numpy

March 17, 2024

[4]:
```
'''An array that has n-D arrays as its elements is called a (n+1)-D array.
We can use .ndim attribute to check the dimension of an array.'''
from numpy import *
```

[11]:
```
'''Types of Array:
0D Array (Scalar):
   - Represents a single value.
   - Zero dimensions.
   - Created using `numpy.array()` with no sequence-like input.
1D Array (Vector):
   - Represents a sequence of values along a single dimension.
   - One dimension.
   - Created using `numpy.array()` with a list or array-like input.
2D Array (Matrix):
   - Represents a table of values with rows and columns.
   - Two dimensions.
   - Created using `numpy.array()` with a list of lists or 2D array-like input.
3D Array:
   - Represents data organized in a three-dimensional cube or volume.
   - Three dimensions.
   - Created using `numpy.array()` with a nested list structure or 3D␣
 ↪array-like input.
Higher-Dimensional Arrays:
   - Represents data in more than three dimensions.
   - Used for complex data structures like volumetric data, 4D spacetime␣
 ↪simulations, or multidimensional arrays.
   - Created using `numpy.array()` with nested structures corresponding to the␣
 ↪desired dimensions.'''
scalar = array(42)  # This is a 0D array (scalar).
matrix = array([[1, 2, 3], [4, 5, 6]])  # This is a 2D array (matrix).
array_3d = array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])  # This is a 3D array.
array_4d = array([[[[1, 2], [3, 4]], [[5, 6], [7, 8]], [[9, 10], [11, 12]]],␣
 ↪[[[13, 14], [15, 16]], [[17, 18], [19, 20]], [[21, 22], [23, 24]]]])
print(f'scalar={scalar}\n matrix={matrix}\n array_3d={array_3d}\n␣
 ↪array_4d={array_4d}')
```

```
scalar=42
 matrix=[[1 2 3]
```

```
 [4 5 6]]
array_3d=[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
array_4d=[[[[ 1  2]
   [ 3  4]]

  [[ 5  6]
   [ 7  8]]

  [[ 9 10]
   [11 12]]]


 [[[13 14]
   [15 16]]

  [[17 18]
   [19 20]]

  [[21 22]
   [23 24]]]]
```

[12]:
```python
array_4d.ndim
```

[12]: 4

[13]:
```python
#When the array is created, you can define the number of dimensions by using␣
↪the ndmin argument.
arr = array([1, 2, 3, 4], ndmin=5)
print(arr)
print('number of dimensions :', arr.ndim)
```

```
[[[[[1 2 3 4]]]]]
number of dimensions : 5
```

[16]:
```python
# Create a 3x3 array filled with zeros
zeros_array = zeros((3, 3), dtype=int)
print(zeros_array)
# Create a 3x3 diagonal array with values 1, 2, and 3 on the diagonal
diag_array = diag([1, 2, 3])
print(diag_array)
```

```
[[0 0 0]
 [0 0 0]
 [0 0 0]]
```

```
[[1 0 0]
 [0 2 0]
 [0 0 3]]
```

[17]:
```python
# Create an array with values from 0 to 9 with a step of 2
arange_array = arange(0, 10, 2)
print(arange_array)
# Create an array with 5 evenly spaced values between 0 and 1
linspace_array = linspace(0, 1, 5)
print(linspace_array)
#For further such functions for generating arrays see attached Table 2.3
```

```
[0 2 4 6 8]
[0.   0.25 0.5  0.75 1.  ]
```

[19]:
```python
#Array Indexing & Slicing:
#Random Example: Access the third element of the second array of the first␣
↪array:
arr = array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(arr[0, 1, 2])
```

```
6
```

[ ]:
```python
'''Slicing in NumPy is a powerful technique for extracting and manipulating␣
↪specific portions of a NumPy array.
It allows you to create subarrays or views of the original array by specifying␣
↪a range of indices or slices along
one or more dimensions. Slicing is an essential tool for data manipulation and␣
↪extraction in NumPy. Note that it
creates view not copy of orginal array so chaning slice will change original␣
↪array. Here's an overview of how slicing works in NumPy:'''
```

[26]:
```python
#Basic Slicing:
'''You can use the colon (`:`) operator to specify a range of indices along a␣
↪single dimension.
   - Syntax: `array[start:stop]` or `array[start:stop:step]`
   - `start` is the index where the slice starts (inclusive).
   - `stop` is the index where the slice ends (exclusive).
   - `step` is the step size between elements (default is 1).'''
arr = array([0, 1, 2, 3, 4, 5])
sliced_arr = arr[1:4]  # Extract elements at indices 1, 2, and 3.
print(sliced_arr)
```

```
[1 2 3]
```

