pre-compiled code.

Question 5

How do you check the memory size of a NumPy array?

Theory

Checking the memory size of a NumPy array is important for performance tuning and memory management, especially when working with large datasets. NumPy provides several attributes on the ndarray object to inspect its memory footprint precisely.

Key Attributes

Let arr be a NumPy ndarray.

1. **arr.nbytes**:
   - **Description**: This is the most direct and common method. It returns an integer representing the **total number of bytes** consumed by the elements of the array.
   - **Calculation**: It is equivalent to arr.size * arr.itemsize.
2.
3. **arr.itemsize**:
   - **Description**: Returns the size in **bytes of a single element** in the array. This is determined by the array's dtype.
   - **Example**: For dtype=float64, itemsize is 8. For dtype=int16, itemsize is 2.
4.
5. **arr.size**:
   - **Description**: Returns the total number of elements in the array. You can calculate the total memory by multiplying this by arr.itemsize.
6.

Code Example

Generated python

```
    import numpy as np
import sys

# Create a sample array with a specific dtype
# 4 rows, 5 columns, with 32-bit floating point numbers (4 bytes each)
arr = np.zeros((4, 5), dtype=np.float32)

# --- Using NumPy attributes ---
print("--- Using NumPy Array Attributes ---")

# itemsize: Size of one element
print(f"Size of one element (arr.itemsize): {arr.itemsize} bytes")

# size: Total number of elements
print(f"Total number of elements (arr.size): {arr.size}")

# nbytes: Total memory consumed by the data
print(f"Total memory of array data (arr.nbytes): {arr.nbytes} bytes")

# --- Verification ---
assert arr.nbytes == arr.size * arr.itemsize

# --- Using sys.getsizeof() (for comparison) ---
# Note: This is generally not the recommended way for NumPy arrays
print("\n--- Using sys.getsizeof() ---")
print(f"Memory reported by sys.getsizeof(arr): {sys.getsizeof(arr)} bytes")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code

```
    --- Using NumPy Array Attributes ---
Size of one element (arr.itemsize): 4 bytes
Total number of elements (arr.size): 20
Total memory of array data (arr.nbytes): 80 bytes

--- Using sys.getsizeof() ---
Memory reported by sys.getsizeof(arr): 192 bytes
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Important Distinction: arr.nbytes vs. sys.getsizeof()

- arr.nbytes: Gives you the size of the **actual data** in the array's buffer. This is usually what you care about when considering the memory footprint of your data.
- sys.getsizeof(arr): Gives you the size of the **Python object** that represents the array. This includes the size of the data buffer *plus* the overhead for the object's metadata (like shape, strides, dtype, etc.). This is why sys.getsizeof() reports a larger value.

For understanding the memory consumption of the data itself, **arr.nbytes** is the correct and preferred attribute to use.

Question 6

How do you create a record array in NumPy?

Theory

A **record array**, or more generally a **structured array**, is a special type of ndarray in NumPy that allows for storing **heterogeneous** data types, similar to a row in a SQL table or a struct in C.

Instead of each element being a single number, each element in a structured array is a **record**, and each record is composed of several named **fields**. Each field can have its own data type.

This is useful for representing tabular data directly within NumPy, although for general-purpose data manipulation, Pandas DataFrames are often more convenient.

How to Create One

There are several ways to create a structured array, but the most common is by defining a dtype that specifies the names and data types of each field.

The dtype can be specified as a list of tuples, where each tuple is ('field_name', 'data_type').

Code Example

Generated python

```
    import numpy as np

# --- 1. Define the data and the dtype ---
# Imagine we have data for a list of products
data = [('Laptop', 1, 1200.50),
        ('Mouse', 2, 25.00),
        ('Keyboard', 3, 75.99)]

# Define the structure of each record
# Field 1: 'name', a Unicode string of max 20 chars
# Field 2: 'id', a 32-bit integer
# Field 3: 'price', a 64-bit float
product_dtype = [('name', 'U20'), ('id', 'i4'), ('price', 'f8')]

# --- 2. Create the structured array ---
products = np.array(data, dtype=product_dtype)

print("--- The Structured Array ---")
print(products)
print(f"\nDtype of the array: {products.dtype}")
print(f"Shape of the array: {products.shape}")

# --- 3. Accessing Data ---
print("\n--- Accessing Data ---")

# Access a specific record (row) by index
print(f"First record: {products[0]}")

# Access a specific field (column) by its name
print(f"All product names: {products['name']}")
print(f"All prices: {products['price']}")

# Access a specific field of a specific record
print(f"Price of the second product: {products[1]['price']}")

# --- 4. Performing Operations ---
# We can perform vectorized operations on the fields
total_value = np.sum(products['price'])
print(f"\nTotal value of all products: {total_value:.2f}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code

```
    --- The Structured Array ---
[('Laptop', 1, 1200.5 ) ('Mouse', 2,   25.  ) ('Keyboard', 3,   75.99)]

Dtype of the array: [('name', '<U20'), ('id', '<i4'), ('price', '<f8')]
Shape of the array: (3,)

--- Accessing Data ---
First record: ('Laptop', 1, 1200.5)
All product names: ['Laptop' 'Mouse' 'Keyboard']
All prices: [1200.5   25.     75.99]
Price of the second product: 25.0

--- Performing Operations ---
Total value of all products: 1301.49
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Explanation

1. **dtype Definition**: We define the structure [('name', 'U20'), ('id', 'i4'), ('price', 'f8')].
    - 'U20': A Unicode string with a maximum length of 20.
    - 'i4': A 4-byte (32-bit) integer.
    - 'f8': An 8-byte (64-bit) float.
2.

3. **Array Creation**: We pass our list of tuples and the custom dtype to np.array(). NumPy creates a 1D array where each element is a record conforming to our defined structure.
4. **Accessing Fields**: The key feature is that you can now access the data for an entire "column" using its field name as a string index (e.g., products['price']). This returns a standard, homogeneous NumPy array, on which you can perform fast vectorized operations.

Question 7

How can NumPy be used for audio signal processing?

Theory

NumPy is the cornerstone of audio signal processing in Python. An audio signal, in its digital form, is simply a **time series—**a sequence of numbers representing the amplitude of the sound wave at discrete, equally spaced points in time. This sequence can be represented perfectly as a **1D NumPy array**.

Once the audio is in a NumPy array, the full power of NumPy's numerical and mathematical functions can be applied to analyze and manipulate it.

Key Applications in Audio Signal Processing

1. **Reading and Representing Audio**:
    - Libraries like scipy.io.wavfile or librosa are used to read audio files (like .wav files). They return two things: the **sampling rate** (e.g., 44100 Hz) and the audio data itself as a **NumPy array**. The array's dtype (e.g., int16 or float32) represents the bit depth of the audio.
2.
3. **Basic Audio Manipulation**:
    - **Changing Volume**: This is simply a multiplication operation. louder_signal = signal_array * 1.5.
    - **Mixing Audio**: Two audio signals can be mixed by adding their NumPy arrays together.
    - **Slicing**: You can easily extract a specific segment of the audio by slicing the array.
4.
5. **Frequency Analysis (Fourier Transform)**:
    - **Concept**: This is the most fundamental analysis task. The **Fast Fourier Transform (FFT)** is used to convert the audio signal from the time domain (amplitude vs. time) to the **frequency domain** (amplitude vs. frequency).
    - **NumPy's Role**: NumPy's np.fft module provides a highly optimized implementation of the FFT algorithm.

    - **Use Case**: This allows you to see which frequencies (musical notes, pitches) are present in the audio signal. It is the basis for equalizers, pitch detection, and many other audio effects.
6.
7. **Filtering**:
    - **Concept**: Applying filters to an audio signal to remove or enhance certain frequencies.
    - **NumPy's Role**: Filtering is often done by performing a **convolution** between the audio signal array and a filter kernel array. NumPy's np.convolve can be used for this. More advanced filtering is done in the frequency domain using the FFT.
8.

Code Example

This example demonstrates reading a WAV file, performing an FFT, and visualizing the frequency spectrum.

Generated python

```
    import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile

# --- 1. Generate a sample audio signal (a sine wave) ---
# In a real case, you would use: sample_rate, signal = wavfile.read('my_audio.wav')
sample_rate = 44100  # Hz
duration = 1.0       # seconds
frequency1 = 440.0   # Hz (A4 note)
frequency2 = 880.0   # Hz (A5 note)

# Create the time axis
t = np.linspace(0., duration, int(sample_rate * duration))
# Create a signal with two frequencies
signal = 0.5 * np.sin(2. * np.pi * frequency1 * t) + 0.3 * np.sin(2. * np.pi * frequency2 * t)

# --- 2. Perform Fourier Transform ---
# Use np.fft.fft to get the complex frequency components
# Use np.fft.fftfreq to get the corresponding frequency bins
n_samples = len(signal)
yf = np.fft.fft(signal)
xf = np.fft.fftfreq(n_samples, 1 / sample_rate)

# --- 3. Visualize the results ---
plt.figure(figsize=(12, 6))

# Plot the time-domain signal
```

```
plt.subplot(1, 2, 1)
plt.plot(t[:500], signal[:500]) # Plot only the first 500 samples for clarity
plt.title("Time Domain Signal")
plt.xlabel("Time [s]")
plt.ylabel("Amplitude")
plt.grid(True)

# Plot the frequency-domain spectrum
plt.subplot(1, 2, 2)
# We plot the absolute value and only the positive frequencies
plt.plot(xf[:n_samples//2], 2.0/n_samples * np.abs(yf[:n_samples//2]))
plt.title("Frequency Domain (Spectrum)")
plt.xlabel("Frequency [Hz]")
plt.ylabel("Amplitude")
plt.xlim(0, 1200) # Zoom in on the relevant frequencies
plt.grid(True)

plt.tight_layout()
plt.show()
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

The plot on the right clearly shows two sharp peaks at 440 Hz and 880 Hz, correctly identifying the frequencies we put into the signal. This demonstrates how NumPy's fft module can be used to analyze the content of an audio signal.

Question 8

How do you handle NaN or infinite values in a NumPy array?

Theory

NaN (Not a Number) and inf (infinity) are special floating-point values that can appear in NumPy arrays as a result of invalid mathematical operations (like 0/0 for NaN or 1/0 for inf) or from loading data with missing entries.

These values are "contagious" in that they will propagate through computations (e.g., 10 + np.nan results in nan). Therefore, they must be identified and handled before performing most numerical analysis. NumPy provides a specialized set of functions for this.

1. Detecting NaN and inf

- **np.isnan(arr)**: Returns a boolean array of the same shape as arr, with True where elements are NaN and False otherwise.
- **np.isinf(arr)**: Returns a boolean array with True for inf and -inf elements.
- **np.isfinite(arr)**: The inverse of the above. Returns True for elements that are *not* NaN or inf. This is often the most useful check.

Important Note: You cannot use standard equality comparison to check for NaN. np.nan == np.nan will always return False. You must use np.isnan().

2. Counting NaN and inf

- You can combine the detection functions with np.sum() to count the number of invalid values. Since True is treated as 1 and False as 0, the sum of the boolean array gives the count.
    - np.isnan(arr).sum()
-

3. Removing or Replacing NaN and inf

- **Removing**: You can use boolean indexing to filter out NaN values, but this is only straightforward for 1D arrays.
    - arr[~np.isnan(arr)] or arr[np.isfinite(arr)]
-
- **Replacing (Imputation)**: This is the most common approach.
    - **np.nan_to_num(arr)**: A powerful function that replaces NaN with a specified value (default is 0.0), positive infinity with a large positive number, and negative infinity with a large negative number.
    - **Boolean Indexing**: You can use the boolean masks to selectively replace values.
        - arr[np.isnan(arr)] = 0
        - Calculate a mean or median ignoring NaNs, and then fill. mean = np.nanmean(arr), then arr[np.isnan(arr)] = mean.
    -
-

4. NaN-Safe Mathematical Functions

- NumPy provides versions of common aggregate functions that automatically ignore NaN values during computation.
- **Examples**:
    - np.nanmean()
    - np.nanmedian()
    - np.nansum()
    - np.nanstd()

-

Code Example

Generated python

```
    import numpy as np

# Create an array with NaN and inf values
arr = np.array([1.0, 2.0, np.nan, 4.0, np.inf, -np.inf, 6.0])
print(f"Original array: {arr}\n")

# --- 1. Detection ---
print("--- Detection ---")
print(f"Is NaN? {np.isnan(arr)}")
print(f"Is Infinite? {np.isinf(arr)}")
print(f"Is Finite? {np.isfinite(arr)}\n")

# --- 2. Counting ---
print("--- Counting ---")
print(f"Number of NaN values: {np.isnan(arr).sum()}\n")

# --- 3. NaN-safe Computations ---
print("--- NaN-safe Computations ---")
# Standard mean will fail (propagate nan)
print(f"np.mean(arr): {np.mean(arr)}")
# NaN-safe mean will work
print(f"np.nanmean(arr): {np.nanmean(arr)}\n")

# --- 4. Replacing ---
print("--- Replacing ---")
# Replace NaN with 0 and inf with large numbers
arr_clean1 = np.nan_to_num(arr)
print(f"Using np.nan_to_num(): {arr_clean1}")

