Array Slicing

Definition	Array slicing allows you to extract specific parts of an array. It works similarly to list slicing in Python.
Example	`arr = np.arr ay([0, 1, 2, 3, 4, 5])`
Slicing syntax	arr[start:stop:step]

Basic slicing	`slice_1 = arr[1:4] # [1, 2, 3]`
	`slice_2 = arr[:3] # [0, 1, 2]`
	`slice_3 = arr[3:] # [3, 4, 5]`

Negative indexing	`slice_4 = arr[-3:] # [3, 4, 5]` `slice_5 = arr[:-2] # [0, 1, 2]`
Step slicing	`slice_6 = arr[::2] # [0, 2, 4]` `slice_7 = arr[1::2] # [1, 3, 5]`
Reverse array	`slice_8 = arr[::-1] # [5, 4, 3, 2, 1, 0]`
Slicing 2D arrays	`arr_2d = np.arr ay([[1, 2, 3], [4, 5, 6], [7, 8, 9]])` `slice_9 = arr_2d[:2, 1:] # [[2, 3], [5, 6]]`

Performance Tips and Tricks

Vectorization	Utilize NumPy's built-in vectorized operations whenever possible. These operations are optimized and significantly faster than equivalent scalar operations.
Avoiding Loops	Minimize the use of Python loops when working with NumPy arrays. Instead, try to express operations as array operations. Loops in Python can be slow compared to vectorized operations.
Use Broadcasting	Take advantage of NumPy's broadcasting rules to perform operations on arrays of different shapes efficiently. Broadcasting allows NumPy to work with arrays of different sizes without making unnecessary copies of data.

Performance Tips and Tricks (cont)

Avoid Copies	Be mindful of unnecessary array copies, especially when working with large datasets. NumPy arrays share memory when possible, but certain operations may create copies, which can impact performance and memory usage.
Use In-Place Operations	Whenever feasible, use in-place operations (+=, *=, etc.) to modify arrays without creating new ones. This reduces memory overhead and can improve performance.
Memory Layout	Understand how memory layout affects performance, especially for large arrays. NumPy arrays can be stored in different memory orders (C-order vs. Fortran-order). Choosing the appropriate memory layout can sometimes lead to better performance, especially when performing operations along specific axes.
Data Types	Choose appropriate data types for your arrays to minimize memory usage and improve performance. Using smaller data types (e.g., np.float32 instead of np.float64) can reduce memory overhead and may lead to faster computations, especially on platforms with limited memory bandwidth.
NumExpr and Numba	Consider using specialized libraries like NumExpr or Numba for performance-critical sections of your code. These libraries can often provide significant speedups by compiling expressions or functions to native machine code.

Performance Tips and Tricks (cont)

Parall-elism	NumPy itself doesn't provide built-in parallelism, but you can leverage multi-threading or multi-processing libraries like concurrent.futures or joblib to parallelize certain operations, especially when working with large datasets or computationally intensive tasks.
Profiling	Use profiling tools like cProfile or specialized profilers such as line_profiler or memory_profiler to identify perfor-mance bottlenecks in your code. Optimizing code based on actual profiling results can lead to more significant performance improvements.

Array Concatenation and Splitting

Concat-enation	`array1 = np.arr ay([[1, 2, 3], [4, 5, 6]])`
	`array2 = np.arr ay([[7, 8, 9]])`
	`concatenated_array = np.con cat ena te( (ar ray1, array2), axis =0)`
	`# vertically`
	`print(concatenated_array)`

numpy.c oncat-enate()	Concatenates arrays along a specified axis.
numpy.v stack() and numpy.h stack()	Stack arrays vertically and horizontally, respectively.
numpy.d stack()	Stack arrays depth-wise.

Splitting	`split_ arrays = np.spl it( con cat ena ted _array, [2],`
	`axis=0)`
	`# split after the second row`
	`print(split_arrays)`

numpy.s plit()	Split an array into multiple sub-arrays along a specified axis.
numpy.h split() and numpy.v split()	Split arrays horizontally and vertically, respectively.
numpy.d split()	Split arrays depth-wise.