[27]:
```python
#Multi-Dimensional Slicing:
'''- You can slice along multiple dimensions by separating slices with commas.
```

```
   - Syntax: `array[start_row:stop_row, start_col:stop_col]`'''
matrix = array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
sliced_matrix = matrix[0:2, 1:3]  # Extract a submatrix.
print(sliced_matrix)
```

```
[[2 3]
 [5 6]]
```

[29]:
```
#Slicing with Steps:
''' - You can specify a step size to skip elements while slicing.
   - Syntax: `array[start:stop:step]`'''
arr = array([0, 1, 2, 3, 4, 5])
sliced_arr = arr[1:5:2]  # Extract elements at indices 1 and 3 with a step of 2.
print(sliced_arr)
```

```
[1 3]
```

[30]:
```
#Slicing with Ellipsis (`...`):
'''- You can use the ellipsis (`...`) to slice along multiple dimensions␣
 ↪without explicitly specifying all slice ranges.'''
arr = array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
sliced_arr = arr[..., 1]  # Extract the second column along the last dimension.
print(sliced_arr)
```

```
[[2 4]
 [6 8]]
```

[31]:
```
#Boolean Indexing:
'''- You can use boolean arrays to create conditional slices based on a␣
 ↪condition.'''
arr = array([1, 2, 3, 4, 5])
mask = arr > 2
sliced_arr = arr[mask]  # Extract elements greater than 2.
print(sliced_arr)
```

```
[3 4 5]
```

[ ]:
```
#For Visual Summary of indexing methods in NumPy see attached Fig. 2.1
```

[36]:
```
#Data Types:
'''The NumPy array object has an attribute called .dtype that returns the data␣
 ↪type of the array.
We use the array() function to create arrays, this function can take an␣
 ↪optional argument: dtype that
allows us to define the expected data type of the array elements:'''
arr = array([1, 2, 3, 4], dtype='str')
print(arr)
```

```
arr.dtype
```

```
['1' '2' '3' '4']
```

[36]:
```
dtype('<U1')
```

[43]:
```
'''copy() method creates a deep copy of an array, ensuring that changes to one␣
↪array do not affect the other.
In contrast, the view() method creates a shallow copy, where both arrays share␣
↪the same data but may have different metadata.
The `.base` attribute in NumPy helps determine if an array is a view or a copy␣
↪of another array.
If an array is a view, its `.base` attribute points to the original array,␣
↪indicating shared data.
If it's a copy, `.base` is `None`, signifying independent data. This␣
↪distinction is crucial for understanding
data relationships and avoiding unintended modifications in NumPy arrays. In␣
↪summary, attributes (i.e .base) are
specific to individual objects and store their state, methods (i.e .copy()) are␣
↪behaviors associated with objects of a class,
and built-in functions (i.e. array()) are general-purpose functions provided by␣
↪Python's standard library that can be applied
to various data types and objects.'''
arr = array([1, 2, 3, 4])
copyarr=arr.copy()
print(copyarr.base)
print(copyarr)
```

```
None
[1 2 3 4]
```

[45]:
```
#Changing data type of an array
arr = array([1, 2, 3, 4], dtype='str')
arrstr=arr.astype('str')
print(arrstr.base)
print(arrstr)
```

```
None
['1' '2' '3' '4']
```

[5]:
```
#Find
'''You can search an array for a certain value, and return the indexes that get␣
↪a match.
To search an array, use the where() function.'''
x=array([[[1,2,3,4],[4,5,6,8],[5,5,9,3]],[[1,2,4,6],[5,4,6,3],[4,5,9,3]]])
y=where(x==3)
print(y)
```

```
(array([0, 0, 1, 1], dtype=int64), array([0, 2, 1, 2], dtype=int64), array([2,
3, 3, 3], dtype=int64))
```

[6]:
```
#Sort:
'''The NumPy ndarray object has a function called sort(), that will sort a␣
↪Specified array.
This method returns a copy of the array, leaving the original array unchanged.
↪'''
arr = array([[3, 2, 4], [5, 0, 1]])
print(sort(arr))
```

```
[[2 3 4]
 [0 1 5]]
```