# Replace NaN with the mean of the finite values
# Create a copy to avoid modifying the original
arr_clean2 = arr.copy()
mean_val = np.nanmean(arr_clean2[np.isfinite(arr_clean2)]) # Calculate mean of finite values
arr_clean2[np.isnan(arr_clean2)] = mean_val
print(f"Replacing NaN with mean: {arr_clean2}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

```
Generated code
    Original array: [  1.   2.  nan   4.  inf  -inf   6.]

--- Detection ---
Is NaN? [False False  True False False False False]
Is Infinite? [False False False False  True  True False]
Is Finite? [ True  True False  True False False  True]

--- Counting ---
Number of NaN values: 1

--- NaN-safe Computations ---
np.mean(arr): nan
np.nanmean(arr): 3.25

--- Replacing ---
Using np.nan_to_num(): [ 1.00000000e+00  2.00000000e+00  0.00000000e+00
4.00000000e+00
  1.79769313e+308 -1.79769313e+308  6.00000000e+00]
Replacing NaN with mean: [  1.    2.    3.25  4.    inf  -inf   6.  ]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Question 9

What methods are there in NumPy to deal with missing data?

Theory

While the **Pandas** library is generally the preferred tool for handling missing data in a structured, tabular format (with its powerful .isnull(), .fillna(), and .dropna() methods), NumPy provides its own set of tools for dealing with missing data at the array level.

In NumPy, missing data is typically represented by the special floating-point value **np.nan** (Not a Number). The primary methods for handling np.nan fall into three categories: **detection**, **removal**, and **computation**.

1. Detection

The first step is to identify where the missing values are.

- **np.isnan(arr)**: This is the fundamental function for detection. It returns a boolean array of the same shape as the input array, with True at the locations of np.nan values and False everywhere else.

2. Removal

This involves creating a new array that excludes the missing values.

- **Boolean Indexing**: This is the most common way to remove NaNs. You create a boolean mask and use it to select only the valid elements.
  - arr[~np.isnan(arr)]
  - arr[np.isfinite(arr)] (This also removes inf values)
-
- **Note**: This is most straightforward for 1D arrays. For 2D arrays, this will flatten the array. To remove entire rows or columns containing NaNs, you would typically use boolean indexing along an axis:
  - arr[~np.isnan(arr).any(axis=1)] (Removes rows with any NaNs)
-

3. Computation (Calculation while ignoring NaNs)

This is often the most practical approach. Instead of changing the array, you use special versions of aggregate functions that are "NaN-aware."

- **NaN-Safe Functions**: NumPy provides a suite of functions that perform a calculation while simply ignoring any NaN values.
  - np.nanmean(arr): Computes the mean, ignoring NaNs.
  - np.nanmedian(arr): Computes the median, ignoring NaNs.
  - np.nansum(arr): Computes the sum, treating NaNs as zero.
  - np.nanstd(arr): Computes the standard deviation, ignoring NaNs.
  - np.nanmin(arr) / np.nanmax(arr): Find the min/max, ignoring NaNs.
-

4. Replacement (Imputation)

This involves filling the NaN values with another value.

- **Boolean Indexing**: Use a boolean mask to select the NaNs and assign a new value.
  - arr[np.isnan(arr)] = 0 (Replace with a constant)
  - mean = np.nanmean(arr), arr[np.isnan(arr)] = mean (Replace with the mean)
-
- **np.nan_to_num()**: A specialized function that replaces NaNs with 0 (or another specified value) and also handles infinite values.

Code Example

Generated python
```
    import numpy as np

arr = np.array([[1.0, 2.0, np.nan],
               [4.0, 5.0, 6.0],
               [np.nan, 8.0, 9.0]])

print(f"Original Array:\n{arr}\n")

# --- 1. Detection ---
nan_mask = np.isnan(arr)
print(f"NaN Mask:\n{nan_mask}\n")

# --- 2. Removal ---
# Removing rows with any NaN values
arr_no_nan_rows = arr[~np.isnan(arr).any(axis=1)]
print(f"Array with NaN rows removed:\n{arr_no_nan_rows}\n")

# --- 3. NaN-aware Computation ---
mean_val = np.nanmean(arr)
mean_val_axis0 = np.nanmean(arr, axis=0)
print(f"Mean ignoring NaNs (overall): {mean_val:.2f}")
print(f"Mean ignoring NaNs (by column): {np.round(mean_val_axis0, 2)}\n")

# --- 4. Replacement ---
arr_imputed = arr.copy()
# Impute with the mean of each column
col_means = np.nanmean(arr_imputed, axis=0)
# Find indices where NaN is present
nan_indices = np.where(np.isnan(arr_imputed))
# Replace NaNs with the mean of their respective columns
arr_imputed[nan_indices] = np.take(col_means, nan_indices[1])

print(f"Array after imputing with column means:\n{np.round(arr_imputed, 2)}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Question 10

How do you find unique values and their counts in a NumPy array?

Theory

Finding the unique elements in an array and counting their occurrences are common operations in data analysis, for tasks like understanding the distribution of categorical variables or building a vocabulary from text data. NumPy provides highly optimized functions for these tasks.

Key Functions

1. **np.unique(arr, return_counts=False)**:
    - **Purpose**: This is the primary function for this task.
    - **Default Behavior**: By default (return_counts=False), it returns a **sorted array** containing only the **unique** elements of the input array.
    - **With return_counts=True**: This is the more powerful usage. When set to True, the function returns a **tuple of two arrays**:
        1. The array of unique values.
        2. An array of the same shape containing the **count** of how many times each unique value appeared in the original array.
    -
2.
3. **np.bincount(arr)**:
    - **Purpose**: This is a very fast and specialized function for counting occurrences, but it has a specific requirement.
    - **Requirement**: It only works on 1D arrays of **non-negative integers**.
    - **How it works**: It returns an array where the value at index i is the number of times i appeared in the input array. The length of the output array is one greater than the maximum value in the input array.
4.

Code Example

Generated python

```
   import numpy as np

# A sample array with duplicate values
arr = np.array([1, 2, 5, 2, 3, 5, 1, 4, 5, 2])

print(f"Original Array: {arr}\n")

# --- 1. Using np.unique ---
print("--- Using np.unique ---")

# Get only the unique values
unique_values = np.unique(arr)
print(f"Unique values (sorted): {unique_values}")
```

```
# Get unique values and their counts
unique_vals, counts = np.unique(arr, return_counts=True)
print(f"Unique values: {unique_vals}")
print(f"Counts: {counts}\n")

# Combine them for a nice display
unique_counts = np.asarray((unique_vals, counts)).T
print("Value | Count")
print("------------")
for val, count in unique_counts:
    print(f"{val:5d} | {count:5d}")


# --- 2. Using np.bincount (for non-negative integers) ---
print("\n--- Using np.bincount ---")
# bincount is often faster for this specific use case
bincount_result = np.bincount(arr)
print(f"Result of np.bincount(arr): {bincount_result}")
print("(Value at index 'i' is the count of 'i')")

# Displaying the results from bincount
print("\nValue | Count (from bincount)")
print("------------------------------")
for i, count in enumerate(bincount_result):
    if count > 0:
        print(f"{i:5d} | {count:5d}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code

```
    Original Array: [1 2 5 2 3 5 1 4 5 2]

--- Using np.unique ---
Unique values (sorted): [1 2 3 4 5]
Unique values: [1 2 3 4 5]
Counts: [2 3 1 1 3]

Value | Count
------------
```

```
   1 |     2
   2 |     3
   3 |     1
   4 |     1
   5 |     3

--- Using np.bincount ---
Result of np.bincount(arr): [0 2 3 1 1 3]
(Value at index 'i' is the count of 'i')

Value | Count (from bincount)
------------------------------
   1 |     2
   2 |     3
   3 |     1
   4 |     1
   5 |     3
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Choosing Between np.unique and np.bincount

- Use **np.unique** when you are working with floats, negative numbers, or strings, or when you don't need the counts. It is the general-purpose solution.
- Use **np.bincount** when you are working with an array of non-negative integers and you need the counts. It is generally faster than np.unique for this specific task.

Question 11

How can you use NumPy arrays with Cython for performance optimization?

Theory

While NumPy is very fast because its core is written in C, sometimes you need to write a custom algorithm with complex logic that cannot be expressed as a simple sequence of vectorized NumPy operations. Writing this algorithm in a pure Python for loop would be very slow.

This is where **Cython** comes in. Cython is a programming language that makes it easy to write C extensions for Python. It is a superset of Python, so you can start with Python code and then gradually add C-style static type declarations to get massive performance boosts.

Using NumPy arrays with Cython is a powerful combination because you can write a function in Cython that operates **directly on the raw data buffer of a NumPy array**, bypassing the slow Python interpreter for the looping logic.

The Process

1. **Write a Cython file (.pyx)**: You write your function in a file with a .pyx extension.
2. **Add Type Declarations**: Inside the function, you declare the types of your variables (e.g., cdef int i) and, most importantly, you create **typed memoryviews** of the NumPy arrays. A memoryview allows Cython to access the array's data buffer directly at the C level.
3. **Disable Safety Checks (Optional but fast)**: For maximum performance, you can turn off Python's default safety checks (like bounds checking) using compiler directives. This is faster but requires you to be sure your code is correct.
4. **Compile the Cython code**: You use a setup.py file to compile the .pyx file into a C extension module (.so on Linux or .pyd on Windows).
5. **Import and Use**: You can then import the compiled module into your Python script and call the super-fast Cython function.

Code Example

Let's create a Cython function that calculates the sum of squared differences between two NumPy arrays, a common calculation in machine learning.

1. The Cython file (fast_sum.pyx):

Generated cython

```
    # To use this, you would save it as 'fast_sum.pyx'
import numpy as np
# cimport is used to import C-level definitions
cimport numpy as np
cimport cython

# This decorator turns off some safety checks for speed
@cython.boundscheck(False)
@cython.wraparound(False)
def sum_squared_diff_cython(np.ndarray[double, ndim=1] a, np.ndarray[double, ndim=1] b):
    """
    Calculates the sum of squared differences between two 1D double arrays.
    """
    # Declare C-level variables with cdef
    cdef double total = 0.0
    cdef int i
    cdef int n = a.shape[0]
    # This loop runs entirely at the C level, which is very fast
    for i in range(n):
        total += (a[i] - b[i]) ** 2

    return total
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Cython
IGNORE_WHEN_COPYING_END

2. The setup file (setup.py):

Generated python
```
     # Save this as 'setup.py' in the same directory
from setuptools import setup
from Cython.Build import cythonize
import numpy

setup(
    ext_modules = cythonize("fast_sum.pyx"),
    include_dirs=[numpy.get_include()]
)
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

To compile: Run python setup.py build_ext --inplace in your terminal. This will create fast_sum.c and a compiled module.

3. The Python script to use it:

Generated python
```
     import numpy as np
import time
# This import works after you've compiled the .pyx file
from fast_sum import sum_squared_diff_cython

# Create some large arrays
a = np.random.rand(10_000_000)
b = np.random.rand(10_000_000)

# --- Compare performance ---
# NumPy vectorized version (very fast)
start = time.time()
result_numpy = np.sum((a - b) ** 2)
numpy_time = time.time() - start
print(f"NumPy vectorized time: {numpy_time:.6f} seconds")

# Cython version
start = time.time()
result_cython = sum_squared_diff_cython(a, b)
cython_time = time.time() - start
print(f"Cython function time:  {cython_time:.6f} seconds")

print(f"\nResults are close: {np.isclose(result_numpy, result_cython)}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Result

When you run this, you will find that the Cython function's performance is very close to (and sometimes even slightly faster than) the pure NumPy vectorized version. This demonstrates that Cython allows you to write custom, complex algorithms with the **same performance as native NumPy operations**, something that is impossible to achieve with a pure Python for loop.

Question 1

How do you create a NumPy array from a regular Python list?

Theory

The most common way to create a NumPy array is by using the numpy.array() function. This function takes a Python list (or a nested list) as its primary argument and converts it into a NumPy ndarray.

You can also optionally specify the dtype (data type) of the elements in the array during creation. If not specified, NumPy will infer the most suitable dtype automatically.

Code Example

Generated python

```
    import numpy as np

# --- 1. Creating a 1D array from a Python list ---
```

```
python_list_1d = [1, 2, 3, 4, 5]
numpy_array_1d = np.array(python_list_1d)

print("--- 1D Array ---")
print(f"Python list: {python_list_1d}")
print(f"NumPy array: {numpy_array_1d}")
print(f"Type of result: {type(numpy_array_1d)}")
print(f"dtype of array: {numpy_array_1d.dtype}") # NumPy infers it's an integer type

# --- 2. Creating a 2D array from a nested list ---
python_list_2d = [[1, 2, 3], [4, 5, 6]]
numpy_array_2d = np.array(python_list_2d)

print("\n--- 2D Array ---")
print(f"Python nested list:\n{python_list_2d}")
print(f"NumPy 2D array:\n{numpy_array_2d}")
print(f"Shape of array: {numpy_array_2d.shape}")

# --- 3. Creating an array with a specific dtype ---
python_list_floats = [1, 2, 3]
# We can explicitly tell NumPy to use 32-bit floats
numpy_array_float32 = np.array(python_list_floats, dtype=np.float32)

print("\n--- Specifying dtype ---")
print(f"NumPy array with specified dtype: {numpy_array_float32}")
print(f"dtype of array: {numpy_array_float32.dtype}")
```

Explanation

1. We import the NumPy library, conventionally aliased as np.
2. We pass a standard Python list to np.array(). NumPy handles the conversion, creating a new ndarray object that stores the data in a contiguous, memory-efficient block.
3. For the 2D array, the nested list structure is automatically interpreted to create an array with the corresponding shape (2 rows, 3 columns).
4. In the final example, the dtype argument is used to force the array to store its elements as float32, which can be important for memory optimization or compatibility with certain libraries.

Question 2

How do you perform matrix multiplication using NumPy?

Theory

NumPy provides several ways to perform matrix multiplication. The modern and recommended approach for Python 3.5+ is to use the dedicated infix operator @, which is concise and highly readable.

For two matrices A and B, the product A @ B is defined only if the number of columns in A is equal to the number of rows in B.

Key Methods

1. **The @ operator (Preferred Method)**:
    - This is the dedicated matrix multiplication operator. It is the cleanest and most "Pythonic" way to perform the operation.
2.
3. **numpy.matmul() function**:
    - This is the function that the @ operator calls under the hood. It behaves identically to @.
4.
5. **.dot() method**:
    - A.dot(B) is another way to perform matrix multiplication. For 2D arrays, it behaves exactly like matmul and @.
    - **Difference**: For higher-dimensional arrays (N > 2), .dot() and matmul have different broadcasting and stacking behaviors. For general-purpose use, matmul or @ is usually the more intuitive and recommended choice.
6.
7. *** operator (Element-wise product)**:
    - It is a **critical mistake** to use the * operator for matrix multiplication. The * operator performs an **element-wise** product (Hadamard product), not a true matrix product.
8.

Code Example

Generated python

```
    import numpy as np

# Create two compatible matrices for multiplication
# A is (2, 3) and B is (3, 2)
A = np.array([[1, 2, 3],
              [4, 5, 6]])

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

```
# The resulting matrix C will have shape (2, 2)
print(f"Matrix A (shape {A.shape}):\n{A}")
print(f"Matrix B (shape {B.shape}):\n{B}\n")

# --- 1. Using the @ operator (Preferred) ---
C_at = A @ B
print(f"--- Using @ operator ---\nResult (shape {C_at.shape}):\n{C_at}\n")

# --- 2. Using np.matmul() ---
C_matmul = np.matmul(A, B)
print(f"--- Using np.matmul() ---\nResult:\n{C_matmul}\n")

# --- 3. Using the .dot() method ---
C_dot = A.dot(B)
print(f"--- Using .dot() method ---\nResult:\n{C_dot}\n")

# --- 4. Demonstrating the WRONG way (element-wise product) ---
print("--- Incorrect Usage: * operator ---")
try:
    C_star = A * B
except ValueError as e:
    print(f"Error: {e}")
    print("This fails because the shapes are not compatible for element-wise multiplication.")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code
```
    Matrix A (shape (2, 3)):
[[1 2 3]
 [4 5 6]]
Matrix B (shape (3, 2)):
[[ 7  8]
 [ 9 10]
 [11 12]]

--- Using @ operator ---
Result (shape (2, 2)):
[[ 58  64]
 [139 154]]
```

```
--- Using np.matmul() ---
Result:
[[ 58  64]
 [139 154]]

--- Using .dot() method ---
Result:
[[ 58  64]
 [139 154]]

--- Incorrect Usage: * operator ---
Error: operands could not be broadcast together with shapes (2,3) (3,2)
This fails because the shapes are not compatible for element-wise multiplication.
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Question 3

Explain how to invert a matrix in NumPy.

