42_2_NP_advanced_numpy_operations

December 15, 2023

1 NumPy Advanced Operations

1.1 Numpy Broadcasting

In NumPy, we can perform mathematical operations on arrays of different shapes. An array with a smaller shape is expanded to match the shape of a larger one. This is called broadcasting.

Let's see an example.

```
array1 = [1, 2, 3]
array2 = [[1], [2], [3]]
```

array1 is a 1-D array and array2 is a 2-D array. Let's perform addition between these two arrays of different shapes.

result = array1 + array2 Here, NumPy automatically broadcasts the size of a 1-D array array1 to perform element-wise addition with a 2-D array array2.

[1]:
```
import numpy as np

# create 1-D array
array1 = np.array([1, 2, 3])

# create 2-D array
array2 = np.array([[1], [2], [3]])

# add arrays of different dimension
# size of array1 expands to match with array2
sum = array1 + array2

print(sum)
```

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

In the example, we added two arrays with different dimensions. Numpy automatically expands the size of 1-D array array1 to match with the size of 2-D array array2.

Then, the element-wise addition is performed between two 2-D arrays.

1.1.1 Compatibility Rules for Broadcasting

Broadcasting only works with compatible arrays. NumPy compares a set of array dimensions from right to left.

Every set of dimensions must be compatible with the arrays to be broadcastable. A set of dimension lengths is compatible when

one of them has a length of 1 or they are equal

Let's see an example.

array1 = shape(6, 7) array2 = shape(6, 1) Here, array1 and array2 are arrays with different dimensions (6,7) and (6,1) respectively.

The dimension length 7 and 1 are compatible because one of them is 1.

Similarly, 6 and 6 are compatible since they are the same.

As both sets of dimensions are compatible, the arrays are broadcastable.

1.1.2 Examples of Broadcastable Shapes

Now, we'll see the list of broadcastable and non-broadcastable shapes.

Broadcastable Shapes

- (6, 7) and (6, 7)
- (6, 7) and (6, 1)
- (6, 7) and (7,)

Two arrays need not have the same number of dimensions to be broadcastable.

The last set of shapes is broadcastable because the right-most dimensions are both 7.

Non-Broadcastable Shapes

- (6, 7) and (7, 6)
- (6, 7) and (6,) The last set of shapes is not broadcastable because the right-most dimensions are not the same.

1.1.3 Broadcasting with Scalars

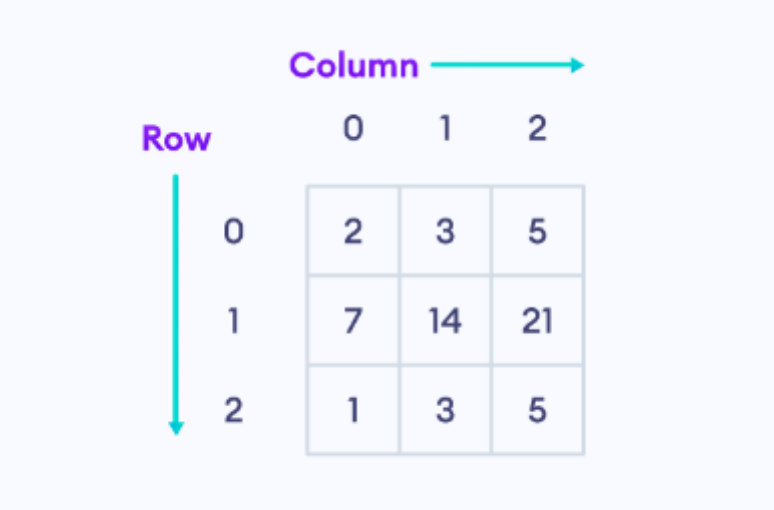
We can also perform mathematical operations between arrays and scalars (single values)

[ ]:
```
# 1-D array
array1 = np.array([1, 2, 3])

# scalar
number = 5

# add scalar and 1-D array
sum = array1 + number

print(sum)
```

In this example, NumPy automatically expands the scalar number to an 1-D array and then performs the element-wise addition.