[10]:
```
#SearchSort:
'''There is a method called searchsorted() which performs a binary search in␣
↪the array, and returns the index where the
specified value would be inserted to maintain the search order. The␣
↪searchsorted() method is assumed to be used on sorted arrays.
##Find the indexes where the values 2, 4, and 6 should be inserted:'''
arr = array([1, 3, 5, 7])
x = searchsorted(arr, [2, 4, 6], side='right')
print(x)
```

```
[1 2 3]
```

[11]:
```
#Filtering Arrays:
'''Getting some elements out of an existing array and creating a new array out␣
↪of them is called filtering.
In NumPy, you filter an array using a boolean index list.A boolean index list␣
↪is a list of booleans corresponding
to indexes in the array. If the value at an index is True that element is␣
↪contained in the filtered array,
if the value at that index is False that element is excluded from the filtered␣
↪array.'''
arr = array([1, 2, 3, 4, 5, 6, 7])
filter_arr = arr % 2 == 0
newarr = arr[filter_arr]
print(filter_arr)
print(newarr)
```

```
[False  True False  True False  True False]
[2 4 6]
```

[12]:
```
#Shape:
'''The shape of an array is the number of elements in each dimension. NumPy␣
↪arrays have an attribute called .shape
```

```
that returns a tuple with each index having the number of corresponding␣
 ↪elements.'''
arr=array([[[1,2,3,5],[1,2,3,6]],[[3,4,5,3],[3,4,5,5]],[[8,5,2,9],[8,5,2,5]]])
print(arr.shape)
print(arr.ndim)
```

```
(3, 2, 4)
3
```

[13]:
```
#Reshaping:
'''It means changing the shape of an array. We can reshape arrays in NumPy␣
 ↪using the reshape() method:'''
arr=array([[[1,2,3,5],[1,2,3,6]],[[3,4,5,3],[3,4,5,5]],[[8,5,2,9],[8,5,2,5]]])
print(arr.shape)
print(arr.ndim)
arr2=arr.reshape(4,2,3)
print(arr2)
print(arr2.shape)
print(arr2.ndim)
```

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

 [[3 6 3]
  [4 5 3]]

 [[3 4 5]
  [5 8 5]]

 [[2 9 8]
  [5 2 5]]]
(4, 2, 3)
3
```

[18]:
```
'''There are rules and restrictions regarding reshaping arrays in NumPy. These␣
 ↪rules are important to ensure that
the reshaping operation is valid and that the total number of elements in the␣
 ↪original and reshaped arrays match.
Here are some key rules and considerations:
-Total Number of Elements: The total number of elements in the original array␣
 ↪must be the same as the total number
of elements in the reshaped array. In other words, the product of the␣
 ↪dimensions of the original array must equal
the product of the dimensions of the reshaped array. For example, a 1D array␣
 ↪with 12 elements can be reshaped into a
```

```
3x4 matrix because 12 = 3 x 4, but it cannot be reshaped into a 2x3 matrix because 12  2 x 3.

-Compatible Shapes: The dimensions of the original array and the desired reshaped array must be compatible.
For example, you can reshape a 1D array into a 2D array, but you cannot reshape a 1D array into a 3D array
unless the total number of elements allows for it.
- -1 Placeholder: You can use the -1 placeholder in one of the dimensions when reshaping, and NumPy will automatically
calculate the size for that dimension based on the total number of elements and the other dimensions. This can be handy
when you want NumPy to infer a dimension for you.

Note that .reshape() creates a new array with the desired shape without modifying the original, while .resize()
modifies the shape of the original array in-place and can add or remove elements as needed. The choice between
these methods depends on whether you want to preserve the original array or make permanent changes to it. i.e.'''

arr=array([[[1,2,3,5],[1,2,3,6]],[[3,4,5,3],[3,4,5,5]],[[8,5,2,9],[8,5,2,5]]])
arr.resize(4,2,3)
print(arr)
```

```
[[[1 2 3]
  [5 1 2]]

 [[3 6 3]
  [4 5 3]]

 [[3 4 5]
  [5 8 5]]

 [[2 9 8]
  [5 2 5]]]
```

[19]:

```
#Flattening array:
'''It means converting a multidimensional array into a 1D array. We can use reshape(-1) to do this.'''
arr=array([[[1,2,3,5],[1,2,3,6]],[[3,4,5,3],[3,4,5,5]],[[8,5,2,9],[8,5,2,5]]])
arr2=arr.reshape(-1)
print(arr2)
```