Theory

Inverting a matrix is a common operation in linear algebra used to solve systems of linear equations. A matrix A can only be inverted if it is **square** and **non-singular** (i.e., its determinant is not zero).

The inverse of a matrix A is denoted A^{-1}. When multiplied by A, it results in the identity matrix I. A @ A^{-1} = I.

NumPy's linear algebra module, numpy.linalg, provides a straightforward function for this purpose: np.linalg.inv().

Code Example

Generated python

```python
    import numpy as np

# --- 1. Inverting a 2x2 matrix ---
A = np.array([[4, 7],
              [2, 6]])

print(f"Original Matrix A:\n{A}\n")
```

```
try:
    A_inv = np.linalg.inv(A)
    print(f"Inverse of A:\n{A_inv}\n")

    # --- 2. Verification ---
    # The product of a matrix and its inverse should be the identity matrix
    identity = A @ A_inv
    print(f"Verification (A @ A_inv):\n{np.round(identity, 5)}\n")
    # np.allclose is used to check if two arrays are element-wise equal within a tolerance
    print(f"Is the result close to the identity matrix? {np.allclose(identity, np.eye(2))}")

except np.linalg.LinAlgError as e:
    print(f"Error: {e}")


# --- 3. Attempting to invert a singular matrix ---
B = np.array([[1, 2],
              [2, 4]]) # This matrix is singular (determinant is 0)

print("\n--- Attempting to invert a singular matrix ---")
print(f"Matrix B:\n{B}\n")

try:
    B_inv = np.linalg.inv(B)
except np.linalg.LinAlgError as e:
    print(f"Error: {e}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code

```
    Original Matrix A:
[[4 7]
 [2 6]]

Inverse of A:
[[ 0.6 -0.7]
 [-0.2  0.4]]

Verification (A @ A_inv):
```

```
[[1. 0.]
 [0. 1.]]

Is the result close to the identity matrix? True

--- Attempting to invert a singular matrix ---
Matrix B:
[[1 2]
 [2 4]]

Error: Singular matrix
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Explanation

1. **Function Call**: The inverse is computed by calling np.linalg.inv(A).
2. **Error Handling**: The code is wrapped in a try...except block. np.linalg.inv() will raise a LinAlgError if the input matrix is not square or if it is singular. This is the proper way to handle cases where an inverse cannot be computed.
3. **Verification**: After computing the inverse, it's a good practice to verify the result. We multiply the original matrix A by its computed inverse A_inv. The result should be the identity matrix (np.eye(2)). Due to floating-point inaccuracies, the result might not be *exactly* the identity matrix (e.g., some elements might be 1.0000000000000002 or a very small number like 2.22e-16). Therefore, we use np.allclose() for a robust comparison instead of ==.

Best Practice Note

In numerical applications, you should **avoid computing the inverse explicitly** whenever possible. For example, to solve a system of linear equations Ax = b, it is both faster and more numerically stable to use np.linalg.solve(A, b) rather than calculating $x = A^{-1}b$.

Question 4

How do you calculate the determinant of a matrix?

Theory

The **determinant** is a scalar value that can be computed from the elements of a **square matrix**. It provides important information about the matrix, such as whether it is invertible. A matrix is invertible if and only if its determinant is non-zero.

NumPy's linear algebra module, numpy.linalg, provides the det() function to calculate this value.

Code Example

Generated python

```
    import numpy as np

# --- 1. A non-singular 2x2 matrix ---
A = np.array([[3, 8],
              [4, 6]])

det_A = np.linalg.det(A)
print(f"Matrix A:\n{A}")
print(f"Determinant of A: {det_A:.2f}") # Formula: (3*6) - (8*4) = 18 - 32 = -14
print("-" * 30)

# --- 2. A singular 2x2 matrix ---
# A matrix is singular if one row/column is a multiple of another
B = np.array([[1, 2],
              [2, 4]])

det_B = np.linalg.det(B)
print(f"Matrix B:\n{B}")
print(f"Determinant of B: {det_B:.2f}") # Formula: (1*4) - (2*2) = 0
print("-" * 30)

# --- 3. A 3x3 matrix ---
C = np.array([[6, 1, 1],
              [4, -2, 5],
              [2, 8, 7]])

det_C = np.linalg.det(C)
print(f"Matrix C:\n{C}")
print(f"Determinant of C: {det_C:.2f}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code
```
    Matrix A:
[[3 8]
 [4 6]]
Determinant of A: -14.00
------------------------------
Matrix B:
[[1 2]
 [2 4]]
Determinant of B: 0.00
------------------------------
Matrix C:
[[6 1 1]
 [4 -2 5]
 [2 8 7]]
Determinant of C: -306.00
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Explanation

1. **Function Call**: The determinant is computed simply by calling np.linalg.det() and passing the square matrix as the argument.
2. **Input Requirements**: The function will raise a LinAlgError if the input matrix is not square.
3. **Singular Matrix**: As shown with matrix B, the determinant is 0. This confirms that the matrix is singular and cannot be inverted. Due to floating-point arithmetic, a computed determinant for a singular matrix might be a very small number close to zero (e.g., 1.23e-17) rather than exactly 0.0. It's good practice to check if the absolute value is close to zero, e.g., np.isclose(det, 0).

Question 5

How do you concatenate two arrays in NumPy?

Theory

Concatenation is the process of joining a sequence of arrays along an existing axis. NumPy provides a general-purpose function, np.concatenate(), for this task.

Key Arguments:

- arrays: A tuple or list of arrays to be concatenated. These arrays must have the same shape, except in the dimension corresponding to the axis of concatenation.
- axis: The axis along which the arrays will be joined.
    - axis=0: Stacks the arrays vertically (row-wise).
    - axis=1: Stacks the arrays horizontally (column-wise).
-

NumPy also provides convenience functions like np.vstack() and np.hstack() for vertical and horizontal stacking, respectively.

Code Example

Generated python

```
   import numpy as np

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

print(f"Array a (shape {a.shape}):\n{a}")
print(f"Array b (shape {b.shape}):\n{b}\n")

# --- 1. Concatenation along axis=0 (Vertical Stacking) ---
# The number of columns must match (a has 2, b has 2)
# The shape of the result will be (2+1, 2) = (3, 2)
c_axis0 = np.concatenate((a, b), axis=0)
print(f"--- Concatenate along axis=0 ---\nResult (shape {c_axis0.shape}):\n{c_axis0}\n")

# Using np.vstack for the same result
c_vstack = np.vstack((a, b))
print(f"--- Using np.vstack ---\nResult:\n{c_vstack}\n")

# --- 2. Concatenation along axis=1 (Horizontal Stacking) ---
# To stack horizontally, the number of rows must match.
# Let's reshape b to be a (2, 1) column vector
b_col = np.array([[5], [6]]) # or b.T.reshape(2,1)
print(f"Array b reshaped to a column (shape {b_col.shape}):\n{b_col}\n")

# The shape of the result will be (2, 2+1) = (2, 3)
c_axis1 = np.concatenate((a, b_col), axis=1)
print(f"--- Concatenate along axis=1 ---\nResult (shape {c_axis1.shape}):\n{c_axis1}\n")

# Using np.hstack for the same result
c_hstack = np.hstack((a, b_col))
```

```
print(f"--- Using np.hstack ---\nResult:\n{c_hstack}\n")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code

```
    Array a (shape (2, 2)):
[[1 2]
 [3 4]]
Array b (shape (1, 2)):
[[5 6]]

--- Concatenate along axis=0 ---
Result (shape (3, 2)):
[[1 2]
 [3 4]
 [5 6]]

--- Using np.vstack ---
Result:
[[1 2]
 [3 4]
 [5 6]]

Array b reshaped to a column (shape (2, 1)):
[[5]
 [6]]

--- Concatenate along axis=1 ---
Result (shape (2, 3)):
[[1 2 5]
 [3 4 6]]

--- Using np.hstack ---
Result:
[[1 2 5]
 [3 4 6]]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.

IGNORE_WHEN_COPYING_END

Question 6

Describe how you would flatten a multi-dimensional array.

Theory

Flattening a multi-dimensional array means converting it into a single, one-dimensional (1D) array. This is a common operation, for example, when preparing the output of convolutional layers in a neural network to be fed into a dense layer.

NumPy provides several methods to do this, with a key difference being whether they return a **copy** of the data or a **view**.

Key Methods

1. **ndarray.flatten()**:
    - **What it is**: A method of the ndarray object.
    - **Returns**: It always returns a **copy** of the data. The new array is completely independent of the original.
    - **Order**: You can specify the order of flattening: 'C' for row-major (default) or 'F' for column-major.
2.
3. **numpy.ravel()**:
    - **What it is**: A top-level NumPy function.
    - **Returns**: It returns a **view** of the original array whenever possible. It will only create a copy if necessary (e.g., if the array's data is not contiguous in memory).
    - **Advantage**: It is generally faster and more memory-efficient than .flatten() because it avoids a data copy if it can.
4.
5. **ndarray.reshape(-1)**:
    - **What it is**: Using the .reshape() method with -1 as the new shape.
    - **Returns**: It also returns a **view** whenever possible, just like ravel().
    - **Advantage**: This is a very common and idiomatic way to flatten an array in the NumPy community. The -1 tells NumPy to automatically infer the correct size for that dimension.
6.

Code Example

Generated python

```
    import numpy as np

# Create a 2x3 array
```

```
arr = np.array([[1, 2, 3],
                [4, 5, 6]])

print(f"Original array (shape {arr.shape}):\n{arr}\n")

# --- 1. Using .flatten() ---
flattened_arr = arr.flatten()
print(f"--- Using .flatten() ---")
print(f"Result: {flattened_arr}")
# flatten() returns a copy, so changes don't affect the original
flattened_arr[0] = 99
print(f"Original array after modifying the flattened copy:\n{arr}\n")


# --- 2. Using .ravel() ---
# Reset the array
arr = np.array([[1, 2, 3], [4, 5, 6]])
raveled_arr = arr.ravel()
print(f"--- Using .ravel() ---")
print(f"Result: {raveled_arr}")
# ravel() returns a view, so changes DO affect the original
raveled_arr[0] = 88
print(f"Original array after modifying the raveled view:\n{arr}\n")


# --- 3. Using .reshape(-1) ---
# Reset the array
arr = np.array([[1, 2, 3], [4, 5, 6]])
reshaped_arr = arr.reshape(-1)
print(f"--- Using .reshape(-1) ---")
print(f"Result: {reshaped_arr}")
# reshape(-1) also returns a view
reshaped_arr[0] = 77
print(f"Original array after modifying the reshaped view:\n{arr}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Recommendation

- Use **.flatten()** when you need a separate copy of the data that you can modify without affecting the original array.

- Use **.ravel()** or **.reshape(-1)** when you want the most memory-efficient and fastest option, and you don't need to modify the flattened array (or you intend for the modifications to affect the original). .reshape(-1) is often the most common and readable choice.

Question 7

How do you calculate the eigenvalues and eigenvectors of a matrix in NumPy?

Theory

Eigenvalues and **eigenvectors** are fundamental concepts in linear algebra that reveal the intrinsic properties of a square matrix. An eigenvector of a matrix is a non-zero vector that, when the matrix is multiplied by it, is simply scaled by a scalar value, the eigenvalue. $Av = \lambda v$.

NumPy's numpy.linalg module provides the eig() function to compute these values.

numpy.linalg.eig()

- **Input**: A square matrix A.
- **Output**: A tuple containing two NumPy arrays:
    1. A 1D array of the **eigenvalues** (λ).
    2. A 2D array where the **columns** are the corresponding **eigenvectors** (v).
-

Code Example

Generated python

```
    import numpy as np

# Create a square, symmetric matrix
A = np.array([[4, 2],
           [2, 1]])

print(f"Original Matrix A:\n{A}\n")

# --- 1. Calculate Eigenvalues and Eigenvectors ---
eigenvalues, eigenvectors = np.linalg.eig(A)

print("--- Results ---")
print(f"Eigenvalues (λ):\n{eigenvalues}\n")
print(f"Eigenvectors (v) (as columns):\n{eigenvectors}\n")

# --- 2. Verification ---
# We can verify the relationship A @ v = λ * v for each pair.
print("--- Verification ---")
for i in range(len(eigenvalues)):
    lambda_val = eigenvalues[i]
    v_vec = eigenvectors[:, i] # Get the i-th column vector

    # Calculate A @ v
    Av = A @ v_vec

    # Calculate λ * v
    lambda_v = lambda_val * v_vec

    print(f"For eigenvalue λ_{i+1} ≈ {lambda_val:.4f}:")
    print(f"  A @ v = {np.round(Av, 4)}")
    print(f"  λ * v = {np.round(lambda_v, 4)}")
    print(f"  Are they close? {np.allclose(Av, lambda_v)}\n")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code

```
    Original Matrix A:
[[4 2]
 [2 1]]

--- Results ---
Eigenvalues (λ):
[5. 0.]

Eigenvectors (v) (as columns):
[[ 0.89442719 -0.4472136 ]
 [ 0.4472136   0.89442719]]

--- Verification ---
For eigenvalue λ_1 ≈ 5.0000:
  A @ v = [4.4721 2.2361]
  λ * v = [4.4721 2.2361]
  Are they close? True

For eigenvalue λ_2 ≈ 0.0000:
  A @ v = [-0. -0.]
```

```
λ * v = [-0.  0.]
Are they close? True
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Explanation

1. **Function Call**: We call np.linalg.eig(A), which performs the eigendecomposition.
2. **Unpacking Results**: The function returns a tuple, which we unpack into two variables: eigenvalues and eigenvectors.
3. **Interpreting Output**:
   - eigenvalues is a 1D array. In this case, [5., 0.]. The presence of a 0 eigenvalue confirms that matrix A is singular.
   - eigenvectors is a 2D array. The **first column** [0.894, 0.447] is the eigenvector corresponding to the first eigenvalue 5.0. The **second column** is the eigenvector for the second eigenvalue 0.0.
4.
5. **Verification**: The verification loop confirms that the fundamental property Av = λv holds true for both pairs, confirming the correctness of NumPy's calculation.

Question 8

How can you reverse an array in NumPy?

Theory

Reversing an array is a common manipulation task. NumPy, with its powerful slicing capabilities, provides a very simple and efficient way to do this. The most "Pythonic" and common method is to use extended slice notation.

Key Methods

1. **Using Slice Notation [::-1] (Preferred Method)**:
   - **Concept**: This is the standard idiom for reversing a sequence in Python, and it works perfectly for NumPy arrays. The slice start:stop:step with a step of -1 means to go through the array from beginning to end, backward.
   - **Returns**: This operation creates a **view** (a shallow copy), not a copy of the data. It is extremely fast and memory-efficient.
   - **Multi-dimensional**: You can apply this slice to a specific axis to reverse only that axis.

2.
3. **Using numpy.flip() function**:
    - **Concept**: A dedicated function for reversing the order of elements in an array along a specified axis.
    - **Returns**: It also returns a **view** whenever possible.
    - **Advantage**: The axis argument makes the intent very explicit, which can improve code readability, especially when reversing a specific axis of a multi-dimensional array.
4.

Code Example

Generated python

