# Numpy

August 21, 2022

## 1 Numpy

Numpy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays. If you are already familiar with MATLAB, you might find this manual useful to get started with Numpy.

### 1.0.1 Arrays

What are NumPy Arrays? NumPy is a Python package that stands for 'Numerical Python'. It is the core library for scientific computing, which contains a powerful n-dimensional array object.

Where is NumPy used? Python NumPy arrays provide tools for integrating C, C++, etc. It is also useful in linear algebra, random number capability etc. NumPy array can also be used as an efficient multi-dimensional container for generic data. Now, let me tell you what exactly is a Python NumPy array.

Python NumPy Array: Numpy array is a powerful N-dimensional array object which is in the form of rows and columns. We can initialize NumPy arrays from nested Python lists and access it elements. In order to perform these NumPy operations, the next question which will come in your mind is:

[1]:
```
import numpy as np

# Create a 1-D array
a = np.array([1, 2, 3])

# Prints "<class 'numpy.ndarray'>"
print(type(a))

# Prints "(3,)"
print(a.shape)

# Prints "1 2 3"
print(a[0], a[1], a[2])

# Change an element of the array
a[0] = 5

# Prints "[5, 2, 3]"
```

```
print(a)
```

```
<class 'numpy.ndarray'>
(3,)
1 2 3
[5 2 3]
```

**Why NumPy is used in Python?** We use python NumPy array instead of a list because of the below three reasons:

Less Memory Fast Convenient

The very first reason to choose python NumPy array is that;

```
1. It occupies less memory as compared to list.
2. It is pretty fast in terms of execution and at the same time.
3. It is very convenient to work with NumPy.
```

So these are the major advantages that Python NumPy array has over list. Consider the below example:

[2]:
```
import numpy as np

import sys

S = range(1000)
print(sys.getsizeof(S) * len(S))

D = np.arange(1000)
print(D.size * D.itemsize)
```

```
48000
4000
```

48000 4000

The above output shows that the memory allocated by list (denoted by S) is 48000 whereas the memory allocated by the NumPy array is just 4000. From this, you can conclude that there is a major difference between the two and this makes Python NumPy array as the preferred choice over list.

[3]:
```
import numpy as np

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

# Prints "(2, 3)"
print(b.shape)

# Prints "1 2 4"
print(b[0, 0], b[0, 1], b[1, 0])
```

```
(2, 3)
1 2 4
```

[4]:
```python
import numpy as np

# Create an array of all zeros
a = np.zeros((2, 2))

print(a)  # Prints "[[ 0.  0.]
                  #[ 0.  0.]]"

# Create an array of all ones
b = np.ones((1, 2))
print(b)  # Prints "[[ 1.  1.]]"

# Create a constant array
c = np.full((2, 2), 7)
print(c)  # Prints "[[ 7.  7.]
                  #[ 7.  7.]]"

# Create a 2x2 identity matrix
d = np.eye(2)
print(d)  # Prints "[[ 1.  0.]
                  #[ 0.  1.]]"

# Create an array filled with random values
e = np.random.random((2, 2))
print(e)
          # Might print "[[ 0.91940167  0.08143941]
                        #[ 0.68744134  0.87236687]]"
```

```
[[0. 0.]
 [0. 0.]]
[[1. 1.]]
[[7 7]
 [7 7]]
[[1. 0.]
 [0. 1.]]
[[0.36623498 0.52746082]
 [0.00818605 0.04093948]]
```

### 1.0.2 Array indexing

Numpy offers several ways to index into arrays.

Slicing: Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

[5]:
```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

print(b)

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])  # Prints "2"
b[0, 0] = 77  # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])  # Prints "77"
```

```
[[2 3]
 [6 7]]
2
77
```

You can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array. Note that this is quite different from the way that MATLAB handles array slicing:

[6]:
```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])

# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row1 = a[1, :]  # Rank 1 view of the second row of a
row2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row1, row1.shape)  # Prints "[5 6 7 8] (4,)"
print(row2, row2.shape)  # Prints "[[5 6 7 8]] (1, 4)"
```

```
# We can make the same distinction when accessing columns of an array:
print("We can make the same distinction when accessing columns of an array")
col1 = a[:, 1]
col2 = a[:, 1:2]
print(col1, col1.shape)  # Prints "[ 2  6 10] (3,)"
print(col2, col2.shape)  # Prints "[[ 2]
                                  #[ 6]
                                  #[10]] (3, 1)"
```

```
[5 6 7 8] (4,)
[[5 6 7 8]] (1, 4)
We can make the same distinction when accessing columns of an array
[ 2  6 10] (3,)
[[ 2]
 [ 6]
 [10]] (3, 1)
```

Integer array indexing: When you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array. In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array. Here is an example:

[7]:
```
import numpy as np

a = np.array([[1, 2], [3, 4], [5, 6]])
print(a)
print(a.shape)  # Prints "(3, 2)"

# indexing
print("indexing")
# An example of integer array indexing.
# The returned array will have shape (3,)
print(a[[0, 1, 2], [0, 1, 0]])

# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])  # Prints "[2 2]"

# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))  # Prints "[2 2]"
```