```
[1 2 3 5 1 2 3 6 3 4 5 3 3 4 5 5 8 5 2 9 8 5 2 5]
```

[20]:
```
#Iteration:
arr = array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
for x in arr:
    print(x)
```

```
[[1 2 3]
 [4 5 6]]
[[ 7  8  9]
 [10 11 12]]
```

[21]:
```
'''To Iterate Down to scalars we can use:'''
arr = array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
for x in arr:
  for y in x:
    for z in y:
      print(z)
```

```
1
2
3
4
5
6
7
8
9
10
11
12
```

[23]:
```
#or shortly we can use:
for x in nditer(arr):
  print(x)
```

```
1
2
3
4
5
6
7
8
9
10
11
12
```

[25]:
```
#To Iterate specific element:
for x in nditer(arr[:, ::2]):
  print(x)
```

```
1
2
3
7
8
9
```

[45]:
```
'''A discrete difference means subtracting two successive elements.
E.g. for [1, 2, 3, 4], the discrete difference would be [2-1, 3-2, 4-3] = [1,␣
↪1, 1]
To find the discrete difference, use the diff() function. We can perform this␣
↪operation repeatedly by giving parameter n.'''
arr = array([10, 15, 25, 5])
newarr = diff(arr, n=2)
print(newarr)
```

```
[  5 -30]
```

[ ]:
```
'''Some Important Points:
-For Matrices Multiplication use @ instead of *. All other Arithmetic␣
↪operations are element wise and simple.
-Also x=np.add(a1,a2)=a1+a2.
-The np.divmod(arr1,arr2) function return both the quotient and the mod. The␣
↪return value is two arrays,
the first array contains the quotient and the second array contains the mod.
-To find LCM of Two numbers x and y we use Lcm=np.lcm(x,y), while for elements␣
↪of an array arr we use LCM=np.lcm.reduce(arr) function.
The reduce() method will use the ufunc(universal), in this case the lcm()␣
↪function, on each element, and reduce the array by one dimension.
Same for gcd just replace lcm by gcd keyword.
-To find the sin of a number we use SIN=np.sin(np.pi/2) >>>1.0, we can apply␣
↪the same to array arr as Sin=np.sin(arr). Same for cos, tan,
arcsin, arccos,arctan,sinh,cosh,tanh. TO convert degree to rad we use x = np.
↪deg2rad(arr). Note: we used np with pi because pi is a maths
or numpy object.
-To find the unique values of two arrays a1,a2, use the np.union1d(a1,a2)␣
↪method, To find only the values that are present in both arrays,
use the np.intersect1d(a1,a2) method, To find only the values in the first set␣
↪that is NOT present in the seconds set, use the
np.setdiff1d(a1,a2) method and To find only the values that are NOT present in␣
↪BOTH sets (Symmetric Difference), use
the np.setxor1d(a1,a2) method. Note all these are applicable for 1D only.
For summary of all other functions see Table 2.6,7,8,12,13,5'''
```

[27]:
```
#Concatenation:
#Note this:
x=array([[1,2,3,4],[3,5,6,7],[4,5,7,9]])
y=ones((3,3),dtype='int')
z=[x,y]
print(f'x={x}\n \n y={y} \n z={z}')
```

```
x=[[1 2 3 4]
 [3 5 6 7]
 [4 5 7 9]]

 y=[[1 1 1]
 [1 1 1]
 [1 1 1]]
 z=[array([[1, 2, 3, 4],
       [3, 5, 6, 7],
       [4, 5, 7, 9]]), array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]])]
```

[35]:
```
'''Hence  [x,y] simply  joins two arrays and is equivalent to np.
 ↪concatenate([arr1,arr2], axis=0) is equivalent to
np.vstack(arr1,arr2) and it Stacks a list of arrays vertically (along axis 0):␣
 ↪for example, given a list of row vectors,
appends the rows to form a matrix. i.e'''
from numpy import array, ones, concatenate

# Create arrays x and y
x = array([[1, 2, 3, 4], [3, 5, 6, 7], [4, 5, 7, 9]])
y = ones((4, 4), dtype='int')

# Print arrays x and y
print(f'x={x}\n\ny={y}')

# Concatenate arrays x and y along axis 0 (rows)
z = concatenate([x, y], axis=0)

# Print the concatenated array
print("\nConcatenated Array:")
print(z)
```

```
x=[[1 2 3 4]
 [3 5 6 7]
 [4 5 7 9]]

y=[[1 1 1 1]
 [1 1 1 1]
```

```
 [1 1 1 1]
 [1 1 1 1]]