```
    import numpy as np

# --- 1. Reversing a 1D array ---
arr_1d = np.array([1, 2, 3, 4, 5, 6])
print("--- 1D Array Reversal ---")
print(f"Original 1D array: {arr_1d}")

# Using slicing
reversed_1d_slice = arr_1d[::-1]
print(f"Reversed with [::-1]: {reversed_1d_slice}")

# Using np.flip
reversed_1d_flip = np.flip(arr_1d)
print(f"Reversed with np.flip(): {reversed_1d_flip}\n")


# --- 2. Reversing a 2D array ---
arr_2d = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])
print("--- 2D Array Reversal ---")
print(f"Original 2D array:\n{arr_2d}\n")

# Reverse all elements (flattens and reverses)
print(f"arr_2d.flatten()[::-1]: {arr_2d.flatten()[::-1]}\n")

# Reverse the order of rows (axis=0)
reversed_rows = arr_2d[::-1, :] # or np.flip(arr_2d, axis=0)
print(f"Reversed rows (arr_2d[::-1, :]):\n{reversed_rows}\n")

# Reverse the order of columns (axis=1)
reversed_cols = arr_2d[:, ::-1] # or np.flip(arr_2d, axis=1)
```

```
print(f"Reversed columns (arr_2d[:, ::-1]):\n{reversed_cols}\n")

# Reverse both rows and columns
reversed_both = arr_2d[::-1, ::-1] # or np.flip(arr_2d, axis=(0, 1))
print(f"Reversed both axes:\n{reversed_both}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Question 9

How do you apply a conditional filter to a NumPy array?

Theory

Applying a conditional filter is one of the most powerful and common operations in NumPy. It is achieved using **boolean array indexing** (or "boolean masking").

The process involves two steps:

1. **Create a Boolean Mask**: Create a boolean array of the same shape as the data array, where each element is True if the corresponding element in the data array satisfies a condition, and False otherwise.
2. **Apply the Mask**: Use this boolean array as an index to the original data array. This will return a new array containing only the elements where the mask was True.

Code Example

Generated python

```
    import numpy as np

# Create an array of integers
arr = np.array([[1, 2, 3, 4, 5],
                [6, 7, 8, 9, 10]])

print(f"Original Array:\n{arr}\n")

# --- 1. Create a boolean mask ---
# Condition: Find all elements greater than 5
mask = arr > 5
print(f"Boolean Mask (arr > 5):\n{mask}\n")

# --- 2. Apply the mask to filter the data ---
# This will return a 1D array containing only the elements that satisfy the condition
filtered_data = arr[mask]
print(f"Filtered Data (elements > 5):\n{filtered_data}\n")

# --- Common Use Cases in a single line ---

# Get all even numbers
even_numbers = arr[arr % 2 == 0]
print(f"Even numbers in the array: {even_numbers}\n")

# Get numbers that are greater than 3 AND less than 9
# Use & for element-wise AND, and wrap each condition in parentheses
complex_filter = arr[(arr > 3) & (arr < 9)]
print(f"Numbers > 3 AND < 9: {complex_filter}\n")

# --- 3. Using np.where() for conditional replacement ---
# np.where() is a powerful function for conditional logic.
# Syntax: np.where(condition, value_if_true, value_if_false)

# Replace all odd numbers with -1 and keep even numbers as they are
arr_replaced = np.where(arr % 2 != 0, -1, arr)
print(f"--- Using np.where ---")
print(f"Replace odd numbers with -1:\n{arr_replaced}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code

```
    Original Array:
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]]

Boolean Mask (arr > 5):
[[False False False False False]
 [ True  True  True  True  True]]

Filtered Data (elements > 5):
[ 6  7  8  9 10]

Even numbers in the array: [ 2  4  6  8 10]
```

```
Numbers > 3 AND < 9: [4 5 6 7 8]

--- Using np.where ---
Replace odd numbers with -1:
[[-1  2 -1  4 -1]
 [ 6 -1  8 -1 10]]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Question 10

Explain how to generate random data with NumPy.

Theory

NumPy's numpy.random module is a powerful and extensive toolkit for generating random data. It can produce random numbers, or entire arrays of them, from a wide variety of probability distributions. This is essential for many tasks in machine learning, such as initializing model weights, creating synthetic datasets for testing, and implementing stochastic algorithms.

Modern NumPy (version 1.17+) uses a new, more robust random number generation system based on **Generators** and **BitGenerators**. The recommended practice is to first create a Generator instance and then call its methods.

Key Functions

1. Creating a Generator
rng = np.random.default_rng(seed=None)

- This creates an instance of the random number generator. Providing a seed is crucial for **reproducibility**.

2. Generating Random Numbers from Standard Distributions

- rng.random(size=None): Returns random floats in the half-open interval [0.0, 1.0).
- rng.integers(low, high=None, size=None): Returns random integers from low (inclusive) to high (exclusive).
- rng.standard_normal(size=None): Returns samples from the standard normal distribution (mean=0, variance=1).
- rng.normal(loc=0.0, scale=1.0, size=None): Returns samples from a normal (Gaussian) distribution with a specified mean (loc) and standard deviation (scale).

3. Shuffling and Choosing

- rng.shuffle(arr): Modifies an array in-place by shuffling its elements.
- rng.choice(arr, size=None, replace=True): Returns a random sample from a given 1D array. replace=False ensures that each item is chosen only once.

Code Example

Generated python

```
    import numpy as np

# 1. Create a random number generator instance with a seed for reproducibility
rng = np.random.default_rng(seed=42)

# --- 2. Generating random floats ---
# Generate a 2x3 array of random floats between 0 and 1
random_floats = rng.random((2, 3))
print(f"--- Random Floats [0, 1) ---\n{random_floats}\n")

# --- 3. Generating random integers ---
# Generate 5 random integers between 10 (inclusive) and 20 (exclusive)
random_integers = rng.integers(low=10, high=20, size=5)
print(f"--- Random Integers ---\n{random_integers}\n")

# --- 4. Generating from a Normal Distribution ---
# Generate a 2x4 array from a normal distribution with mean=50, std_dev=5
normal_data = rng.normal(loc=50, scale=5, size=(2, 4))
print(f"--- Normal Distribution Data ---\n{normal_data}\n")

# --- 5. Shuffling an array ---
original_array = np.arange(10)
print(f"--- Shuffling ---")
print(f"Original array: {original_array}")
rng.shuffle(original_array) # Shuffles in-place
print(f"Shuffled array: {original_array}\n")

# --- 6. Random Choice ---
options = np.array(['cat', 'dog', 'fish', 'bird'])
print(f"--- Random Choice ---")
# Choose 3 items without replacement
choices = rng.choice(options, size=3, replace=False)
print(f"Random choices (no replacement): {choices}")
```

IGNORE_WHEN_COPYING_START
content_copy download

Use code with caution. Python
IGNORE_WHEN_COPYING_END

Question 11

How do you normalize an array in NumPy?

Theory

Normalization is a common preprocessing step that rescales the values in a feature to a standard range. The most common type of normalization is **Min-Max Scaling**, which scales the data to the range [0, 1].

This is important for algorithms that are sensitive to the magnitude of features. The formula is:
x_scaled = (x - min(x)) / (max(x) - min(x))

Code Example

Generated python

```
import numpy as np

def normalize_array(arr):
    """
    Normalizes a NumPy array to the range [0, 1] using Min-Max scaling.
    """
    min_val = arr.min()
    max_val = arr.max()

    # Avoid division by zero if all values are the same
    if max_val == min_val:
        return np.zeros_like(arr)

    return (arr - min_val) / (max_val - min_val)

# Create a sample array
data = np.array([10, 20, 30, 40, 50], dtype=float)

# Normalize it
normalized_data = normalize_array(data)

print(f"Original data: {data}")
print(f"Normalized data: {normalized_data}")
```

IGNORE_WHEN_COPYING_START
content_copy download

Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code
```
    Original data: [10. 20. 30. 40. 50.]
Normalized data: [0.   0.25 0.5  0.75 1.  ]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Important Note: Normalizing in a Machine Learning Context

In a real ML workflow, you must **fit** the scaler **only on the training data** and then use that same scaler to transform both the training and test data. This prevents data leakage.

Generated python
```python
    from sklearn.preprocessing import MinMaxScaler

# Create sample train and test data
train_data = np.array([[10], [20], [30], [40], [50]])
test_data = np.array([[5], [25], [55]]) # Test data outside the original range

# 1. Initialize the scaler
scaler = MinMaxScaler()

# 2. Fit the scaler ONLY on the training data
scaler.fit(train_data)

# 3. Transform both train and test data
train_normalized = scaler.transform(train_data)
test_normalized = scaler.transform(test_data)

print("--- Using Scikit-learn's MinMaxScaler ---")
print(f"Normalized training data:\n{train_normalized.flatten()}")
print(f"Normalized test data:\n{test_normalized.flatten()}")
# Note: Test data can have values outside [0, 1], which is correct behavior.
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Question 12

How can you compute percentiles with NumPy?

Theory

A **percentile** is a measure used in statistics indicating the value below which a given percentage of observations in a group of observations falls. For example, the 25th percentile is the value below which 25% of the data may be found. The 50th percentile is the **median**.

NumPy's np.percentile() function is the primary tool for this calculation.

numpy.percentile()

- **a**: The input array.
- **q**: The percentile or sequence of percentiles to compute, which must be between 0 and 100 inclusive.
- **axis**: The axis along which to compute the percentile.

Code Example

Generated python

```
    import numpy as np

data = np.array([0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100])

print(f"Data: {data}\n")

# --- 1. Calculate a single percentile ---
# Calculate the 50th percentile (the median)
p50 = np.percentile(data, 50)
print(f"50th Percentile (Median): {p50}")

# --- 2. Calculate multiple percentiles at once ---
# Calculate the quartiles: 25th, 50th, and 75th percentiles
quartiles = np.percentile(data, [25, 50, 75])
print(f"Quartiles (25th, 50th, 75th): {quartiles}\n")

# --- 3. Using with a 2D array ---
data_2d = np.array([[1, 2, 3],
                    [4, 5, 6],
                    [7, 8, 9]])

print(f"2D Data:\n{data_2d}\n")
```

```
# Calculate the 50th percentile along the columns (axis=0)
p50_cols = np.percentile(data_2d, 50, axis=0)
print(f"50th Percentile of each column (axis=0): {p50_cols}")

# Calculate the 50th percentile along the rows (axis=1)
p50_rows = np.percentile(data_2d, 50, axis=1)
print(f"50th Percentile of each row (axis=1): {p50_rows}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code

```
    Data: [  0  10  20  30  40  50  60  70  80  90 100]

50th Percentile (Median): 50.0
Quartiles (25th, 50th, 75th): [25. 50. 75.]

2D Data:
[[1 2 3]
 [4 5 6]
 [7 8 9]]

50th Percentile of each column (axis=0): [4. 5. 6.]
50th Percentile of each row (axis=1): [2. 5. 8.]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Use Cases

- **Exploratory Data Analysis**: Understanding the distribution of your data and identifying the range where most of your data lies.
- **Outlier Detection**: The Interquartile Range (IQR), calculated as 75th percentile - 25th percentile, is a robust measure used to identify outliers.
- **Setting Thresholds**: Percentiles are often used to define thresholds in a data-driven way.

Question 13

How do you calculate the correlation coefficient using NumPy?

Theory

The **correlation coefficient** (specifically, the Pearson correlation coefficient) is a measure of the linear correlation between two sets of data. It is a value between -1 and +1.

- **+1**: Total positive linear correlation.
- **0**: No linear correlation.
- **-1**: Total negative linear correlation.

NumPy's np.corrcoef() function is used to compute this value. It returns a **correlation matrix**.

numpy.corrcoef()

- **Input**: Can take two 1D arrays as separate arguments, or a 2D array where each **row** represents a variable.
- **Output**: A 2D correlation matrix. For two variables, this will be a 2x2 matrix.
    - matrix[0, 0]: Correlation of the first variable with itself (always 1).
    - matrix[1, 1]: Correlation of the second variable with itself (always 1).
    - matrix[0, 1] and matrix[1, 0]: The correlation between the two variables.
-

Code Example

Generated python

```
    import numpy as np

# --- 1. Correlation between two 1D arrays ---
x = np.array([1, 2, 3, 4, 5])
y_positive = np.array([2, 4, 6, 8, 10]) # Perfect positive correlation
y_negative = np.array([10, 8, 6, 4, 2]) # Perfect negative correlation
y_none = np.array([5, -2, 3, 1, -4])   # No obvious correlation

# Calculate the correlation matrices
corr_matrix_pos = np.corrcoef(x, y_positive)
corr_matrix_neg = np.corrcoef(x, y_negative)
corr_matrix_none = np.corrcoef(x, y_none)

print("--- Correlation between two variables ---")
print(f"Correlation (positive): {corr_matrix_pos[0, 1]:.2f}")
print(f"Correlation (negative): {corr_matrix_neg[0, 1]:.2f}")
print(f"Correlation (none): {corr_matrix_none[0, 1]:.2f}\n")
```

```
# --- 2. Correlation matrix for a 2D array ---
# Each ROW is a variable
data = np.array([[1, 2, 3, 4],    # Variable 1
                 [2, 4, 6, 8],    # Variable 2 (correlated with 1)
                 [10, 5, 2, 1]])  # Variable 3 (anti-correlated with 1)

corr_matrix_full = np.corrcoef(data)

print("--- Correlation matrix for multiple variables ---")
print("Correlation Matrix:")
print(np.round(corr_matrix_full, 2))
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code
```
    --- Correlation between two variables ---
Correlation (positive): 1.00
Correlation (negative): -1.00
Correlation (none): -0.42

--- Correlation matrix for multiple variables ---
Correlation Matrix:
[[ 1.    1.   -0.98]
 [ 1.    1.   -0.98]
 [-0.98 -0.98  1.  ]]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Interpretation

- The first part shows the expected results: perfect positive (1.0), perfect negative (-1.0), and low correlation for the random data.
- The second part produces a 3x3 correlation matrix. You can see that variable 1 and variable 2 have a correlation of 1.0, and variable 1 and variable 3 have a correlation of -0.98 (strong negative correlation), as expected. This matrix is essential in **Exploratory Data Analysis** to identify multicollinearity between features.