Basic Operations

Addition	array1 + array2
Subtraction	array1 - array2
Multiplication	array1 * array2
Division	array1 / array2
Floor Division	array1 // array2
Modulus	array1 % array2
Exponentiation	array1 ** array2
Absolute	np.abs(array)
Negative	-array
Reciprocal	1 / array
Sum	np.sum(array)
Minimum	np.min(array)
Maximum	np.max(array)
Mean	np.mean(array)
Median	np.median(array)
Standard Deviation	np.std(array)
Variance	np.var(array)
Dot Product	np.dot(array1, array2)
Cross Product	np.cross(array1, array2)

NaN Handling

Identi-fying NaNs	Use np.isnan() function to check for NaN values in an array.
Removing NaNs	Use np.isnan() to create a boolean mask, then use boolean indexing to select non-NaN values.
Replacing NaNs	Use np.nan_to_num() to replace NaNs with a specified value. Use np.nanmean(), np.nanmedian(), etc., to compute mean, median, etc., ignoring NaNs.

NaN Handling (cont)

Interpolating NaNs	Sure, here's a short content for "NaN Handling" on your NumPy cheat sheet: NaN Handling: Identifying NaNs: Use np.isnan() function to check for NaN values in an array. Removing NaNs: Use np.isnan() to create a boolean mask, then use boolean indexing to select non-NaN values. Replacing NaNs: Use np.nan_to_num() to replace NaNs with a specified value. Use np.nanmean(), np.nanmedian(), etc., to compute mean, median, etc., ignoring NaNs. Interpolating NaNs
Ignoring NaNs in Operations	Many NumPy functions have NaN-aware counterparts, like np.nanmean(), np.nansum(), etc., that ignore NaNs in computations.
Handling NaNs in Aggregations	Aggregation functions (np.sum(), np.mean(), etc.) typically return NaN if any NaNs are present in the input array. Use skipna=True parameter in pandas DataFrame functions for NaN handling.
Dealing with NaNs in Linear Algebra	NumPy's linear algebra functions (np.linalg.inv(), np.linalg.solve(), etc.) handle NaNs by raising LinAlgError.

Broadcasting

Broadcasting is a powerful feature in NumPy that allows arrays of different shapes to be combined in arithmetic operations.

When operating on arrays of different shapes, NumPy automatically broadcasts the smaller array across the larger array so that they have compatible shapes.

This eliminates the need for explicit looping over array elements, making code more concise and efficient.

Broadcasting is particularly useful for performing operations between arrays of different dimensions or sizes without needing to reshape them explicitly.

Mathematical Functions

Definition	NumPy provides a wide range of mathematical functions that operate element-wise on arrays, allowing for efficient computation across large datasets.
Trigonometric Functions	np.sin(), np.cos(), np.tan(), np.arcsin(), np.arccos(), np.arctan()
Hyperbolic Functions	np.sinh(), np.cosh(), np.tanh(), np.arcsinh(), np.arccosh(), np.arctanh()
Exponential and Logarithmic Functions	np.exp(), np.log(), np.log2(), np.log10()
Rounding	np.round(), np.floor(), np.ceil(), np.trunc()
Absolute Value	np.abs()
Factorial and Combinations	np.factorial(), np.comb()
Gamma and Beta Functions	np.gamma(), np.beta()
Sum, Mean, Median	np.sum(), np.mean(), np.median()
Standard Deviation, Variance	np.std(), np.var()
Matrix Operations	np.dot(), np.inner(), np.outer(), np.cross()
Eigenvalues and Eigenvectors	np.linalg.eig(), np.linalg.eigh(), np.linalg.eigvals()
Matrix Decompositions	np.linalg.svd(), np.linalg.qr(), np.linalg.cholesky()

Array Creation

numpy.array()	Create an array from a Python list or tuple.
Example	`arr = np.arr ay([1, 2, 3])`
numpy.zeros()	Create an array filled with zeros.
Example	`zeros_arr = np.zer os((3, 3))`
numpy.ones()	Create an array filled with ones.
Example	`ones_arr = np.one s((2, 2))`
numpy.arange()	Create an array with a range of values.
Example	`range_arr = np.ara nge(0, 10, 2)  # array([C )`
numpy.l-in-space()	Create an array with evenly spaced values.