Concatenated Array:
[[1 2 3 4]
 [3 5 6 7]
 [4 5 7 9]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]
```

[36]:

```
'''np.concatenate([arr1,arr2], axis=1) is equivalent to np.hstack(arr1,arr2)␣
↪and it Stacks a list of arrays
horizontally (along axis 1): for example, given a list of column vectors,␣
↪appends the columns to form a matrix. i.e'''
# Create arrays x and y
x = array([[1, 2, 3, 4], [3, 5, 6, 7], [4, 5, 7, 9]])
y = ones((3, 4), dtype='int')

# Print arrays x and y
print(f'x={x}\n\ny={y}')

# Concatenate arrays x and y along axis 1 (columns)
z = concatenate([x, y], axis=1)

# Print the concatenated array
print("\nConcatenated Array:")
print(z)
```

```
x=[[1 2 3 4]
 [3 5 6 7]
 [4 5 7 9]]

y=[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]

Concatenated Array:
[[1 2 3 4 1 1 1 1]
 [3 5 6 7 1 1 1 1]
 [4 5 7 9 1 1 1 1]]
```

[38]:

```
'''np.dstack Stacks arrays depth-wise (along axis 2) i.e element wise stacking␣
↪based on indexing:'''
x = array([[1, 2, 3, 4], [3, 5, 6, 7], [4, 5, 7, 9]])
y = ones((3, 4), dtype='int')
```

```
print(f'x={x}\n\ny={y}')
z = dstack((x, y))
print("\nStacked Array:")
print(z)
```

```
x=[[1 2 3 4]
 [3 5 6 7]
 [4 5 7 9]]

y=[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]

Stacked Array:
[[[1 1]
  [2 1]
  [3 1]
  [4 1]]

 [[3 1]
  [5 1]
  [6 1]
  [7 1]]

 [[4 1]
  [5 1]
  [7 1]
  [9 1]]]
```

[40]:

```
'''Addition(add or +) is done between two arrays element wise whereas summation␣
 ↪happens over specified elements.
Similarly, Multiplication(multiply or *,@) is element wise b/w two or more␣
 ↪arrays while Product is over specified elements.
Add and multiply functions don't take axis arguments, while Sum and Product␣
 ↪take axis as arguments. Note that while
using an axis argument we combine arrays in a square [] bracket.'''
x=array([[1,2,3,4],[3,5,6,7],[4,5,7,9]])
print(f'x={x} \n y={y} \n Sum of Elements of x={sum(x)} \n Sum of Rows of␣
 ↪x={sum(x,axis=0)} \n Sum of Colums of x={sum(x,axis=1)}')
```

```
x=[[1 2 3 4]
 [3 5 6 7]
 [4 5 7 9]]
 y=[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]
 Sum of Elements of x=56
```

```
Sum of Rows of x=[ 8 12 16 20]
Sum of Colums of x=[10 21 25]
```

[44]:

```
x=array([[1,2,3,4],[3,5,6,7],[4,5,7,9]])
y=ones((3,4),dtype='int')
z=[x,y]
print(f'x={x} \n y={y} \n z={z} \n Sum of Elements of z={sum(z)} \n Sum of Rows␣
↪of z (Vertical Sum of Elements)={sum(z,axis=0)} \n Sum of Colums of␣
↪z(Horizontal Sum of Elements)={sum(z,axis=1)}')
```