Question 14

Explain the use of the np.cumsum() and np.cumprod() functions.

Theory

cumsum (cumulative sum) and cumprod (cumulative product) are useful NumPy functions for computing cumulative statistics along an array.

- **np.cumsum()**: Returns an array of the same shape where each element at index i is the sum of all elements up to and including index i from the original array.
- **np.cumprod()**: Returns an array of the same shape where each element at index i is the product of all elements up to and including index i.

Both functions can also operate along a specific axis of a multi-dimensional array.

Code Example

Generated python

```
    import numpy as np

# --- 1. 1D Array Example ---
arr_1d = np.array([1, 2, 3, 4, 5])
print(f"Original 1D array: {arr_1d}\n")

# Cumulative Sum
cumsum_1d = np.cumsum(arr_1d)
print(f"Cumulative Sum (cumsum): {cumsum_1d}")
# [1, 1+2, 1+2+3, 1+2+3+4, 1+2+3+4+5]

# Cumulative Product
cumprod_1d = np.cumprod(arr_1d)
print(f"Cumulative Product (cumprod): {cumprod_1d}\n")
# [1, 1*2, 1*2*3, 1*2*3*4, 1*2*3*4*5]

# --- 2. 2D Array Example ---
arr_2d = np.array([[1, 2, 3],
                   [4, 5, 6]])
print(f"Original 2D array:\n{arr_2d}\n")

# Cumsum along columns (axis=0)
cumsum_axis0 = np.cumsum(arr_2d, axis=0)
print(f"Cumsum along columns (axis=0):\n{cumsum_axis0}\n")
# [[1, 2, 3], [1+4, 2+5, 3+6]]
# Cumsum along rows (axis=1)
cumsum_axis1 = np.cumsum(arr_2d, axis=1)
print(f"Cumsum along rows (axis=1):\n{cumsum_axis1}\n")
# [[1, 1+2, 1+2+3], [4, 4+5, 4+5+6]]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Use Cases

- **Simulating Random Walks**: A random walk can be simulated by taking the cumulative sum of a series of random steps.
- **Calculating Compound Growth**: The cumulative product can be used to calculate the value of an investment over time given a series of daily or monthly returns. cumulative_return = np.cumprod(1 + daily_returns)
- **Data Analysis**: Calculating running totals or products in a dataset.

Question 15

How do you stack multiple arrays vertically and horizontally?

Theory

Stacking is the process of joining a sequence of arrays along a new axis. NumPy provides several convenience functions for this, with np.vstack() (vertical stack) and np.hstack() (horizontal stack) being the most common.

- **Vertical Stacking (np.vstack)**: Stacks arrays on top of each other (row-wise). The number of **columns** must be the same for all arrays. This is equivalent to np.concatenate with axis=0.
- **Horizontal Stacking (np.hstack)**: Stacks arrays side-by-side (column-wise). The number of **rows** must be the same for all arrays. This is equivalent to np.concatenate with axis=1.

Code Example

Generated python

```
    import numpy as np

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

```
b = np.array([[5, 6],
              [7, 8]])

c = np.array([9, 10]) # A 1D array

print(f"Array a:\n{a}")
print(f"Array b:\n{b}")
print(f"Array c: {c}\n")

# --- 1. Vertical Stacking ---
# Stacking two 2D arrays
v_stacked_ab = np.vstack((a, b))
print(f"--- Vertical Stacking ---")
print(f"vstack of a and b:\n{v_stacked_ab}\n")

# Stacking a 2D array and a 1D array
# The 1D array is automatically promoted to 2D
v_stacked_ac = np.vstack((a, c))
print(f"vstack of a and c:\n{v_stacked_ac}\n")


# --- 2. Horizontal Stacking ---
# Stacking two 2D arrays
h_stacked_ab = np.hstack((a, b))
print(f"--- Horizontal Stacking ---")
print(f"hstack of a and b:\n{h_stacked_ab}\n")

# Stacking a 2D array and a 1D array
# For hstack, c must be reshaped into a column vector to match dimensions
c_col = c.reshape(2, 1)
h_stacked_ac = np.hstack((a, c_col))
print(f"hstack of a and c (reshaped):\n{h_stacked_ac}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code

```
    Array a:
[[1 2]
 [3 4]]
Array b:
```

```
[[5 6]
 [7 8]]
Array c: [ 9 10]

--- Vertical Stacking ---
vstack of a and b:
[[1 2]
 [3 4]
 [5 6]
 [7 8]]

vstack of a and c:
[[ 1  2]
 [ 3  4]
 [ 9 10]]

--- Horizontal Stacking ---
hstack of a and b:
[[1 2 5 6]
 [3 4 7 8]]

hstack of a and c (reshaped):
[[ 1  2  9]
 [ 3  4 10]]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Question 16

Describe the process for creating a structured array in NumPy.

Theory

A **structured array** in NumPy is an array where each element is a record composed of named fields, each with its own data type. This is useful for storing heterogeneous data, similar to a table.

The process involves two main steps:

1. **Define the dtype**: Create a custom data type that specifies the name and dtype for each field in the record.

2. **Create the Array**: Create the NumPy array, passing the data and the custom dtype.

Code Example

Generated python
```
    import numpy as np

# --- 1. Define the data ---
# Data can be a list of tuples, where each tuple is a record
data = [('Alice', 25, 65.5),
        ('Bob', 32, 80.2),
        ('Charlie', 28, 75.0)]

# --- 2. Define the dtype ---
# Method A: List of tuples ('field_name', 'type_code')
# 'U10' -> Unicode string, max 10 chars
# 'i4' -> 4-byte (32-bit) integer
# 'f4' -> 4-byte (32-bit) float
my_dtype = [('name', 'U10'), ('age', 'i4'), ('weight', 'f4')]

# --- 3. Create the structured array ---
structured_array = np.array(data, dtype=my_dtype)

print("--- The Structured Array ---")
print(structured_array)
print(f"\nDtype of the array: {structured_array.dtype}")

# --- 4. Accessing the data ---
print("\n--- Accessing Data ---")
# Access the 'name' field for all records
print(f"All names: {structured_array['name']}")

# Access the first record
first_record = structured_array[0]
print(f"First record: {first_record}")

# Access the 'age' field of the first record
print(f"Age of first person: {first_record['age']}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code

```
    --- The Structured Array ---
[('Alice', 25, 65.5) ('Bob', 32, 80.2) ('Charlie', 28, 75. )]

Dtype of the array: [('name', '<U10'), ('age', '<i4'), ('weight', '<f4')]

--- Accessing Data ---
All names: ['Alice' 'Bob' 'Charlie']
First record: ('Alice', 25, 65.5)
Age of first person: 25
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Question 17

How do you save and load NumPy arrays to and from disk?

Theory

Saving and loading NumPy arrays is essential for model persistence and data storage. NumPy provides several simple and efficient functions for this.

Key Methods

1. **np.save() and np.load() (for single arrays)**:
   - **Format**: Saves a single array to a binary file with a .npy extension. This format is efficient and preserves the array's shape, dtype, and other metadata.
   - This is the standard, recommended way to save a single NumPy array.
2.
3. **np.savez() and np.load() (for multiple arrays)**:
   - **Format**: Saves multiple arrays into a single, uncompressed zip archive with a .npz extension. You pass the arrays as keyword arguments, which then become the keys for accessing the arrays upon loading.
4.
5. **np.savez_compressed()**:
   - **Format**: The same as np.savez(), but it saves the arrays in a **compressed** zip archive. This is useful for large arrays where disk space is a concern.
6.
7. **np.savetxt() and np.loadtxt() (for text files)**:
   - **Format**: Saves an array to a human-readable text file (like a .csv or .txt).

    - **Limitations**: This is much less efficient in terms of both speed and disk space. It can only be used for 1D and 2D arrays and loses some metadata like the dtype. It's mainly useful for exporting data to be read by other non-Python programs.
8.

Code Example

Generated python

```
    import numpy as np
import os

# Create some sample arrays
a = np.arange(10)
b = np.array([[1, 2, 3], [4, 5, 6]])

# --- 1. Save and load a single array ---
print("--- Saving/Loading a single array (.npy) ---")
np.save('array_a.npy', a)
loaded_a = np.load('array_a.npy')
print(f"Loaded array a: {loaded_a}\n")

# --- 2. Save and load multiple arrays ---
print("--- Saving/Loading multiple arrays (.npz) ---")
np.savez('arrays_ab.npz', array1=a, array2=b)

# Load the .npz file
data_archive = np.load('arrays_ab.npz')
print(f"Keys in the archive: {list(data_archive.keys())}")
loaded_a_from_npz = data_archive['array1']
loaded_b_from_npz = data_archive['array2']
print(f"Loaded array1: {loaded_a_from_npz}")
print(f"Loaded array2:\n{loaded_b_from_npz}\n")
data_archive.close()

# --- 3. Save to a text file ---
print("--- Saving to a text file (.txt) ---")
np.savetxt('array_b.txt', b, delimiter=',')
loaded_b_from_txt = np.loadtxt('array_b.txt', delimiter=',')
print(f"Loaded from text file:\n{loaded_b_from_txt}")

# --- Clean up the created files ---
os.remove('array_a.npy')
os.remove('arrays_ab.npz')
os.remove('array_b.txt')
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Question 18

Write a NumPy code to create a 3x3 identity matrix.

Theory

An **identity matrix** is a square matrix with ones on the main diagonal and zeros everywhere else. It is the matrix equivalent of the number 1.

NumPy provides two main functions for this:

- np.identity(n): Creates an n x n identity matrix.
- np.eye(N, M=None): More general. Creates a 2D array with ones on the k-th diagonal and zeros elsewhere. If M is not specified, it defaults to N, creating a square matrix.

For creating a simple identity matrix, both are fine, but np.identity() is slightly more direct.

Code Example

Generated python
```
    import numpy as np

# --- Method 1: Using np.identity() (Preferred) ---
n = 3
identity_matrix_1 = np.identity(n, dtype=int)
print(f"--- Using np.identity({n}) ---")
print(identity_matrix_1)

# --- Method 2: Using np.eye() ---
identity_matrix_2 = np.eye(3, dtype=int)
print(f"\n--- Using np.eye(3) ---")
print(identity_matrix_2)
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code

```
    --- Using np.identity(3) ---
[[1 0 0]
 [0 1 0]
 [0 0 1]]

--- Using np.eye(3) ---
[[1 0 0]
 [0 1 0]
 [0 0 1]]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Question 19

Code a function in NumPy to compute the moving average of a 1D array.

Theory

A **moving average** (or rolling mean) is a common technique used to smooth out short-term fluctuations in time series data and highlight longer-term trends. It is calculated by taking the average of the data in a sliding window of a fixed size.

A simple and efficient way to implement this in NumPy is to use the np.convolve() function. By convolving the array with a window of ones and then dividing by the window size, we can compute the moving average in a single, vectorized operation.

Code Example

Generated python

```python
    import numpy as np

def moving_average(a, window_size):
    """
    Computes the moving average of a 1D array.

    Args:
        a (np.ndarray): The input 1D array.
        window_size (int): The size of the sliding window.

    Returns:
        np.ndarray: The array of moving averages.
    """
    # Create a window of weights (all ones)
    weights = np.ones(window_size) / window_size
    # Use 'valid' mode to only return results where the window is fully overlapping
    return np.convolve(a, weights, mode='valid')

# Create some sample data
data = np.array([10, 20, 30, 40, 50, 60, 70, 80, 90, 100])
window = 3

# Calculate the moving average
ma = moving_average(data, window)

print(f"Original Data: {data}")
print(f"Window Size: {window}")
print(f"Moving Average: {ma}")
# Expected: [(10+20+30)/3, (20+30+40)/3, ...]
# [20., 30., 40., 50., 60., 70., 80., 90.]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Question 20

Generate a 2D NumPy array of random integers and normalize it between 0 and 1.

Theory

This is a two-step process:

1. **Generate Data**: Use NumPy's random number generator (np.random.default_rng()) and its integers() method to create a 2D array of random integers within a specified range.
2. **Normalize**: Apply Min-Max scaling to the generated array to rescale its values to the range [0, 1].

Code Example

Generated python
```
    import numpy as np

# 1. Generate a 4x5 array of random integers between 10 and 100
rng = np.random.default_rng(42)
low = 10
```

```
high = 101 # Exclusive, so it will go up to 100
shape = (4, 5)
random_integers = rng.integers(low=low, high=high, size=shape)

print("--- Original Random Integer Array ---")
print(random_integers)

# 2. Normalize the array using Min-Max scaling
min_val = random_integers.min()
max_val = random_integers.max()
normalized_array = (random_integers - min_val) / (max_val - min_val)

print("\n--- Normalized Array [0, 1] ---")
print(np.round(normalized_array, 3))

# Verify the new min and max
print(f"\nNew Minimum: {normalized_array.min()}")
print(f"New Maximum: {normalized_array.max()}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Question 21

Create a NumPy code snippet to extract all odd numbers from an array.

Theory

This is a classic filtering problem that can be solved elegantly and efficiently using **boolean indexing**. We create a boolean mask where the condition is that the number is odd (arr % 2 != 0) and use it to select elements from the original array.

Code Example

Generated python

```
    import numpy as np

# Create an array of integers
arr = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# Use boolean indexing to extract odd numbers
odd_numbers = arr[arr % 2 != 0]
```

```
print(f"Original Array: {arr}")
print(f"Odd Numbers: {odd_numbers}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code

```
    Original Array: [ 0  1  2  3  4  5  6  7  8  9 10]
Odd Numbers: [1 3 5 7 9]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Question 22

Implement a routine to calculate the outer product of two vectors in NumPy.

Theory

The **outer product** of two vectors u (size m) and v (size n) results in a matrix A (size m x n) where each element A[i, j] is the product of u[i] and v[j].

NumPy's np.outer() function is designed specifically for this. Alternatively, you can achieve the same result by reshaping the vectors and using broadcasting with multiplication.

Code Example

Generated python

```
    import numpy as np

u = np.array([1, 2, 3])
v = np.array([10, 20, 30, 40])

print(f"Vector u: {u}")
print(f"Vector v: {v}\n")

# --- Method 1: Using np.outer() (Preferred) ---
outer_product_1 = np.outer(u, v)
print("--- Using np.outer() ---")
print(outer_product_1)

# --- Method 2: Using broadcasting ---
# Reshape u to a column vector (3, 1) and v to a row vector (1, 4)
# Broadcasting will then compute the element-wise product
outer_product_2 = u[:, np.newaxis] * v
print("\n--- Using broadcasting ---")
print(outer_product_2)
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code

```
    Vector u: [1 2 3]
Vector v: [10 20 30 40]

--- Using np.outer() ---
[[ 10  20  30  40]
 [ 20  40  60  80]
 [ 30  60  90 120]]

--- Using broadcasting ---
[[ 10  20  30  40]
 [ 20  40  60  80]
 [ 30  60  90 120]]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Question 23

Write a NumPy program to create a checkerboard 8x8 matrix using the tile function.

Theory

A checkerboard pattern consists of alternating values (e.g., 0 and 1). We can create a small base pattern (like [[0, 1], [1, 0]]) and then use the np.tile() function to repeat this pattern to build the larger 8x8 matrix.

np.tile(A, reps): Constructs an array by repeating A the number of times given by reps.

Code Example

Generated python

```
    import numpy as np

# Create the base 2x2 pattern
base_pattern = np.array([[0, 1],
                         [1, 0]])

# Tile the pattern 4 times in each direction (4*2=8) to create an 8x8 board
checkerboard = np.tile(base_pattern, (4, 4))

print("--- 8x8 Checkerboard Matrix ---")
print(checkerboard)
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Question 24

Code a NumPy snippet to create a border around an existing array.