1.2 NumPy Matrix Operations

A matrix is a two-dimensional data structure where numbers are arranged into rows and columns.



The above matrix is a 3x3 (pronounced "three by three") matrix because it has 3 rows and 3 columns.

It's also called a squared matrix because it has the same number of rows and columns.

NumPy Matrix Operations Here are some of the basic matrix operations provided by NumPy.

Functions	Descriptions
`array()`	Create a matrix
`dot()`	performs matrix multiplication
`transpose()`	Transpose a matrix
`trace()`	Find the sum of diagonal elements
`linalg.det()`	Find the determinant
`linalg.inv()`	Find the inverse
`flatten()`	Flatten a matrix
`reshape()`	Reshape a matrix

and many others function. See the full list of NumPy matrix functions here.

Create Matrix in NumPy In NumPy, we use the np.array() function to create a matrix.

[ ]:
```
# create a 2x2 matrix
matrix1 = np.array([[1, 3],
                    [5, 7]])

print("2x2 Matrix:\n",matrix1)
```

```
# create a 3x3  matrix
matrix2 = np.array([[2, 3, 5],
                    [7, 14, 21],
                    [1, 3, 5]])

print("\n3x3 Matrix:\n",matrix2)
```

Here, we have created two matrices: 2x2 matrix and 3x3 matrix by passing a list of lists to the np.array() function respectively.

1.2.1 Perform Matrix Multiplication in NumPy

We use the np.dot() function to perform multiplication between two matrices.

[4]:
```
# create two matrices
matrix1 = np.array([[1,2,3], [4,5,6], [7,8,9]])

matrix2 = np.array([[1,4],
                    [2,5], [3,6]])

# calculate the dot product of the two matrices
result = np.dot(matrix1, matrix2)

print("matrix1 x matrix2: \n",result)
```

```
matrix1 x matrix2:
 [[ 14  32]
 [ 32  77]
 [ 50 122]]
```

In this example, we have used the np.dot(matrix1, matrix2) function to perform matrix multiplication between two matrices: matrix1 and matrix2.

To learn more about Matrix multiplication, please visit NumPy Matrix Multiplication.

Note : We can only take a dot product of matrices when they have a common dimension size. For example, For A = (M x N) and B = (N x K) when we take a dot product of C = A . B the resulting matrix is of size C = (M x K).

Transpose NumPy Matrix The transpose of a matrix is a new matrix that is obtained by exchanging the rows and columns. For 2x2 matrix,

```
Matrix:
a11    a12
a21    a22

Transposed Matrix:
a11    a21
a12    a22
```

In NumPy, we can obtain the transpose of a matrix using the np.transpose() function.

2 create a matrix

matrix1 = np.array([[1, 3], [5, 7]])

3 get transpose of matrix1

result = np.transpose(matrix1)

print(result)

Note : Alternatively, we can use the .T attribute to get the transpose of a matrix. For example, if we used matrix1.T in our previous example, the result would be the same.

3.0.1 Calculate Inverse of a Matrix in NumPy

In NumPy, we use the np.linalg.inv() function to calculate the inverse of the given matrix.

However, it is important to note that not all matrices have an inverse. Only square matrices that have a non-zero determinant have an inverse.

Now, let's use np.linalg.inv() to calculate the inverse of a square matrix.

[ ]:
```
# create a 3x3 square matrix
matrix1 = np.array([[1, 3, 5],
                    [7, 9, 2],
                    [4, 6, 8]])

# find inverse of matrix1
result = np.linalg.inv(matrix1)

print(result)
```

Note : If we try to find the inverse of a non-square matrix, we will get an error message: numpy.linalg.linalgerror: Last 2 dimensions of the array must be square

3.0.2 Find Determinant of a Matrix in NumPy

We can find the determinant of a square matrix using the np.linalg.det() function to calculate the determinant of the given matrix.

Suppose we have a 2x2 matrix A:

```
a b
c d
```

So, the determinant of a 2x2 matrix will be:

```
det(A) = ad - bc
```

where a, b, c, and d are the elements of the matrix.