Array Creation (cont)

Example	`linspa ce_arr = np.lin spa ce(0, 10, 5)   # array([ 0., 2.5, 5., 7.5, 10. ])`
numpy.eye()	Create an identity matrix.
Example	`identi ty_mat = np.eye(3)`
numpy.random.rand()	Create an array with random values from a uniform distribution.
Example	`random_arr = np.ran dom.ra nd(2, 2)`
numpy.random.randn()	Create an array with random values from a standard normal distribution.
Example	`random _no rma l_arr = np.ran dom.ra ndn(2, 2)`
numpy.random.randint()	Create an array with random integers.
Example	`random _in t_arr = np.ran dom.ra ndi nt(0, 10, size=(2, 2))`
numpy.full()	Create an array filled with a specified value.
Example	`full_arr = np.ful l((2, 2), 7)`
numpy.empty()	Create an uninitialized array (values are not set, might be arbitrary).
Example	`empty_arr = np.emp ty((2, 2))`

Linear Algebra

Matrix Multiplication	np.dot() or @ operator for matrix multiplication.
Transpose	np.transpose() or .T attribute for transposing a matrix.
Inverse	np.linalg.inv() for calculating the inverse of a matrix.
Determinant	np.linalg.det() for computing the determinant of a matrix.
Eigenvalues and Eigenvectors	np.linalg.eig() for computing eigenvalues and eigenvectors.
Matrix Decompositions	Functions like np.linalg.qr(), np.linalg.svd(), and np.linalg.cholesky() for various matrix decompositions.
Solving Linear Systems	np.linalg.solve() for solving systems of linear equations.
Vectorization	Leveraging NumPy's broadcasting and array operations for efficient linear algebra computations.

Statistical Functions

mean	Computes the arithmetic mean along a specified axis.
median	Computes the median along a specified axis.
average	Computes the weighted average along a specified axis.
std	Computes the standard deviation along a specified axis.
var	Computes the variance along a specified axis.
amin	Finds the minimum value along a specified axis.
amax	Finds the maximum value along a specified axis.
argmin	Returns the indices of the minimum value along a specified axis.
argmax	Returns the indices of the maximum value along a specified axis.
percentile	Computes the q-th percentile of the data along a specified axis.
histogram	Computes the histogram of a set of data.

Comparison with Python Lists

Performance	NumPy arrays are faster and more memory efficient compared to Python lists, especially for large datasets. This is because NumPy arrays are stored in contiguous blocks of memory and have optimized functions for mathematical operations, whereas Python lists are more flexible but slower due to their dynamic nature.
Vectorized Operations	NumPy allows for vectorized operations, which means you can perform operations on entire arrays without the need for explicit looping. This leads to concise and efficient code compared to using loops with Python lists.
Multidimensional Arrays	NumPy supports multidimensional arrays, whereas Python lists are limited to one-dimensional arrays or nested lists, which can be less intuitive for handling multi-dimensional data.

