NumPy standard data types :NumPy provides a wide range of standard data types to represent different kinds of numerical data.

1. **int8, int16, int32, int64**: Signed integer types with 8, 16, 32, and 64 bits respectively.
2. **uint8, uint16, uint32, uint64**: Unsigned integer types with 8, 16, 32, and 64 bits respectively.
3. **float16, float32, float64**: Floating-point types with 16, 32, and 64 bits respectively.
4. **complex64, complex128**: Complex number types with 64 and 128 bits respectively.
5. **bool**: Boolean type representing True or False values.
6. **object**: Object type, allowing any Python object to be stored in an array.
7. **string_**: Variable-length string type.
8. **unicode_**: Variable-length unicode type.

These data types can be used when creating NumPy arrays by specifying the `dtype` parameter. For example:

```
import numpy as np
arr_int = np.array([1, 2, 3], dtype=np.int32)
arr_float = np.array([1.0, 2.0, 3.0], dtype=np.float64)
arr_complex = np.array([1+2j, 3+4j], dtype=np.complex128)
arr_bool = np.array([True, False, True], dtype=np.bool)
print(arr_int.dtype)      # int32
```

```
print(arr_float.dtype)    # float64
print(arr_complex.dtype)  # complex128
print(arr_bool.dtype)     # bool
```

output

```
int32
float64
complex128
bool
```

In addition to these standard data types, NumPy also allows creating custom data types using the `dtype` class. This allows you to define structured data types or specify the byte order of the elements, among other things. Custom data types provide more flexibility in representing and manipulating complex data structures.

NumPy Array Attributes:NumPy, short for Numerical Python, is a popular library in Python for numerical computations. It provides powerful array data structures and functions for manipulating and analyzing numerical data. NumPy arrays are homogeneous, multidimensional containers of items with the same type. Here are some of the common attributes of NumPy arrays:

1. **shape**: The shape attribute returns a tuple that represents the size of each dimension of the array. For a 1-dimensional array, it

gives the length of the array. For a 2-dimensional array, it gives the number of rows and columns, and so on. For example:

```
import numpy as np

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

2. **dtype**: The dtype attribute indicates the data type of the elements in the array. NumPy provides various data types such as int, float, bool, etc., and allows you to define custom data types. For example:

```
import numpy as np
arr = np.array([1, 2, 3], dtype=np.float64)
print(arr.dtype)  # float64
```

3. **size**: The size attribute returns the total number of elements in the array. It is equal to the product of the dimensions specified in the shape attribute. For example:

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr.size)  # 6
```

4. **ndim**: The ndim attribute returns the number of dimensions or axes of the array. For example:

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

```
print(arr.ndim)  # 2
```

5. **itemsize**: The itemsize attribute returns the size in bytes of each element in the array. For example, an array of type np.float64 will have an itemsize of 8 (8 bytes). For example:

```
import numpy as np

arr = np.array([1, 2, 3], dtype=np.float64)
print(arr.itemsize)  # 8
```

6. **data**: The data attribute returns a buffer object pointing to the start of the data of the array. It is usually not used directly, but it can be useful in certain situations where you need to access the underlying memory of the array.

Array Indexing: Accessing Single Elements

In NumPy, we can access individual elements of an array using indexing. Array indexing is similar to indexing in Python lists, with the addition of supporting multi-dimensional arrays.

1. Accessing 1-D arrays:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
# Accessing a single element at index 2
print(arr[2])  # Output: 3
# Accessing multiple elements using slicing
print(arr[1:4])  # Output: [2 3 4]
```

2. Accessing 2-D arrays:

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
# Accessing a single element at row 0, column 1
print(arr[0, 1])  # Output: 2
# Accessing an entire row
print(arr[1])  # Output: [4 5 6]
# Accessing a column
print(arr[:, 2])  # Output: [3 6]
```

3. Accessing higher-dimensional arrays:

```
import numpy as np
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
# Accessing a single element
print(arr[1, 0, 1])  # Output: 6
# Accessing a subarray
print(arr[:, 0, :])
# Output:
# [[1 2]
#  [5 6]]
```

4. Boolean indexing:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
# Boolean indexing to access elements greater than 3
mask = arr > 3
```