[ ]:
```
import numpy as np
```

```
# create a matrix
matrix1 = np.array([[1, 2, 3],
                    [4, 5, 1],
                   [2, 3, 4]])

# find determinant of matrix1
result = np.linalg.det(matrix1)

print(result)
```

3.0.3 Flatten Matrix in NumPy

Flattening a matrix simply means converting a matrix into a 1D array.

To flatten a matrix into a 1-D array we use the array.flatten() function.

[ ]:
```
# create a 2x3 matrix
matrix1 = np.array([[1, 2, 3],
                            [4, 5, 7]])

result = matrix1.flatten()
print("Flattened 2x3 matrix:", result)
```

Here, we have used the matrix1.flatten() function to flatten matrix1 into a 1D array, without compromising any of its elements

3.1 NumPy Vectorization

NumPy vectorization involves performing mathematical operations on entire arrays, eliminating the need to loop through individual elements.

We will see an overview of NumPy vectorization and demonstrate its advantages through examples.

3.1.1 NumPy Vectorization

We've used the concept of vectorization many times in NumPy. It refers to performing element-wise operations on arrays.

Let's take a simple example. When we add a number with a NumPy array, it adds up with each element of the array.

[5]:
```
array1 = np.array([1, 2, 3, 4, 5 ])
number = 10

#  number sums up with each array element
result = array1 + number

print(result)
```

```
[11 12 13 14 15]
```

Here, the number 10 adds up with each array element. This is possible because of vectorization.

Without vectorization, performing the operation would require the use of loops.

Example: Numpy Vectorization to Add Two Arrays Together

[6]:
```
# define two 2D arrays
array1 = np.array([[1, 2, 3], [4, 5, 6]])
array2 = np.array([[0, 1, 2], [0, 1, 2]])

# add two arrays (vectorization)
array_sum = array1 + array2

print("Sum between two arrays:\n", array_sum)
```

```
Sum between two arrays:
 [[1 3 5]
 [4 6 8]]
```

In this example, we have created two 2D arrays array1 and array2, and added them together.

This is a vectorized operation, where corresponding elements of two arrays are added together element-wise.

NumPy Vectorization Vs Python for Loop

Even though NumPy is a Python library, it inherited vectorization from C programming. As C is efficient in terms of speed and memory, NumPy vectorization is also much faster than Python.

Let's compare the time it takes to perform a vectorized operation with that of an equivalent loop-based operation.

Python for loop

[24]:
```
import time

start = time.time()

array1 = [1, 2, 3, 4, 5]

for i in range(len(array1)):
    array1[i] += 10

end = time.time()

print("For loop time:", end - start)
```

```
For loop time: 5.507469177246094e-05
```

NumPy Vectorization

[13]:
```
start = time.time()

array1 = np.array([1, 2, 3, 4, 5 ])
```

```
result = array1 + 10

end = time.time()

print("Vectorization time:", end - start)
```

```
Vectorization time: 0.0001442432403564453
```

Here, the difference in execution time between vectorization and a for loop is significant, even for simple operation.

This comparison illustrates the performance benefits of vectorization, especially when working with large datasets.

3.1.2 NumPy Vectorize() Function

In NumPy, every mathematical operation with arrays is automatically vectorized. So we don't always need to use the vectorize() function.

Let's take a scenario. You have an array and a function that returns the square of a positive number.

[ ]:
```
# array
array1 = np.array([-1, 0, 2, 3, 4])

# function that returns the square of a positive number
def find_square(x):
    if x < 0:
        return 0
    else:
        return x ** 2
```

Now, to apply the function find_square() to array1, we have two options: use a loop or vectorize the operation.

Since loops are complicated and slow by nature, it's efficient and convenient to use vectorize().