```
x=[[1 2 3 4]
 [3 5 6 7]
 [4 5 7 9]]
 y=[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]
 z=[array([[1, 2, 3, 4],
       [3, 5, 6, 7],
       [4, 5, 7, 9]]), array([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]])]
 Sum of Elements of z=68
 Sum of Rows of z (Vertical Sum of Elements)=[[ 2  3  4  5]
 [ 4  6  7  8]
 [ 5  6  8 10]]
 Sum of Colums of z(Horizontal Sum of Elements)=[[ 8 12 16 20]
 [ 3  3  3  3]]
```

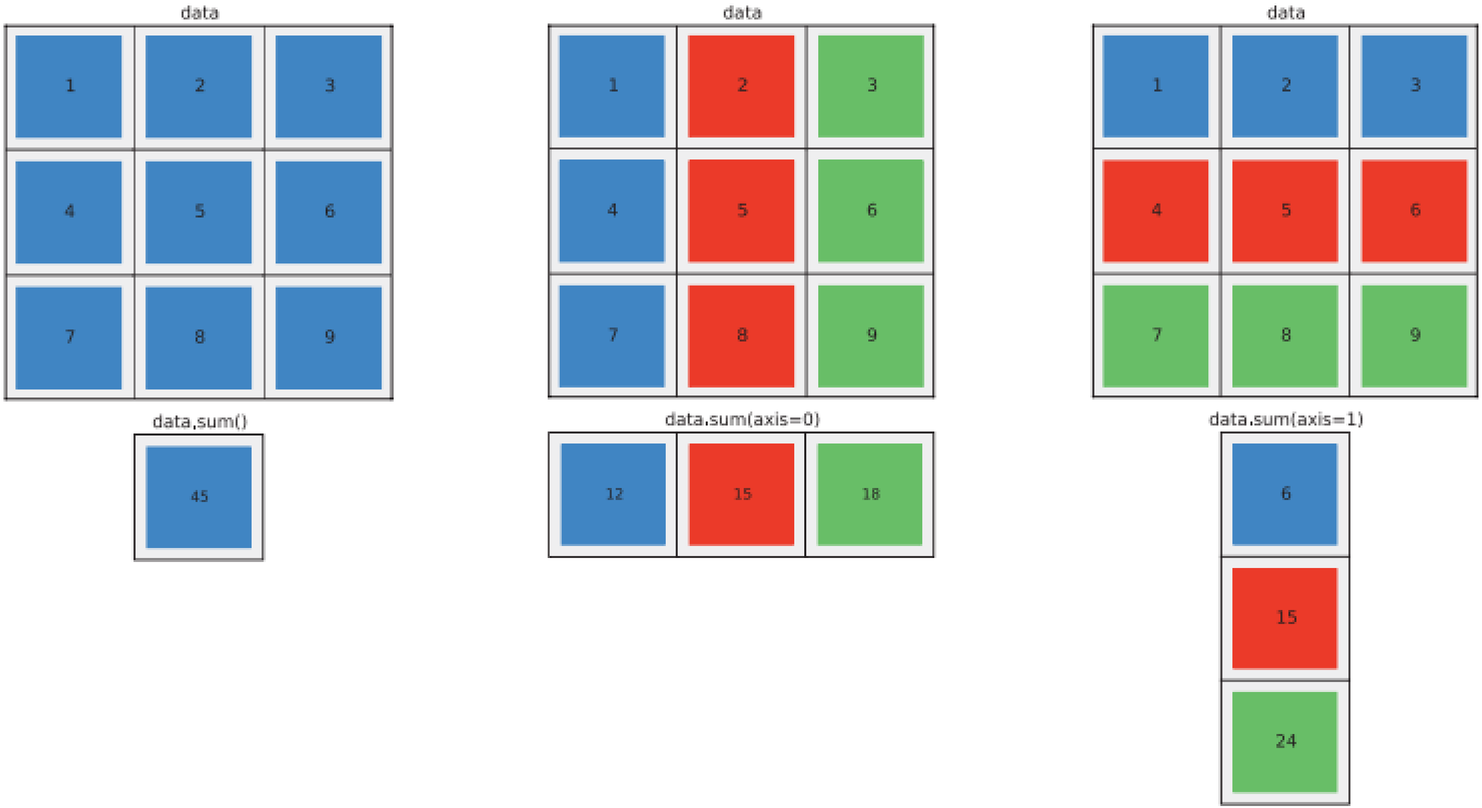

Figure 2-3. *Illustration of array aggregation functions along all axes (left), the first axis (center), and the second axis (right) of a two-dimensional array of shape 3 × 3*

Table 2-3. Summary of NumPy Functions for Generating Arrays

Function Name	Type of Array
`np.array`	Creates an array for which the elements are given by an array-like object, which, for example, can be a (nested) Python list, a tuple, an iterable sequence, or another `ndarray` instance.
`np.zeros`	Creates an array with the specified dimensions and data type that is filled with zeros.
`np.ones`	Creates an array with the specified dimensions and data type that is filled with ones.
`np.diag`	Creates a diagonal array with specified values along the diagonal and zeros elsewhere.
`np.arange`	Creates an array with evenly spaced values between the specified start, end, and increment values.
`np.linspace`	Creates an array with evenly spaced values between specified start and end values, using a specified number of elements.
`np.logspace`	Creates an array with values that are logarithmically spaced between the given start and end values.
`np.meshgrid`	Generates coordinate matrices (and higher-dimensional coordinate arrays) from one-dimensional coordinate vectors.
`np.fromfunction`	Creates an array and fills it with values specified by a given function, which is evaluated for each combination of indices for the given array size.
`np.fromfile`	Creates an array with the data from a binary (or text) file. NumPy also provides a corresponding function `np.tofile` with which NumPy arrays can be stored to disk and later read back using `np.fromfile`.
`np.genfromtxt`, `np.loadtxt`	Create an array from data read from a text file, for example, a comma-separated value (CSV) file. The function `np.genfromtxt` also supports data files with missing values.