Theory

This can be achieved efficiently using the np.pad() function. We can specify a constant padding mode and add a border of size 1 around the array.

Code Example

Generated python

```
    import numpy as np

# Create an existing 3x3 array of ones
existing_array = np.ones((3, 3), dtype=int)
print("--- Existing Array ---")
print(existing_array)

# Add a border of 1 pixel width with a constant value of 0
bordered_array = np.pad(existing_array, pad_width=1, mode='constant', constant_values=0)

print("\n--- Array with Border ---")
print(bordered_array)
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Question 25

Write a function to compute the convolution of two matrices in NumPy.

Theory

2D convolution is a fundamental operation in image processing and CNNs. It involves sliding a small matrix (the **kernel**) over a larger matrix (the **input**) and computing the sum of element-wise products at each position.

While you could implement this with nested loops, a more efficient approach for this specific case is to use scipy.signal.convolve2d, as NumPy's np.convolve is primarily for 1D.

Code Example

Generated python

```
    import numpy as np
from scipy.signal import convolve2d

def convolution_2d(input_matrix, kernel):
    """
    Computes the 2D convolution of a matrix and a kernel.
    """
    return convolve2d(input_matrix, kernel, mode='valid')

# Define an input matrix and a kernel
input_matrix = np.array([[1, 2, 3, 4],
                         [5, 6, 7, 8],
                         [9, 10, 11, 12],
                         [13, 14, 15, 16]])

kernel = np.array([[1, 0, -1],
                   [1, 0, -1],
                   [1, 0, -1]]) # A simple vertical edge detection kernel

# Compute the convolution
convolved_matrix = convolution_2d(input_matrix, kernel)

print("--- Input Matrix ---")
print(input_matrix)
print("\n--- Kernel ---")
print(kernel)
print("\n--- Convolved Matrix ('valid' mode) ---")
print(convolved_matrix)
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

The 'valid' mode means the output only contains pixels where the kernel and input fully overlap.

Question 26

Implement a script that computes the Fibonacci sequence using a NumPy matrix.

Theory

The Fibonacci sequence has a linear recurrence relationship that can be expressed in matrix form.
The relationship [F(n+1), F(n)] = [F(n) + F(n-1), F(n)] can be represented by the matrix transformation:
[[F(n+1)], [F(n)]] = [[1, 1], [1, 0]] @ [[F(n)], [F(n-1)]]

By raising the transformation matrix [[1, 1], [1, 0]] to the n-th power, we can compute the n-th Fibonacci number very quickly.

Code Example

Generated python

```
    import numpy as np

def fibonacci_matrix(n):
    """
    Computes the n-th Fibonacci number using matrix exponentiation.
    """
    if n == 0:
        return 0

    # Define the transformation matrix
    F = np.array([[1, 1],
                  [1, 0]], dtype=object) # Use dtype=object for large numbers
    # Use matrix exponentiation to find F^(n-1)
    F_n_minus_1 = np.linalg.matrix_power(F, n - 1)

    # The n-th Fibonacci number is in the top-left corner
    return F_n_minus_1[0, 0]

# Compute the 10th Fibonacci number
n = 10
fib_n = fibonacci_matrix(n)

print(f"The {n}-th Fibonacci number is: {fib_n}") # Should be 55

# Compute a large Fibonacci number
n_large = 80
fib_n_large = fibonacci_matrix(n_large)
print(f"The {n_large}-th Fibonacci number is: {fib_n_large}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

This matrix exponentiation method is computationally much more efficient for finding large Fibonacci numbers than the standard recursive approach.

Question 27

Write a code to replace all elements greater than a certain threshold in a NumPy array with a specific value.

Theory

This task is a perfect use case for **boolean indexing**. We can create a boolean mask to identify the elements that satisfy the condition and then use this mask to assign a new value to those specific elements.

Code Example

Generated python

```
 import numpy as np

# Create an array
arr = np.array([[10, 55, 23],
            [78, 42, 91],
              [31, 65, 15]])

threshold = 50
replacement_value = -1

print(f"Original Array:\n{arr}\n")

# --- Method 1: Boolean Indexing (In-place) ---
arr_copy = arr.copy()
arr_copy[arr_copy > threshold] = replacement_value
print(f"--- Using Boolean Indexing ---")
print(f"Array after replacement:\n{arr_copy}\n")

# --- Method 2: Using np.where() (Returns a new array) ---
arr_replaced_where = np.where(arr > threshold, replacement_value, arr)
print(f"--- Using np.where ---")
print(f"Array after replacement:\n{arr_replaced_where}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

np.where is often preferred when you want to create a new array without modifying the original. Boolean indexing is great for fast, in-place modifications.

Question 28

Implement an efficient rolling window calculation for a 1D array using NumPy.

Theory

This can be done very efficiently without any Python loops by using the **stride tricks** of NumPy. numpy.lib.stride_tricks.sliding_window_view is the modern, safe way to create a view of the array that represents the rolling windows. Once this view is created, you can apply an aggregate function (like .mean()) along the last axis.

Code Example

Generated python
```
    import numpy as np
from numpy.lib.stride_tricks import sliding_window_view

# Create a 1D array
```

```
data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
window_size = 3

print(f"Original Data: {data}")
print(f"Window Size: {window_size}\n")

# 1. Create the rolling window view
# This is a zero-copy, memory-efficient operation
window_view = sliding_window_view(data, window_shape=window_size)
print("--- Rolling Window View ---")
print(window_view)

# 2. Calculate the rolling mean
# We take the mean along the last axis (axis=1) of the view
rolling_mean = window_view.mean(axis=1)
print("\n--- Rolling Mean ---")
print(np.round(rolling_mean, 2)) # [2. 3. 4. 5. 6. 7. 8. 9.]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

This is the most performant way to compute rolling statistics in NumPy as it avoids data duplication and leverages vectorized calculations.

Question 29

Explain how you would implement gradient descent optimization with NumPy.

Theory

Gradient descent is an iterative optimization algorithm used to minimize a function. In a machine learning context, we use it to minimize a model's **loss function** by finding the optimal parameters.

The implementation in NumPy involves:

1. Representing model parameters as a NumPy vector (theta).
2. Representing the dataset as a NumPy matrix X and vector y.
3. In a loop, calculating the gradient of the loss function with respect to the parameters using vectorized NumPy operations.
4. Updating the parameter vector using the formula: theta = theta - learning_rate * gradient.

Code Example (for Logistic Regression)

Generated python

```
 import numpy as np

# Sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def gradient_descent_logistic(X, y, learning_rate=0.1, epochs=1000):
    # Add intercept term to X
    X_b = np.c_[np.ones((X.shape[0], 1)), X]
    n_samples, n_features = X_b.shape

    # 1. Initialize parameters
    theta = np.zeros(n_features)

    for _ in range(epochs):
        # Calculate predictions (hypothesis)
        z = X_b @ theta
        h = sigmoid(z)

        # 3. Calculate the gradient using vectorized operations
        gradient = (1 / n_samples) * X_b.T @ (h - y)

        # 4. Update parameters
        theta -= learning_rate * gradient

    return theta

# --- Example Usage ---
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=100, n_features=2, n_informative=2, n_redundant=0,
random_state=42)

# Run gradient descent
optimal_params = gradient_descent_logistic(X, y)

print("--- Optimal Parameters Found ---")
print(f"Intercept: {optimal_params[0]:.4f}")
print(f"Coefficients: {optimal_params[1:]}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python

IGNORE_WHEN_COPYING_END

This code demonstrates how to use NumPy's powerful matrix operations (@ for dot product, .T for transpose) to compute the gradient for the entire dataset in a single, efficient, vectorized step, which is the core of implementing gradient-based optimizers in NumPy.

Question 1

How do you inspect the shape and size of a NumPy array?

Theory

Inspecting the shape and size of a NumPy array is a fundamental operation, crucial for debugging, understanding your data's dimensions, and ensuring compatibility between arrays for operations like matrix multiplication. NumPy provides several straightforward attributes for this purpose.

Key Attributes

Let arr be a NumPy ndarray.

1. **arr.shape**:
    - **What it is**: A **tuple** of integers representing the dimensions of the array. The length of the tuple is the number of dimensions (ndim).
    - **Example**: For a 2D array (matrix) with 3 rows and 5 columns, arr.shape will be (3, 5).
2.
3. **arr.size**:
    - **What it is**: An **integer** representing the total number of elements in the array.
    - **Example**: For the (3, 5) array, arr.size will be 3 * 5 = 15.
4.
5. **arr.ndim**:
    - **What it is**: An **integer** representing the number of axes or dimensions of the array.
    - **Example**: For the (3, 5) array, arr.ndim will be 2. For a vector, it would be 1.
6.

Code Example

Generated python
```
    import numpy as np

# Create a 3D array
# (2 layers, 3 rows, 4 columns)
arr_3d = np.arange(24).reshape(2, 3, 4)
```

```
print("--- Array Information ---")
print(f"The array:\n{arr_3d}\n")

# --- Inspecting the attributes ---

# 1. Shape
shape_tuple = arr_3d.shape
print(f"Shape of the array (arr.shape): {shape_tuple}")
print(f" - Number of layers: {shape_tuple[0]}")
print(f" - Number of rows: {shape_tuple[1]}")
print(f" - Number of columns: {shape_tuple[2]}")

# 2. Size
total_elements = arr_3d.size
print(f"\nTotal number of elements (arr.size): {total_elements}")

# 3. Number of dimensions
num_dimensions = arr_3d.ndim
print(f"\nNumber of dimensions (arr.ndim): {num_dimensions}")

# Verification
# size should be the product of the shape elements
assert total_elements == np.prod(shape_tuple)
# ndim should be the length of the shape tuple
assert num_dimensions == len(shape_tuple)
```

Output:

```
Generated code
    --- Array Information ---
The array:
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]

Shape of the array (arr.shape): (2, 3, 4)
 - Number of layers: 2
 - Number of rows: 3
 - Number of columns: 4
```

Total number of elements (arr.size): 24

Number of dimensions (arr.ndim): 3

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Question 2

How do you perform element-wise operations in NumPy?

Theory

Element-wise operations are operations that are applied independently to each element in an array or between corresponding elements of multiple arrays. This is a core concept of **vectorization** in NumPy and is the reason for its high performance.

Instead of writing a Python for loop, you can use standard arithmetic operators or NumPy's universal functions (ufuncs) to perform these operations on entire arrays at once.

Key Methods

1. **Using Standard Arithmetic Operators**:
    - The standard Python arithmetic operators (+, -, *, /, **, etc.) are overloaded in NumPy to perform element-wise operations.
    - These operations support **broadcasting**, allowing you to combine arrays of different but compatible shapes.
2.
3. **Using NumPy Universal Functions (ufuncs)**:
    - NumPy provides a rich library of ufuncs that perform element-wise operations. For every arithmetic operator, there is a corresponding ufunc (e.g., + corresponds to np.add()).
    - This category also includes a vast range of other functions like np.sqrt(), np.exp(), np.sin(), and np.log().
4.

Code Example

Generated python

```
    import numpy as np

# Create two arrays of the same shape
a = np.array([[1, 2],
            [3, 4]])
b = np.array([[10, 20],
            [30, 40]])

print(f"Array a:\n{a}")
print(f"Array b:\n{b}\n")

# --- Element-wise operations with operators ---
print("--- Using Operators ---")
# Element-wise addition
add_result = a + b
print(f"a + b:\n{add_result}\n")

# Element-wise multiplication
mul_result = a * b
print(f"a * b:\n{mul_result}\n")

# --- Element-wise operations with ufuncs ---
print("--- Using Ufuncs ---")
# Element-wise subtraction using the ufunc
sub_result = np.subtract(b, a)
print(f"np.subtract(b, a):\n{sub_result}\n")

# Element-wise square root on a single array
sqrt_result = np.sqrt(a)
print(f"np.sqrt(a):\n{sqrt_result}\n")

# --- Operations with a scalar ---
# The scalar is broadcast to every element of the array
scalar_result = a * 10
print("--- Operation with a Scalar ---")
print(f"a * 10:\n{scalar_result}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code

```
    Array a:
[[1 2]
 [3 4]]
Array b:
```

```
[[10 20]
 [30 40]]

--- Using Operators ---
a + b:
[[11 22]
 [33 44]]

a * b:
[[ 10  40]
 [ 90 160]]

--- Using Ufuncs ---
np.subtract(b, a):
[[ 9 18]
 [27 36]]

np.sqrt(a):
[[1.         1.41421356]
 [1.73205081 2.        ]]

--- Operation with a Scalar ---
a * 10:
[[10 20]
 [30 40]]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

This demonstrates how simple and readable it is to perform complex mathematical operations on entire datasets without writing a single explicit loop in Python.

Question 3

How do you compute the mean, median, and standard deviation with NumPy?

Theory

NumPy provides simple, highly optimized functions for computing the most common descriptive statistics of an array. These functions can be called either as top-level NumPy functions (e.g., np.mean(arr)) or as methods of the ndarray object itself (e.g., arr.mean()).

These functions can also be applied along a specific **axis** of a multi-dimensional array to compute the statistic for each row or column.