Comparison with Python Lists (cont)

Broadcasting	NumPy arrays support broadcasting, which enables operations between arrays of different shapes and sizes. In contrast, performing similar operations with Python lists would require explicit looping or list comprehensions.
Type Stability	NumPy arrays have a fixed data type, which leads to better performance and memory efficiency. Python lists can contain elements of different types, leading to potential type conversion overhead.
Rich Set of Functions	NumPy provides a wide range of mathematical and statistical functions optimized for arrays, whereas Python lists require manual implementation or the use of external libraries for similar functionality.
Memory Usage	NumPy arrays typically consume less memory compared to Python lists, especially for large datasets, due to their fixed data type and efficient storage format.
Indexing and Slicing	NumPy arrays offer more powerful and convenient indexing and slicing capabilities compared to Python lists, making it easier to manipulate and access specific elements or subarrays.
Parallel Processing	NumPy operations can leverage parallel processing capabilities of modern CPUs through libraries like Intel MKL or OpenBLAS, resulting in significant performance gains for certain operations compared to Python lists.
Interoperability	NumPy arrays can be easily integrated with other scientific computing libraries in Python ecosystem, such as SciPy, Pandas, and Matplotlib, allowing seamless data exchange and interoperability.

Masked Arrays

Why?	Masked arrays in NumPy allow you to handle missing or invalid data efficiently.
What are Masked Arrays?	Masked arrays are arrays with a companion boolean mask array, where elements that are marked as "masked" are ignored during computations.
Creating Masked Arrays	You can create masked arrays using the numpy.ma.masked_array function, specifying the data array and the mask array.
Masking	Masking is the process of marking certain elements of an array as invalid or missing. You can manually create masks or use functions like numpy.ma.masked_where to create masks based on conditions.
Operations with Masked Arrays	Operations involving masked arrays automatically handle masked values by ignoring them in computations. This allows for easy handling of missing data without explicitly removing or replacing them.
Masked Array Methods	NumPy provides methods for masked arrays to perform various operations like calculating statistics, manipulating data, and more. These methods are similar to regular array methods but handle masked values appropriately.
Applications	Masked arrays are useful in scenarios where datasets contain missing or invalid data points. They are commonly used in scientific computing, data analysis, and handling time series data where missing values are prevalent.

Random Number Generation

np.random.rand	Generates random numbers from a uniform distribution over [0, 1).
np.random.randn	Generates random numbers from a standard normal distribution (mean 0, standard deviation 1).
np.random.randint	Generates random integers from a specified low (inclusive) to high (exclusive) range.
np.random.random_sample or np.random.random	Generates random floats in the half-open interval [0.0, 1.0).
np.random.choice	Generates random samples from a given 1-D array or list.
np.random.shuffle	Shuffles the elements of an array in place.
np.random.permutation	Randomly permutes a sequence or returns a permuted range.
np.random.seed	Sets the random seed to ensure reproducibility of results.

Filtering Arrays

Filtering Arrays	NumPy provides powerful tools for filtering arrays based on certain conditions. Filtering allows you to select elements from an array that meet specific criteria.
Syntax	`filtered_array = array[condition]`

Example	`import numpy as np`
	`arr = np.array([1, 2, 3, 4, 5])`
	`filtered = arr[arr > 2]`
	`# Select elements greater than 2`
	`print(filtered)`
	`# Output: [3 4 5]`

Combining Conditions	Conditions can be combined using logical operators like & (and) and | (or).

Filtering Arrays (cont)

Example	`arr = np.array([1, 2, 3, 4, 5])`
	`filtered = arr[(arr > 2) & (arr < 5)]`
	`# Select elements between 2 and 5`
	`print(filtered)`
	`# Output: [3 4]`

Using Functions	NumPy also provides functions like np.where() and np.extract() for more complex filtering.

Example	`arr = np.array([1, 2, 3, 4, 5])`
	`filtered = np.where(arr % 2 == 0, arr, 0)`
	
	`# Replace odd elements with 0`
	`print(filtered)`
	`# Output: [0 2 0 4 0]`

Array Iteration

For Loops	Iterate over arrays using traditional for loops. This is useful for basic iteration but might not be the most efficient method for large arrays.
nditer	The nditer object allows iterating over arrays in a more efficient and flexible way. It provides options to specify the order of iteration, data type casting, and external loop handling.
Flat Iteration	The flat attribute of NumPy arrays returns an iterator that iterates over all elements of the array as if it were a flattened 1D array. This is useful for simple element-wise operations.
Broadcasting	When performing operations between arrays of different shapes, NumPy automatically broadcasts the arrays to compatible shapes. Understanding broadcasting rules can help efficiently iterate over arrays without explicit loops.