```
print(arr[mask])  # Output: [4 5]
```

ARRAY SLICING

Array slicing in NumPy allows you to access subarrays by specifying ranges or indices. Slicing can be used on both 1-D and multi-dimensional arrays.

1. Slicing 1-D arrays:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
# Accessing a subarray from index 1 to index 3 (exclusive)
print(arr[1:3])  # Output: [2 3]
# Accessing a subarray from the beginning to index 4 (exclusive)
print(arr[:4])  # Output: [1 2 3 4]
# Accessing a subarray from index 2 to the end
print(arr[2:])  # Output: [3 4 5]
# Accessing a subarray with a step size of 2
print(arr[::2])  # Output: [1 3 5]
```

2. Slicing 2-D arrays:

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# Accessing a subarray with rows 1 and 2, and columns 0 to 2 (exclusive)
print(arr[1:3, 0:2])
# Output:
# [[4 5]
```

```
#  [7 8]]
# Accessing a subarray with all rows and columns 1 and 2
print(arr[:, 1:3])
# Output:
# [[2 3]
#  [5 6]
#  [8 9]]
```

3. Slicing with steps:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
# Accessing a subarray with a step size of 2
print(arr[::2])  # Output: [1 3 5 7 9]
# Accessing a subarray in reverse order
print(arr[::-1])  # Output: [10  9  8  7  6  5  4  3  2  1]
```

4. Slicing with boolean indexing:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
# Boolean indexing to access elements greater than 2
mask = arr > 2
print(arr[mask])  # Output: [3 4 5]
```

Slicing in NumPy provides a flexible way to access subarrays and can be used to extract specific portions of an array based on indices or ranges. It is a powerful feature that allows for efficient data manipulation and analysis.

Reshaping arrays

Reshaping arrays in NumPy refers to changing the shape or dimensions of an existing array without modifying its data. Reshaping allows you to transform an array from one shape to another, as long as the total number of elements remains the same. Here's an overview of reshaping arrays in NumPy:

1. Reshaping 1-D arrays:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6])
# Reshape to a 2x3 array
reshaped_arr = arr.reshape((2, 3))
print(reshaped_arr)
# Output:
# [[1 2 3]
#  [4 5 6]]
```

2. Reshaping multi-dimensional arrays:

```
import numpy as np
arr = np.array([[1, 2], [3, 4], [5, 6]])
# Reshape to a 1-D array
reshaped_arr = arr.reshape((6,))
print(reshaped_arr)  # Output: [1 2 3 4 5 6]
# Reshape to a 3x2 array
reshaped_arr = arr.reshape((3, 2))
print(reshaped_arr)
# Output:
```

```
# [[1 2]
#  [3 4]
#  [5 6]]
```

3. Reshaping with -1 parameter:

The -1 parameter can be used in the reshape function to automatically infer the size of one dimension based on the given shape and the size of the other dimensions.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6])
# Reshape to a 2x3 array (inferred from the size of the array)
reshaped_arr = arr.reshape((2, -1))
print(reshaped_arr)
# Output:
# [[1 2 3]
#  [4 5 6]]
# Reshape to a 3x2 array (inferred from the size of the array)
reshaped_arr = arr.reshape((-1, 2))
print(reshaped_arr)
# Output:
# [[1 2]
#  [3 4]
#  [5 6]]
```

4. Flattening arrays:

Flattening an array means converting a multi-dimensional array into a 1-D array.

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
# Flattening the array
flattened_arr = arr.flatten()
print(flattened_arr)  # Output: [1 2 3 4 5 6]
```

Reshaping arrays can be useful when we need to change the dimensions of an array to match a specific requirement or perform operations that require a different shape. NumPy provides the `reshape` function to modify the shape of arrays efficiently.

Array Concatenation:

1. `np.concatenate`: The `np.concatenate` function is used to concatenate arrays along a specified axis. It takes a sequence of arrays as input and concatenates them into a single array.