[25]:
```
# array whose square we need to find
array1 = np.array([-1, 0, 2, 3, 4])

# function to find the square
def find_square(x):
    if x < 0:
        return 0
    else:
        return x ** 2

# vectorize() to vectorize the function find_square()
vectorized_function = np.vectorize(find_square)
```

```
# passing an array to a vectorized function
result = vectorized_function(array1)

print(result)
```

```
[ 0  0  4  9 16]
```

In this example, we used the vectorize() function to vectorize the find_square() function. We then passed array1 as a parameter to the vectorized function.

3.2 NumPy Boolean Indexing

In NumPy, boolean indexing allows us to filter elements from an array based on a specific condition.

We use boolean masks to specify the condition.

Before we learn about boolean indexing, we need to know about boolean masks.

3.2.1 Boolean Masks in NumPy

Boolean mask is a numpy array containing truth values (True/False) that correspond to each element in the array.

Suppose we have an array named array1.

[27]:
```
array1 = np.array([12, 24, 16, 21, 32, 29, 7, 15])
```

Now let's create a mask that selects all elements of array1 that are greater than 20.

```
boolean_mask = array1 > 20
```

Here, array1 > 20 creates a boolean mask that evaluates to True for elements that are greater than 20, and False for elements that are less than or equal to 20.

The resulting mask is an array stored in the boolean_mask variable as:

[28]:
```
boolean_mask = array1 > 20
boolean_mask
```

[28]:
```
array([False,  True, False,  True,  True,  True, False, False])
```

3.2.2 1D Boolean Indexing in NumPy

Boolean Indexing allows us to create a filtered subset of an array by passing a boolean mask as an index.

The boolean mask selects only those elements in the array that have a True value at the corresponding index position.

Let's create a boolean indexing of the boolean mask in the above example.

[29]:
```
array1[boolean_mask]
```

[29]:
```
array([24, 21, 32, 29])
```

We'll use the boolean indexing to select only the odd numbers from an array.

[30]:
```
# create an array of numbers
array1 = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# create a boolean mask
boolean_mask = array1 % 2 != 0

# boolean indexing to filter the odd numbers
result = array1[boolean_mask]

print(result)

# Output: [ 1  3  5  7 9]
```

```
[1 3 5 7 9]
```

In this example, we have used the boolean indexing to select only the odd numbers from the array1 array.

Here, the expression numbers % 2 != 0 is a boolean mask. If the elements of array1 meet the condition specified in the boolean mask, it replaces the element (odd numbers) with True, and even numbers with False.

With boolean indexing, a filtered array with only the True valued elements is returned. Hence, we get an array with odd numbers.

Example: 1D Boolean Indexing in NumPy

[ ]:
```
# create an array of integers
array1 = np.array([1, 2, 4, 9, 11, 16, 18, 22, 26, 31, 33, 47, 51, 52])

# create a boolean mask using combined logical operators
boolean_mask = (array1 < 10) | (array1 > 40)

# apply the boolean mask to the array
result = array1[boolean_mask]

print(result)

# Output: [ 1  2  4  9 47 51 52]
```

Here, we have created a boolean mask using the | operator to select all the elements in array1 that are less than 10 or greater than 40.

Modify Elements Using Boolean Indexing In NumPy, we can use boolean indexing to modify elements of the array.

[31]:
```
# create an array of numbers
numbers = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# make a copy of the array
numbers_copy = numbers.copy()

# change all even numbers to 0 in the copy
numbers_copy[numbers % 2 == 0] = 0

# print the modified copy
print(numbers_copy)
```

```
[1 0 3 0 5 0 7 0 9 0]
```

Here, numbers_copy[numbers % 2 == 0] accesses all even numbers of the array and then we have assigned 0 to those numbers.

2D Boolean Indexing in NumPy Boolean indexing can also be applied to multi-dimensional arrays in NumPy.

[ ]:
```
# create a 2D  array
array1 = np.array([[1, 7, 9],
                   [14, 19, 21],
                   [25, 29, 35]])

# create a boolean mask based on the condition
# that elements are greater than 9
boolean_mask = array1 > 9

# select only the elements that satisfy the condition
result = array1[boolean_mask]

print(result)
```

In this example, we have applied boolean indexing to the 2D array named array1.