Array Iteration (cont)

Vectorized Operations	Instead of explicit iteration, utilize NumPy's built-in vectorized operations which operate on entire arrays rather than individual elements. This often leads to faster and more concise code.

Array Reshaping

Array Reshaping	Reshaping arrays in NumPy allows you to change the shape or dimensions of an existing array without changing its data. This is useful for tasks like converting a 1D array into a 2D array or vice versa, or for preparing data for certain operations like matrix multiplication.
reshape()	The reshape() function in NumPy allows you to change the shape of an array to a specified shape.
For example:	import numpy as np arr = np.array([1, 2, 3, 4, 5, 6]) reshaped_arr = arr.reshape((2, 3))
Explanation	This will reshape the array arr into a 2x3 matrix.
resize()	Similar to reshape(), resize() changes the shape of an array, but it also modifies the original array if necessary to accommodate the new shape.
Example	`arr = np.arr ay([[1, 2], [3, 4]])` `resized_arr = np.res ize (arr, (3, 2))`
Explanation	If the new shape requires more elements than the original array has, resize() repeats the original array to fill in the new shape.
flatten()	The flatten() method collapses a multi-dimensional array into a 1D array by iterating over all elements in row-major (C-style) order.
Example	`arr = np.arr ay([[1, 2], [3, 4]])` `flattened_arr = arr.fl atten()`
Explanation	This will flatten the 2D array into a 1D array.

Array Reshaping (cont)

ravel()	Similar to flatten(), ravel() also flattens multi-dimensional arrays into a 1D array, but it returns a view of the original array whenever possible.
Example	`arr = np.arr ay([[1, 2], [3, 4]])` `raveled_arr = arr.ra vel()`
Explanation	This method can be more efficient in terms of memory usage than flatten().
transpose()	The transpose() method rearranges the dimensions of an array. For 2D arrays, it effectively swaps rows and columns.
Example	`arr = np.arr ay([[1, 2], [3, 4]])` `transposed_arr = arr.tr ans pose()`
Explanation	This will transpose the 2x2 matrix, swapping rows and columns.

Sorting Arrays

np.sort(arr)	Returns a sorted copy of the array.
arr.sort()	Sorts the array in-place.
np.argsort(arr)	Returns the indices that would sort the array.
np.lexsort()	Performs an indirect sort using a sequence of keys.
np.sort_complex(arr)	Sorts the array of complex numbers based on the real part first, then the imaginary part.
np.partition(arr, k)	Rearranges the elements in such a way that the kth element will be in its correct position in the sorted array, with all smaller elements to its left and all larger elements to its right.
np.argpartition(arr, k)	Returns the indices that would partition the array.

Array Indexing

Single Element Access	Use square brackets [] to access individual elements of an array by specifying the indices for each dimension. For example, arr[0, 1] accesses the element at the first row and second column of the array arr.

Array Indexing (cont)

Negative Indexing	Negative indices can be used to access elements from the end of the array. For instance, arr[-1] accesses the last element of the array arr.
Slice Indexing	NumPy arrays support slicing similar to Python lists. You can use the colon : operator to specify a range of indices. For example, arr[1:3] retrieves elements from index 1 to index 2 (inclusive) along the first axis.
Integer Array Indexing	You can use arrays of integer indices to extract specific elements from an array. For example, arr[[0, 2, 4]] retrieves elements at indices 0, 2, and 4 along the first axis.
Boolean Array Indexing (Boolean Masking)	You can use boolean arrays to filter elements from an array based on a condition. For example, arr[arr > 0] retrieves all elements of arr that are greater than zero.
Fancy Indexing	Fancy indexing allows you to select multiple elements from an array using arrays of indices or boolean masks. This method can be used to perform advanced selection operations efficiently.