- Syntax: `np.concatenate((array1, array2, ...), axis=0)`
- `array1`, `array2`, ...: The arrays to be concatenated.
- `axis`: The axis along which the arrays will be concatenated. By default, it is 0 (concatenation along the first axis).
- Example:

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
result = np.concatenate((arr1, arr2))
print(result)
```

Output:

[1 2 3 4 5 6]

2. `np.vstack` and `np.hstack`: These functions are used for vertical and horizontal stacking of arrays, respectively.

- Syntax: `np.vstack((array1, array2, ...))` or `np.hstack((array1, array2, ...))`

- Example:

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
result_vstack = np.vstack((arr1, arr2))
result_hstack = np.hstack((arr1, arr2))
print(result_vstack)
print(result_hstack)
```

Output:

```
[[1 2 3]
 [4 5 6]]
[1 2 3 4 5 6]
```

Array Splitting:

1. `np.split`: The `np.split` function splits an array into multiple subarrays of equal size. It takes an array, the indices or sections at which to split, and the axis along which to split.

- Syntax: `np.split(array, indices_or_sections, axis=0)`

- `array`: The array to be split.

- `indices_or_sections`: An integer specifying the number of equally-sized subarrays or a sequence of indices indicating the split positions.

- `axis`: The axis along which the array will be split. By default, it is 0 (splitting along the first axis).

- Example:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6])
result = np.split(arr, 3)
print(result)
```

Output:

```
[array([1, 2]), array([3, 4]), array([5, 6])]
```

2. `np.vsplit` and `np.hsplit`: These functions split an array vertically and horizontally, respectively.

- Syntax: `np.vsplit(array, indices_or_sections)` or `np.hsplit(array, indices_or_sections)`

- `array`: The array to be split.

- `indices_or_sections`: An integer specifying the number of equally-sized subarrays or a sequence of indices indicating the split positions.

- Example:

```
import numpy as np
arr = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
result_vsplit = np.vsplit(arr, 3)
result_hsplit = np.hsplit(arr, 3)
print(result_vsplit)
```

```
print(result_hsplit)
```

Output:

```
[array([[1, 2, 3]]), array([[4, 5, 6]]), array([[7, 8, 9]])]
[array([[1],
        [4],
        [7]]),
 array([[2],
        [5],
        [8]]),
 array([[3],
        [6],
        [9]])]
```

These numpy functions provide flexible ways to concatenate arrays along different axes and split arrays into multiple smaller arrays.

Array aggregations

Array aggregations in numpy involve performing operations to summarize or compute statistical metrics on arrays. These operations help in analyzing data and extracting useful insights. 1. `np.sum`: This function calculates the sum of array elements along the specified axis or over the entire array.

- Syntax: `np.sum(array, axis=None)`
- `array`: The input array.
- `axis`: The axis along which the sum is calculated. By default, it sums over all elements.
- Example:

```
import numpy as np
arr = np.array([[1, 2, 3],
                [4, 5, 6]])
total_sum = np.sum(arr)
row_sum = np.sum(arr, axis=1)
col_sum = np.sum(arr, axis=0)
print(total_sum)
print(row_sum)
print(col_sum)
```

Output:

21

[6 15]

[5 7 9]

2. `np.mean`: This function calculates the arithmetic mean of array elements along the specified axis or over the entire array.

- Syntax: `np.mean(array, axis=None)`
- `array`: The input array.
- `axis`: The axis along which the mean is calculated. By default, it calculates the mean over all elements.
- Example:

```
import numpy as np
arr = np.array([[1, 2, 3],
                [4, 5, 6]])
overall_mean = np.mean(arr)
row_mean = np.mean(arr, axis=1)
```

```
    col_mean = np.mean(arr, axis=0)
    print(overall_mean)
    print(row_mean)
    print(col_mean)
```

Output:

```
3.5
[2. 5.]
[2.5 3.5 4.5]
```

3. `np.min` and `np.max`: These functions compute the minimum and maximum values of array elements along the specified axis or over the entire array.

- Syntax: `np.min(array, axis=None)` and `np.max(array, axis=None)`
- `array`: The input array.
- `axis`: The axis along which the minimum and maximum are calculated. By default, they are calculated over all elements.
- Example:

```
import numpy as np
arr = np.array([[1, 2, 3],
                [4, 5, 6]])
min_value = np.min(arr)
max_value = np.max(arr)
row_min = np.min(arr, axis=1)
col_max = np.max(arr, axis=0)
print(min_value)
```

```
print(max_value)
print(row_min)
print(col_max)
```

Output:

```
1
6
[1 4]
[4 5 6]
```

Computations on arrays in numpy involve performing mathematical operations on arrays and applying functions element-wise. These operations allow for efficient computation and manipulation of array data.

1. Element-wise Operations:

- Arithmetic Operations: Numpy supports basic arithmetic operations such as addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and exponentiation (`**`) on arrays. These operations are performed element-wise.

- Example:

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
# Element-wise addition
result_add = arr1 + arr2
print(result_add)
```

```
# Element-wise multiplication
result_mul = arr1 * arr2
print(result_mul)
```

Output:

```
[5 7 9]
[4 10 18]
```

2. Universal Functions (ufuncs):

- Numpy provides a wide range of universal functions (ufuncs) that operate element-wise on arrays. These functions can perform mathematical operations, trigonometric calculations, logarithmic operations, etc.

- Example:

```
import numpy as np
arr = np.array([1, 2, 3])
# Square root
result_sqrt = np.sqrt(arr)
print(result_sqrt)
# Exponential function
result_exp = np.exp(arr)
print(result_exp)
```

Output:

```
[1.         1.41421356 1.73205081]
[ 2.71828183  7.3890561  20.08553692]
```

3. Aggregation Functions:

- Numpy's aggregation functions (e.g., `np.sum`, `np.mean`, `np.min`, `np.max`) can be applied element-wise to arrays, computing summary statistics over the entire array or along specific axes.

- Example:

```
import numpy as np
arr = np.array([[1, 2, 3],
                [4, 5, 6]])
# Sum of all elements
total_sum = np.sum(arr)
print(total_sum)
# Mean along axis 0
mean_axis0 = np.mean(arr, axis=0)
print(mean_axis0)
```

Output:

21

[2.5 3.5 4.5]

NumPy's structured arrays

NumPy's structured arrays, also known as structured data types, allow for storing and manipulating heterogeneous data in a tabular format similar to a database table or spreadsheet. A structured array can contain elements of different data types organized into named fields. This provides a way to work with structured data within the NumPy framework.

1. Defining a Structured Data Type:

- Structured data types are defined using the `np.dtype` function, specifying the names and data types of each field.

- Example:

```
import numpy as np
# Define a structured data type with two fields: 'name' and 'age'
dt = np.dtype([('name', np.str_, 16), ('age', np.int32)])
print(dt)
```

Output:

```
[('name', '<U16'), ('age', '<i4')]
```

2. Creating a Structured Array:

- A structured array is created by providing data and specifying the structured data type.

- Example:

```
import numpy as np
# Define a structured data type with two fields: 'name' and 'age'
dt = np.dtype([('name', np.str_, 16), ('age', np.int32)])
# Create a structured array using the defined data type
arr = np.array([('Alice', 25), ('Bob', 30)], dtype=dt)
print(arr)
```

Output:

```
[('Alice', 25) ('Bob', 30)]
```

3. Accessing Structured Array Fields:

- Fields of a structured array can be accessed using field names or indices.

- Example:

```
import numpy as np
# Define a structured data type with two fields: 'name' and 'age'
dt = np.dtype([('name', np.str_, 16), ('age', np.int32)])
# Create a structured array using the defined data type
arr = np.array([('Alice', 25), ('Bob', 30)], dtype=dt)
# Access the 'name' field
print(arr['name'])
# Access the 'age' field
print(arr['age'])
```

Output:

['Alice' 'Bob']

[25 30]

4. Performing Operations on Structured Arrays:

- Structured arrays support various operations, including filtering, slicing, sorting, and aggregation.

- Example:

```
import numpy as np
# Define a structured data type with two fields: 'name' and 'age'
dt = np.dtype([('name', np.str_, 16), ('age', np.int32)])
# Create a structured array using the defined data type
arr = np.array([('Alice', 25), ('Bob', 30), ('Charlie', 35)], dtype=dt)
# Filter the structured array based on age
filtered_arr = arr[arr['age'] > 30]
print(filtered_arr)
```

```
# Sort the structured array by name
sorted_arr = np.sort(arr, order='name')
print(sorted_arr)
```

Output:

```
[('Charlie', 35)]
[('Alice', 25) ('Bob', 30) ('Charlie', 35)]
```