`np.random.rand`	Generates an array with random numbers that are uniformly distributed between 0 and 1. Other types of distributions are also available in the `np.random` module.

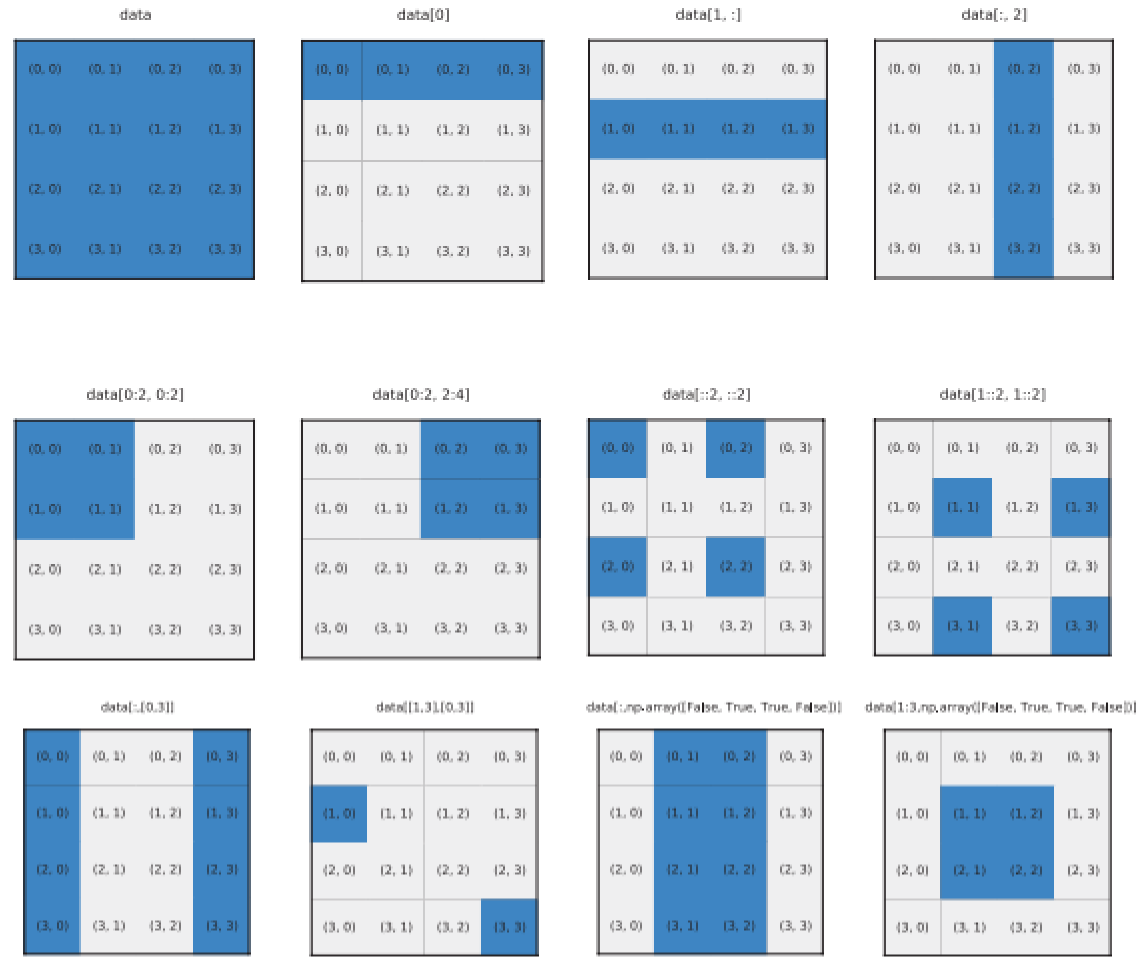

***Figure 2-1.** Visual summary of indexing methods for NumPy arrays. These diagrams represent NumPy arrays of shape (4, 4), and the highlighted elements are those that are selected using the indexing expression shown above the block representations of the arrays.*

Table 2-6. *Operators for Elementwise Arithmetic Operation on NumPy Arrays*

Operator	Operation
+, +=	Addition
-, -=	Subtraction
*, *=	Multiplication
/, /=	Division
//, //=	Integer division
**, **=	Exponentiation

Table 2-7. Selection of NumPy Functions for Elementwise Elementary Mathematical Functions

NumPy Function	Description
`np.cos, np.sin, np.tan`	Trigonometric functions.
`np.arccos, np.arcsin, np.arctan`	Inverse trigonometric functions.
`np.cosh, np.sinh, np.tanh`	Hyperbolic trigonometric functions.
`np.arccosh, np.arcsinh, np.arctanh`	Inverse hyperbolic trigonometric functions.
`np.sqrt`	Square root.
`np.exp`	Exponential.
`np.log, np.log2, np.log10`	Logarithms of base e, 2, and 10, respectively.

***Table 2-8.** Summary of NumPy Functions for Elementwise Mathematical Operations*

NumPy Function	Description
`np.add`, `np.subtract`, `np.multiply`, `np.divide`	Addition, subtraction, multiplication, and division of two NumPy arrays.
`np.power`	Raises first input argument to the power of the second input argument (applied elementwise).
`np.remainder`	The remainder of division.
`np.reciprocal`	The reciprocal (inverse) of each element.
`np.real`, `np.imag`, `np.conj`	The real part, imaginary part, and the complex conjugate of the elements in the input arrays.
`np.sign`, `np.abs`	The sign and the absolute value.
`np.floor`, `np.ceil`, `np.rint`	Convert to integer values.
`np.round`	Rounds to a given number of decimals.

Table 2-5. (*continued*)

Function/Method	Description
np.resize	Resizes an array. Creates a new copy of the original array, with the requested size. If necessary, the original array will be repeated to fill up the new array.