We then created boolean_mask based on the condition that elements are greater than 9. The resulting mask is,

```
[[False, False, False],
 [ True,  True,  True],
 [ True,  True,  True]]
```

We then use this boolean mask to index array1, which returns a flattened 1D array containing only the elements that satisfy the condition.

```
[14, 19, 21, 25, 29, 35]
```

3.3 NumPy Fancy Indexing

In NumPy, fancy indexing allows us to use an array of indices to access multiple array elements at once.

Fancy indexing can perform more advanced and efficient array operations, including conditional filtering, sorting, and so on.

3.3.1 Select Multiple Elements Using NumPy Fancy Indexing

[32]:
```
# create a numpy array
array1 = np.array([1, 2, 3, 4, 5, 6, 7, 8])

# select elements at index 1, 2, 5, 7
select_elements = array1[[1, 2, 5, 7]]

print(select_elements)
```

```
[2 3 6 8]
```

In this example, the resulting array select_elements contains the elements of array1 that correspond to the indices [1, 2, 5, 7] which are 2, 3, 6, and 8 respectively.

3.4 Example: NumPy Fancy Indexing

[36]:
```
array1 = np.array([1, 2, 3, 4, 5, 6, 7, 8])

# select a single element
simple_indexing = array1[3]

print("Simple Indexing:",simple_indexing)   # 4

# select multiple elements
fancy_indexing = array1[[1, 3, 5, 1]]

print("Fancy Indexing:",fancy_indexing)   # [2 3 6 8]
```

```
Simple Indexing: 4
Fancy Indexing: [2 4 6 2]
```

3.5 Fancy Indexing for Sorting NumPy Array

Fancy indexing can also sort a NumPy array.

[35]:
```
array1 = np.array([3, 2, 6, 1, 8, 5, 7, 4])

# sort array1 using fancy indexing
sorted_array = array1[np.argsort(array1)]

print(sorted_array)
```

```
[1 2 3 4 5 6 7 8]
```

Here, we are using the fancy indexing with the argsort() function to sort the array1 in the ascending order.

We could also use fancy indexing to sort the array in descending order.

[38]:
```
array1 = np.array([3, 2, 6, 1, 8, 5, 7, 4])

# sort array1 using fancy indexing in descending order
sorted_array = array1[np.argsort(-array1)]

print(sorted_array)
```

```
[8 7 6 5 4 3 2 1]
```

Here, first we multiplied array1 by -1 to sort in descending order and then used the fancy indexing to return the sorted array.

3.6 Fancy Indexing to Assign New Values to Specific Elements

We can also assign new values to specific elements of a NumPy array using fancy indexing.

[ ]:
```
array1 = np.array([3, 2, 6, 1, 8, 5, 7, 4])

# create a list of indices to assign new values
indices = [1, 3, 6]

# create a new array of values to assign
new_values = [10, 20, 30]

# use fancy indexing to assign new values to specific elements
array1[indices] = new_values

print(array1)

# Output: [ 3 10  6 20  8  5 30  4]
```

In this example, first we have created a list of indices called indices which specifies the elements of array1 that we want to assign new values to.

Then we created the array for new values called new_values that we want to assign to the specified indices.

Finally, we used fancy indexing with the list of indices to assign the new values to the specified elements of array1.

3.7 Fancy Indexing on N-d Arrays

We can also use fancy indexing on multi-dimensional arrays.

Let's see an example to select specific rows using fancy indexing.

[39]:
```
# create a 2D array
array1 = np.array([[1, 3, 5],
                   [11, 7, 9],
```

```
                [13, 18, 29]])

# create an array of row indices
row_indices = np.array([0, 2])

# use fancy indexing to select specific rows
selected_rows = array1[row_indices, :]

print(selected_rows)
```

```
[[ 1  3  5]
 [13 18 29]]
```

Here, we have created a 2D array named array1 and an array of row indices named row_indices. Then, we used fancy indexing to select the rows with indices 0 and 2 from array1.