Key Functions

1. **Mean (np.mean() or .mean())**:
    - **Description**: Calculates the arithmetic average of the elements.
2.
3. **Median (np.median())**:
    - **Description**: Calculates the median (the middle value) of the elements. The median is a more robust measure of central tendency than the mean, as it is not affected by outliers.
4.
5. **Standard Deviation (np.std() or .std())**:
    - **Description**: Calculates the standard deviation, which is a measure of the amount of variation or dispersion of the data. It is the square root of the variance.
6.

Code Example

Generated python

```
    import numpy as np

# Create a 2D array
data = np.array([[1, 2, 3, 4],
             [5, 6, 7, 8],
             [9, 10, 11, 50]]) # Added an outlier (50)

print(f"Data:\n{data}\n")

# --- 1. Compute statistics for the entire array (flattened) ---
print("--- Overall Statistics ---")
mean_total = np.mean(data)
median_total = np.median(data)
std_total = np.std(data)

print(f"Mean: {mean_total:.2f}")
print(f"Median: {median_total:.2f}  <-- Note: Median is less affected by the outlier 50")
print(f"Standard Deviation: {std_total:.2f}\n")

# --- 2. Compute statistics along a specific axis ---
print("--- Axis-wise Statistics ---")
# axis=0 -> compute down the columns
mean_cols = data.mean(axis=0)
print(f"Mean of each column (axis=0): {np.round(mean_cols, 2)}")
```

```
# axis=1 -> compute across the rows
median_rows = np.median(data, axis=1)
print(f"Median of each row (axis=1): {median_rows}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code
```
    Data:
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 50]]

--- Overall Statistics ---
Mean: 9.67
Median: 6.50  <-- Note: Median is less affected by the outlier 50
Standard Deviation: 12.37

--- Axis-wise Statistics ---
Mean of each column (axis=0): [ 5.   6.   7.  20.67]
Median of each row (axis=1): [ 2.5  6.5 10.5]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Interpretation of the Axis-wise Results

- **mean(axis=0)**: This collapses the rows and calculates the mean for each of the 4 columns independently, resulting in a 1D array of 4 values. This is a very common operation in machine learning for calculating the mean of each feature.
- **median(axis=1)**: This collapses the columns and calculates the median for each of the 3 rows independently, resulting in a 1D array of 3 values.

Question 4

Why is NumPy more efficient for numerical computations than pure Python?

Theory

NumPy's efficiency for numerical computations stems from a combination of its low-level implementation, its memory layout, and its support for vectorized operations. It effectively bridges the gap between Python's ease of use and the raw speed of compiled languages like C and Fortran.

Key Reasons for Efficiency

1. **Implementation in C and Fortran**:
    - **Python**: Python is an interpreted language, which means there is a lot of overhead in executing code line by line.
    - **NumPy**: The core of NumPy, including the ndarray object and its operations (ufuncs), is written in highly optimized, compiled C and Fortran code. When you perform a NumPy operation, the heavy computation is handed off to this fast, low-level code.
2.
3. **Vectorization**:
    - **Python**: To perform an operation on a collection of numbers, you typically need to use an explicit for loop. Each iteration of this loop involves overhead from the Python interpreter.
    - **NumPy**: NumPy allows you to perform operations on entire arrays at once. A single Python statement like a + b triggers a single, highly optimized loop in C that iterates over the data. This is known as **vectorization**.
4.
5. **Contiguous Memory Layout**:
    - **Python Lists**: A list stores a collection of pointers to Python objects that can be scattered all over memory. Accessing the data requires an extra step of following these pointers.
    - **NumPy Arrays**: An ndarray is a homogeneous data structure stored in a **contiguous block of memory**. This is crucial for performance. When the data is contiguous, the computer's CPU can make effective use of its **memory cache**, which dramatically speeds up data access. This layout is also ideal for passing data to low-level C or Fortran libraries.
6.
7. **Homogeneous Data Type**:
    - **Python Lists**: Can contain objects of different types. This flexibility comes at a cost: for every operation, Python has to perform a type check.
    - **NumPy Arrays**: Are homogeneous (all elements are of the same dtype). Since the type is known for the entire array, NumPy can use optimized, specialized C functions for that specific data type without any need for type checking inside the main loop.
8.

In summary, NumPy is more efficient because it moves the computationally intensive work from the slow, flexible Python world to the fast, specialized C/Fortran world. It achieves this through **vectorization**, which leverages its **contiguous and homogeneous memory layout** to execute operations in highly optimized, pre-compiled code.

Question 5

How do you check the memory size of a NumPy array?

Theory

Checking the memory size of a NumPy array is important for performance tuning and memory management, especially when working with large datasets. NumPy provides several attributes on the ndarray object to inspect its memory footprint precisely.

Key Attributes

Let arr be a NumPy ndarray.

1. **arr.nbytes**:
    - **Description**: This is the most direct and common method. It returns an integer representing the **total number of bytes** consumed by the elements of the array.
    - **Calculation**: It is equivalent to arr.size * arr.itemsize.
2.
3. **arr.itemsize**:
    - **Description**: Returns the size in **bytes of a single element** in the array. This is determined by the array's dtype.
    - **Example**: For dtype=float64, itemsize is 8. For dtype=int16, itemsize is 2.
4.
5. **arr.size**:
    - **Description**: Returns the total number of elements in the array. You can calculate the total memory by multiplying this by arr.itemsize.
6.

Code Example

Generated python

```
    import numpy as np
import sys

# Create a sample array with a specific dtype
# 4 rows, 5 columns, with 32-bit floating point numbers (4 bytes each)
arr = np.zeros((4, 5), dtype=np.float32)
```

```
# --- Using NumPy attributes ---
print("--- Using NumPy Array Attributes ---")

# itemsize: Size of one element
print(f"Size of one element (arr.itemsize): {arr.itemsize} bytes")

# size: Total number of elements
print(f"Total number of elements (arr.size): {arr.size}")

# nbytes: Total memory consumed by the data
print(f"Total memory of array data (arr.nbytes): {arr.nbytes} bytes")

# --- Verification ---
assert arr.nbytes == arr.size * arr.itemsize

# --- Using sys.getsizeof() (for comparison) ---
# Note: This is generally not the recommended way for NumPy arrays
print("\n--- Using sys.getsizeof() ---")
print(f"Memory reported by sys.getsizeof(arr): {sys.getsizeof(arr)} bytes")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code
```
    --- Using NumPy Array Attributes ---
Size of one element (arr.itemsize): 4 bytes
Total number of elements (arr.size): 20
Total memory of array data (arr.nbytes): 80 bytes

--- Using sys.getsizeof() ---
Memory reported by sys.getsizeof(arr): 192 bytes
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Important Distinction: arr.nbytes vs. sys.getsizeof()

- arr.nbytes: Gives you the size of the **actual data** in the array's buffer. This is usually what you care about when considering the memory footprint of your data.

- sys.getsizeof(arr): Gives you the size of the **Python object** that represents the array. This includes the size of the data buffer *plus* the overhead for the object's metadata (like shape, strides, dtype, etc.). This is why sys.getsizeof() reports a larger value.

For understanding the memory consumption of the data itself, **arr.nbytes** is the correct and preferred attribute to use.

Question 6

How do you create a record array in NumPy?

Theory

A **record array**, or more generally a **structured array**, is a special type of ndarray in NumPy that allows for storing **heterogeneous** data types, similar to a row in a SQL table or a struct in C.

Instead of each element being a single number, each element in a structured array is a **record**, and each record is composed of several named **fields**. Each field can have its own data type.

This is useful for representing tabular data directly within NumPy, although for general-purpose data manipulation, Pandas DataFrames are often more convenient.

How to Create One

There are several ways to create a structured array, but the most common is by defining a dtype that specifies the names and data types of each field.

The dtype can be specified as a list of tuples, where each tuple is ('field_name', 'data_type').

Code Example

Generated python

```
    import numpy as np

# --- 1. Define the data and the dtype ---
# Imagine we have data for a list of products
data = [('Laptop', 1, 1200.50),
        ('Mouse', 2, 25.00),
        ('Keyboard', 3, 75.99)]

# Define the structure of each record
# Field 1: 'name', a Unicode string of max 20 chars
# Field 2: 'id', a 32-bit integer
# Field 3: 'price', a 64-bit float
```

```
product_dtype = [('name', 'U20'), ('id', 'i4'), ('price', 'f8')]

# --- 2. Create the structured array ---
products = np.array(data, dtype=product_dtype)

print("--- The Structured Array ---")
print(products)
print(f"\nDtype of the array: {products.dtype}")
print(f"Shape of the array: {products.shape}")

# --- 3. Accessing Data ---
print("\n--- Accessing Data ---")

# Access a specific record (row) by index
print(f"First record: {products[0]}")

# Access a specific field (column) by its name
print(f"All product names: {products['name']}")
print(f"All prices: {products['price']}")

# Access a specific field of a specific record
print(f"Price of the second product: {products[1]['price']}")

# --- 4. Performing Operations ---
# We can perform vectorized operations on the fields
total_value = np.sum(products['price'])
print(f"\nTotal value of all products: {total_value:.2f}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code

```
    --- The Structured Array ---
[('Laptop', 1, 1200.5 ) ('Mouse', 2,   25.  ) ('Keyboard', 3,   75.99)]

Dtype of the array: [('name', '<U20'), ('id', '<i4'), ('price', '<f8')]
Shape of the array: (3,)

--- Accessing Data ---
First record: ('Laptop', 1, 1200.5)
All product names: ['Laptop' 'Mouse' 'Keyboard']
All prices: [1200.5   25.     75.99]
Price of the second product: 25.0

--- Performing Operations ---
Total value of all products: 1301.49
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Explanation

1. **dtype Definition**: We define the structure [('name', 'U20'), ('id', 'i4'), ('price', 'f8')].
   - 'U20': A Unicode string with a maximum length of 20.
   - 'i4': A 4-byte (32-bit) integer.
   - 'f8': An 8-byte (64-bit) float.
2.
3. **Array Creation**: We pass our list of tuples and the custom dtype to np.array(). NumPy creates a 1D array where each element is a record conforming to our defined structure.
4. **Accessing Fields**: The key feature is that you can now access the data for an entire "column" using its field name as a string index (e.g., products['price']). This returns a standard, homogeneous NumPy array, on which you can perform fast vectorized operations.

Question 7

How can NumPy be used for audio signal processing?

Theory

NumPy is the cornerstone of audio signal processing in Python. An audio signal, in its digital form, is simply a **time series**—a sequence of numbers representing the amplitude of the sound wave at discrete, equally spaced points in time. This sequence can be represented perfectly as a **1D NumPy array**.

Once the audio is in a NumPy array, the full power of NumPy's numerical and mathematical functions can be applied to analyze and manipulate it.

Key Applications in Audio Signal Processing

1. **Reading and Representing Audio**:
   - Libraries like scipy.io.wavfile or librosa are used to read audio files (like .wav files). They return two things: the **sampling rate** (e.g., 44100 Hz) and the audio

data itself as a **NumPy array**. The array's dtype (e.g., int16 or float32) represents the bit depth of the audio.

2.
3. **Basic Audio Manipulation**:
    - **Changing Volume**: This is simply a multiplication operation. louder_signal = signal_array * 1.5.
    - **Mixing Audio**: Two audio signals can be mixed by adding their NumPy arrays together.
    - **Slicing**: You can easily extract a specific segment of the audio by slicing the array.
4.
5. **Frequency Analysis (Fourier Transform)**:
    - **Concept**: This is the most fundamental analysis task. The **Fast Fourier Transform (FFT)** is used to convert the audio signal from the time domain (amplitude vs. time) to the **frequency domain** (amplitude vs. frequency).
    - **NumPy's Role**: NumPy's np.fft module provides a highly optimized implementation of the FFT algorithm.
    - **Use Case**: This allows you to see which frequencies (musical notes, pitches) are present in the audio signal. It is the basis for equalizers, pitch detection, and many other audio effects.
6.
7. **Filtering**:
    - **Concept**: Applying filters to an audio signal to remove or enhance certain frequencies.
    - **NumPy's Role**: Filtering is often done by performing a **convolution** between the audio signal array and a filter kernel array. NumPy's np.convolve can be used for this. More advanced filtering is done in the frequency domain using the FFT.
8.

Code Example

This example demonstrates reading a WAV file, performing an FFT, and visualizing the frequency spectrum.

Generated python
```
    import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile

# --- 1. Generate a sample audio signal (a sine wave) ---
# In a real case, you would use: sample_rate, signal = wavfile.read('my_audio.wav')
sample_rate = 44100  # Hz
duration = 1.0       # seconds
frequency1 = 440.0   # Hz (A4 note)
frequency2 = 880.0   # Hz (A5 note)
```

```python
# Create the time axis
t = np.linspace(0., duration, int(sample_rate * duration))
# Create a signal with two frequencies
signal = 0.5 * np.sin(2. * np.pi * frequency1 * t) + 0.3 * np.sin(2. * np.pi * frequency2 * t)

# --- 2. Perform Fourier Transform ---
# Use np.fft.fft to get the complex frequency components
# Use np.fft.fftfreq to get the corresponding frequency bins
n_samples = len(signal)
yf = np.fft.fft(signal)
xf = np.fft.fftfreq(n_samples, 1 / sample_rate)

# --- 3. Visualize the results ---
plt.figure(figsize=(12, 6))

# Plot the time-domain signal
plt.subplot(1, 2, 1)
plt.plot(t[:500], signal[:500]) # Plot only the first 500 samples for clarity
plt.title("Time Domain Signal")
plt.xlabel("Time [s]")
plt.ylabel("Amplitude")
plt.grid(True)

# Plot the frequency-domain spectrum
plt.subplot(1, 2, 2)
# We plot the absolute value and only the positive frequencies
plt.plot(xf[:n_samples//2], 2.0/n_samples * np.abs(yf[:n_samples//2]))
plt.title("Frequency Domain (Spectrum)")
plt.xlabel("Frequency [Hz]")
plt.ylabel("Amplitude")
plt.xlim(0, 1200) # Zoom in on the relevant frequencies
plt.grid(True)

plt.tight_layout()
plt.show()
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

The plot on the right clearly shows two sharp peaks at 440 Hz and 880 Hz, correctly identifying the frequencies we put into the signal. This demonstrates how NumPy's fft module can be used to analyze the content of an audio signal.

Question 8

How do you handle NaN or infinite values in a NumPy array?

Theory

NaN (Not a Number) and inf (infinity) are special floating-point values that can appear in NumPy arrays as a result of invalid mathematical operations (like 0/0 for NaN or 1/0 for inf) or from loading data with missing entries.

These values are "contagious" in that they will propagate through computations (e.g., 10 + np.nan results in nan). Therefore, they must be identified and handled before performing most numerical analysis. NumPy provides a specialized set of functions for this.

1. Detecting NaN and inf

- **np.isnan(arr)**: Returns a boolean array of the same shape as arr, with True where elements are NaN and False otherwise.
- **np.isinf(arr)**: Returns a boolean array with True for inf and -inf elements.
- **np.isfinite(arr)**: The inverse of the above. Returns True for elements that are *not* NaN or inf. This is often the most useful check.

Important Note: You cannot use standard equality comparison to check for NaN. np.nan == np.nan will always return False. You must use np.isnan().

2. Counting NaN and inf

- You can combine the detection functions with np.sum() to count the number of invalid values. Since True is treated as 1 and False as 0, the sum of the boolean array gives the count.
    - np.isnan(arr).sum()
-

3. Removing or Replacing NaN and inf

- **Removing**: You can use boolean indexing to filter out NaN values, but this is only straightforward for 1D arrays.
    - arr[~np.isnan(arr)] or arr[np.isfinite(arr)]
-
- **Replacing (Imputation)**: This is the most common approach.

    - **np.nan_to_num(arr)**: A powerful function that replaces NaN with a specified value (default is 0.0), positive infinity with a large positive number, and negative infinity with a large negative number.
    - **Boolean Indexing**: You can use the boolean masks to selectively replace values.
        - arr[np.isnan(arr)] = 0
        - Calculate a mean or median ignoring NaNs, and then fill. mean = np.nanmean(arr), then arr[np.isnan(arr)] = mean.
    -
-

4. NaN-Safe Mathematical Functions

- NumPy provides versions of common aggregate functions that automatically ignore NaN values during computation.
- **Examples**:
    - np.nanmean()
    - np.nanmedian()
    - np.nansum()
    - np.nanstd()
-

Code Example

Generated python