np.append	Appends an element to an array. Creates a new copy of the array.
np.insert	Inserts a new element at a given position. Creates a new copy of the array.
np.delete	Deletes an element at a given position. Creates a new copy of the array.

Table 2-13. Summary of NumPy Functions for Matrix Operations

NumPy Function	Description
`np.dot`	Matrix multiplication (dot product) between two given arrays representing vectors, arrays, or tensors.
`np.inner`	Scalar multiplication (inner product) between two arrays representing vectors.
`np.cross`	The cross product between two arrays that represent vectors.
`np.tensordot`	Dot product along specified axes of multidimensional arrays.
`np.outer`	Outer product (tensor product of vectors) between two arrays representing vectors.
`np.kron`	Kronecker product (tensor product of matrices) between arrays representing matrices and higher-dimensional arrays.
`np.einsum`	Evaluates Einstein's summation convention for multidimensional arrays.

***Table 2-12.** Summary of NumPy Functions for Array Operations*

Function	Description
`np.transpose`, `np.ndarray.transpose`, `np.ndarray.T`	The transpose (reverse axes) of an array.
`np.fliplr/np.flipud`	Reverse the elements in each row/column.
`np.rot90`	Rotates the elements along the first two axes by 90 degrees.
`np.sort`, `np.ndarray.sort`	Sort the elements of an array along a given specified axis (which default to the last axis of the array). The `np.ndarray` method sort performs the sorting in place, modifying the input array.

Table 2-9. NumPy Functions for Calculating Aggregates of NumPy Arrays

NumPy Function	Description
`np.mean`	The average of all values in the array.
`np.std`	Standard deviation.
`np.var`	Variance.
`np.sum`	Sum of all elements.
`np.prod`	Product of all elements.
`np.cumsum`	Cumulative sum of all elements.
`np.cumprod`	Cumulative product of all elements.
`np.min, np.max`	The minimum/maximum value in an array.
`np.argmin, np.argmax`	The index of the minimum/maximum value in an array.
`np.all`	Returns True if all elements in the argument array are nonzero.
`np.any`	Returns True if any of the elements in the argument array is nonzero.