```
    import numpy as np

# Create an array with NaN and inf values
arr = np.array([1.0, 2.0, np.nan, 4.0, np.inf, -np.inf, 6.0])
print(f"Original array: {arr}\n")

# --- 1. Detection ---
print("--- Detection ---")
print(f"Is NaN? {np.isnan(arr)}")
print(f"Is Infinite? {np.isinf(arr)}")
print(f"Is Finite? {np.isfinite(arr)}\n")

# --- 2. Counting ---
print("--- Counting ---")
print(f"Number of NaN values: {np.isnan(arr).sum()}\n")

# --- 3. NaN-safe Computations ---
print("--- NaN-safe Computations ---")
# Standard mean will fail (propagate nan)
print(f"np.mean(arr): {np.mean(arr)}")
# NaN-safe mean will work
```

```
print(f"np.nanmean(arr): {np.nanmean(arr)}\n")

# --- 4. Replacing ---
print("--- Replacing ---")
# Replace NaN with 0 and inf with large numbers
arr_clean1 = np.nan_to_num(arr)
print(f"Using np.nan_to_num(): {arr_clean1}")

# Replace NaN with the mean of the finite values
# Create a copy to avoid modifying the original
arr_clean2 = arr.copy()
mean_val = np.nanmean(arr_clean2[np.isfinite(arr_clean2)]) # Calculate mean of finite values
arr_clean2[np.isnan(arr_clean2)] = mean_val
print(f"Replacing NaN with mean: {arr_clean2}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code
```
   Original array: [  1.   2.  nan   4.  inf  -inf   6.]

--- Detection ---
Is NaN? [False False  True False False False False]
Is Infinite? [False False False False  True  True False]
Is Finite? [ True  True False  True False False  True]

--- Counting ---
Number of NaN values: 1

--- NaN-safe Computations ---
np.mean(arr): nan
np.nanmean(arr): 3.25

--- Replacing ---
Using np.nan_to_num(): [ 1.00000000e+00  2.00000000e+00  0.00000000e+00
4.00000000e+00
  1.79769313e+308 -1.79769313e+308  6.00000000e+00]
Replacing NaN with mean: [  1.    2.    3.25  4.    inf  -inf   6.  ]
```

IGNORE_WHEN_COPYING_START
content_copy download

Use code with caution.
IGNORE_WHEN_COPYING_END

Question 9

What methods are there in NumPy to deal with missing data?

Theory

While the **Pandas** library is generally the preferred tool for handling missing data in a structured, tabular format (with its powerful .isnull(), .fillna(), and .dropna() methods), NumPy provides its own set of tools for dealing with missing data at the array level.

In NumPy, missing data is typically represented by the special floating-point value **np.nan** (Not a Number). The primary methods for handling np.nan fall into three categories: **detection**, **removal**, and **computation**.

1. Detection

The first step is to identify where the missing values are.

- **np.isnan(arr)**: This is the fundamental function for detection. It returns a boolean array of the same shape as the input array, with True at the locations of np.nan values and False everywhere else.

2. Removal

This involves creating a new array that excludes the missing values.

- **Boolean Indexing**: This is the most common way to remove NaNs. You create a boolean mask and use it to select only the valid elements.
    - arr[~np.isnan(arr)]
    - arr[np.isfinite(arr)] (This also removes inf values)
-
- **Note**: This is most straightforward for 1D arrays. For 2D arrays, this will flatten the array. To remove entire rows or columns containing NaNs, you would typically use boolean indexing along an axis:
    - arr[~np.isnan(arr).any(axis=1)] (Removes rows with any NaNs)
-

3. Computation (Calculation while ignoring NaNs)

This is often the most practical approach. Instead of changing the array, you use special versions of aggregate functions that are "NaN-aware."

- **NaN-Safe Functions**: NumPy provides a suite of functions that perform a calculation while simply ignoring any NaN values.
    - np.nanmean(arr): Computes the mean, ignoring NaNs.
    - np.nanmedian(arr): Computes the median, ignoring NaNs.
    - np.nansum(arr): Computes the sum, treating NaNs as zero.
    - np.nanstd(arr): Computes the standard deviation, ignoring NaNs.
    - np.nanmin(arr) / np.nanmax(arr): Find the min/max, ignoring NaNs.
-

4. Replacement (Imputation)

This involves filling the NaN values with another value.

- **Boolean Indexing**: Use a boolean mask to select the NaNs and assign a new value.
    - arr[np.isnan(arr)] = 0 (Replace with a constant)
    - mean = np.nanmean(arr), arr[np.isnan(arr)] = mean (Replace with the mean)
-
- **np.nan_to_num()**: A specialized function that replaces NaNs with 0 (or another specified value) and also handles infinite values.

Code Example

Generated python

```
    import numpy as np

arr = np.array([[1.0, 2.0, np.nan],
                [4.0, 5.0, 6.0],
                [np.nan, 8.0, 9.0]])

print(f"Original Array:\n{arr}\n")

# --- 1. Detection ---
nan_mask = np.isnan(arr)
print(f"NaN Mask:\n{nan_mask}\n")

# --- 2. Removal ---
# Removing rows with any NaN values
arr_no_nan_rows = arr[~np.isnan(arr).any(axis=1)]
print(f"Array with NaN rows removed:\n{arr_no_nan_rows}\n")

# --- 3. NaN-aware Computation ---
mean_val = np.nanmean(arr)
mean_val_axis0 = np.nanmean(arr, axis=0)
print(f"Mean ignoring NaNs (overall): {mean_val:.2f}")
print(f"Mean ignoring NaNs (by column): {np.round(mean_val_axis0, 2)}\n")
```

```
# --- 4. Replacement ---
arr_imputed = arr.copy()
# Impute with the mean of each column
col_means = np.nanmean(arr_imputed, axis=0)
# Find indices where NaN is present
nan_indices = np.where(np.isnan(arr_imputed))
# Replace NaNs with the mean of their respective columns
arr_imputed[nan_indices] = np.take(col_means, nan_indices[1])

print(f"Array after imputing with column means:\n{np.round(arr_imputed, 2)}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Question 10

How do you find unique values and their counts in a NumPy array?

Theory

Finding the unique elements in an array and counting their occurrences are common operations in data analysis, for tasks like understanding the distribution of categorical variables or building a vocabulary from text data. NumPy provides highly optimized functions for these tasks.

Key Functions

1. **np.unique(arr, return_counts=False)**:
    - **Purpose**: This is the primary function for this task.
    - **Default Behavior**: By default (return_counts=False), it returns a **sorted array** containing only the **unique** elements of the input array.
    - **With return_counts=True**: This is the more powerful usage. When set to True, the function returns a **tuple of two arrays**:
        1. The array of unique values.
        2. An array of the same shape containing the **count** of how many times each unique value appeared in the original array.
    -
2.
3. **np.bincount(arr)**:
    - **Purpose**: This is a very fast and specialized function for counting occurrences, but it has a specific requirement.
    - **Requirement**: It only works on 1D arrays of **non-negative integers**.

    ○ **How it works**: It returns an array where the value at index i is the number of times i appeared in the input array. The length of the output array is one greater than the maximum value in the input array.

4.

Code Example

Generated python

```
    import numpy as np

# A sample array with duplicate values
arr = np.array([1, 2, 5, 2, 3, 5, 1, 4, 5, 2])

print(f"Original Array: {arr}\n")

# --- 1. Using np.unique ---
print("--- Using np.unique ---")

# Get only the unique values
unique_values = np.unique(arr)
print(f"Unique values (sorted): {unique_values}")

# Get unique values and their counts
unique_vals, counts = np.unique(arr, return_counts=True)
print(f"Unique values: {unique_vals}")
print(f"Counts: {counts}\n")

# Combine them for a nice display
unique_counts = np.asarray((unique_vals, counts)).T
print("Value | Count")
print("------------")
for val, count in unique_counts:
    print(f"{val:5d} | {count:5d}")


# --- 2. Using np.bincount (for non-negative integers) ---
print("\n--- Using np.bincount ---")
# bincount is often faster for this specific use case
bincount_result = np.bincount(arr)
print(f"Result of np.bincount(arr): {bincount_result}")
print("(Value at index 'i' is the count of 'i')")

# Displaying the results from bincount
print("\nValue | Count (from bincount)")
print("-----------------------------")
```

```
for i, count in enumerate(bincount_result):
    if count > 0:
        print(f"{i:5d} | {count:5d}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Output:

Generated code
```
    Original Array: [1 2 5 2 3 5 1 4 5 2]

--- Using np.unique ---
Unique values (sorted): [1 2 3 4 5]
Unique values: [1 2 3 4 5]
Counts: [2 3 1 1 3]

Value | Count
------------
    1 |     2
    2 |     3
    3 |     1
    4 |     1
    5 |     3

--- Using np.bincount ---
Result of np.bincount(arr): [0 2 3 1 1 3]
(Value at index 'i' is the count of 'i')

Value | Count (from bincount)
------------------------------
    1 |     2
    2 |     3
    3 |     1
    4 |     1
    5 |     3
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.
IGNORE_WHEN_COPYING_END

Choosing Between np.unique and np.bincount

- Use **np.unique** when you are working with floats, negative numbers, or strings, or when you don't need the counts. It is the general-purpose solution.
- Use **np.bincount** when you are working with an array of non-negative integers and you need the counts. It is generally faster than np.unique for this specific task.

Question 11

How can you use NumPy arrays with Cython for performance optimization?

Theory

While NumPy is very fast because its core is written in C, sometimes you need to write a custom algorithm with complex logic that cannot be expressed as a simple sequence of vectorized NumPy operations. Writing this algorithm in a pure Python for loop would be very slow.

This is where **Cython** comes in. Cython is a programming language that makes it easy to write C extensions for Python. It is a superset of Python, so you can start with Python code and then gradually add C-style static type declarations to get massive performance boosts.

Using NumPy arrays with Cython is a powerful combination because you can write a function in Cython that operates **directly on the raw data buffer of a NumPy array**, bypassing the slow Python interpreter for the looping logic.

The Process

1. **Write a Cython file (.pyx)**: You write your function in a file with a .pyx extension.
2. **Add Type Declarations**: Inside the function, you declare the types of your variables (e.g., cdef int i) and, most importantly, you create **typed memoryviews** of the NumPy arrays. A memoryview allows Cython to access the array's data buffer directly at the C level.
3. **Disable Safety Checks (Optional but fast)**: For maximum performance, you can turn off Python's default safety checks (like bounds checking) using compiler directives. This is faster but requires you to be sure your code is correct.
4. **Compile the Cython code**: You use a setup.py file to compile the .pyx file into a C extension module (.so on Linux or .pyd on Windows).
5. **Import and Use**: You can then import the compiled module into your Python script and call the super-fast Cython function.

Code Example

Let's create a Cython function that calculates the sum of squared differences between two NumPy arrays, a common calculation in machine learning.

1. The Cython file (fast_sum.pyx):

Generated cython

```
 # To use this, you would save it as 'fast_sum.pyx'
import numpy as np
# cimport is used to import C-level definitions
cimport numpy as np
cimport cython

# This decorator turns off some safety checks for speed
@cython.boundscheck(False)
@cython.wraparound(False)
def sum_squared_diff_cython(np.ndarray[double, ndim=1] a, np.ndarray[double, ndim=1] b):
    """
    Calculates the sum of squared differences between two 1D double arrays.
    """
    # Declare C-level variables with cdef
    cdef double total = 0.0
    cdef int i
    cdef int n = a.shape[0]

    # This loop runs entirely at the C level, which is very fast
    for i in range(n):
        total += (a[i] - b[i]) ** 2

    return total
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Cython
IGNORE_WHEN_COPYING_END

2. The setup file (setup.py):

Generated python

```
 # Save this as 'setup.py' in the same directory
from setuptools import setup
from Cython.Build import cythonize
import numpy

setup(
    ext_modules = cythonize("fast_sum.pyx"),
    include_dirs=[numpy.get_include()]
)
```

IGNORE_WHEN_COPYING_START
content_copy download

Use code with caution. Python
IGNORE_WHEN_COPYING_END

To compile: Run python setup.py build_ext --inplace in your terminal. This will create fast_sum.c and a compiled module.

3. The Python script to use it:

Generated python
```
    import numpy as np
import time
# This import works after you've compiled the .pyx file
from fast_sum import sum_squared_diff_cython

# Create some large arrays
a = np.random.rand(10_000_000)
b = np.random.rand(10_000_000)

# --- Compare performance ---

# NumPy vectorized version (very fast)
start = time.time()
result_numpy = np.sum((a - b) ** 2)
numpy_time = time.time() - start
print(f"NumPy vectorized time: {numpy_time:.6f} seconds")

# Cython version
start = time.time()
result_cython = sum_squared_diff_cython(a, b)
cython_time = time.time() - start
print(f"Cython function time:  {cython_time:.6f} seconds")

print(f"\nResults are close: {np.isclose(result_numpy, result_cython)}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

Result

When you run this, you will find that the Cython function's performance is very close to (and sometimes even slightly faster than) the pure NumPy vectorized version. This demonstrates that Cython allows you to write custom, complex algorithms with the **same performance as native NumPy operations**, something that is impossible to achieve with a pure Python for loop.