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

[8]:
```
import numpy as np

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

# An example of matrix
print(a)

# Create an array of indices
b = np.array([0, 2, 0, 1])
print(b)

# Create an array of indices
print('Create an array of indices')
b = np.array([0, 2, 0, 1])

# Select one element from each row of a using the indices in b
print('Select one element from each row of a using the indices in b')
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"

# Mutate one element from each row of a using the indices in b
print('Mutate one element from each row of a using the indices in b')
a[np.arange(4), b] += 10

print(a)  # prints "array([[11,  2,  3],
                        #[ 4,  5, 16],
                        #[17,  8,  9],
                        #[10, 21, 12]])
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
[0 2 0 1]
Create an array of indices
Select one element from each row of a using the indices in b
[ 1  6  7 11]
Mutate one element from each row of a using the indices in b
[[11  2  3]
 [ 4  5 16]
 [17  8  9]
 [10 21 12]]
```

### 1.0.3 Datatypes

Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype. Here is an example:

[9]:
```
import numpy as np

x = np.array([1, 2])  # Let numpy choose the datatype
print(x.dtype)  # Prints "int64"

x = np.array([1.0, 2.0])  # Let numpy choose the datatype
print(x.dtype)  # Prints "float64"

x = np.array([1, 2], dtype=np.int64)  # Force a particular datatype
print(x.dtype)  # Prints "int64"
```

```
int32
float64
int64
```

### 1.0.4 Copy and View in NumPy Array

While working with NumPy, you might have seen some functions return the copy whereas some functions return the view. The main difference between copy and view is that the copy is the new array whereas the view is the view of the original array. In other words, it can be said that the copy is physically stored at another location and view has the same memory location as the original array.

[10]:
```
import numpy as np

# creating array
arr = np.array([2, 4, 6, 8, 10])

# assigning arr to nc
nc = arr

# both arr and nc have same id
print("id of arr", id(arr))
print("id of nc", id(nc))

# updating nc
nc[0] = 12

# printing the values
print("original array: ", arr)
print("assigned array: ", nc)
```

```
id of arr 2244484784368
id of nc 2244484784368
original array:  [12  4  6  8 10]
assigned array:  [12  4  6  8 10]
```

[11]:
```
import numpy as np

# creating array
arr = np.array([2, 4, 6, 8, 10])

# creating view
v = arr.view()

# both arr and v have different id
print("id of arr", id(arr))
print("id of v", id(v))

# changing original array
# will effect view
arr[0] = 12

# printing array and view
print("original array: ", arr)
print("view: ", v)
```

```
id of arr 2244485380688
id of v 2244485379056
original array:  [12  4  6  8 10]
view:  [12  4  6  8 10]
```

### 1.0.5 NumPy Array Iterating

Iterating means going through elements one by one.

As we deal with multi-dimensional arrays in numpy, we can do this using basic for loop of python.

If we iterate on a 1-D array it will go through each element one by one.

[12]:
```
import numpy as np

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

for x in arr:
    print(x)
```

```
1
2
3
```

[13]:
```
# In a 2-D array it will go through all the rows.

import numpy as np

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

```
for x in arr:
    print(x)
```

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

[14]:
```
# Iterate on each scalar element of the 2-D array:

import numpy as np

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

for x in arr:
    for y in x:
        print(y)
```

```
1
2
3
4
5
6
```

[15]:
```
import numpy as np

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

for x in np.nditer(arr):
    print(x, end=' ')
```

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

### 1.0.6 Reshape

[16]:
```
# Convert the following 1-D array with 12 elements into a 2-D array.

# The outermost dimension will have 4 arrays, each with 3 elements:

import numpy as np

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

newarr = arr.reshape(4, 3)

print(newarr)
```

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

```
 [ 7  8  9]
 [10 11 12]]
```

[17]:
```
# Convert the following 1-D array with 12 elements into a 3-D array.

# The outermost dimension will have 2 arrays that contains 3 arrays, each with␣
 ↪2 elements:

import numpy as np

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

newarr = arr.reshape(2, 3, 2)

print(newarr)
```

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

 [[ 7  8]
  [ 9 10]
  [11 12]]]
```

[18]:
```
# Flattening the arrays
# Flattening array means converting a multidimensional array into a 1D array.
# We can use reshape(-1) to do this.

# Convert the array into a 1D array:

import numpy as np

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

newarr = arr.reshape(-1)

print(newarr)
```

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

### 1.0.7 Joining NumPy Arrays

Joining means putting contents of two or more arrays in a single array.

In SQL we join tables based on a key, whereas in NumPy we join arrays by axes.

We pass a sequence of arrays that we want to join to the concatenate() function, along with the axis. If axis is not explicitly passed, it is taken as 0.

[19]:
```
import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.concatenate((arr1, arr2))

print(arr)
```

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

[20]:
```
# Join two 2-D arrays along rows (axis=1):

import numpy as np

arr1 = np.array([[1, 2], [3, 4]])

arr2 = np.array([[5, 6], [7, 8]])

arr = np.concatenate((arr1, arr2), axis=1)

print(arr)
```

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

### 1.0.8 NumPy Splitting Array

Splitting is reverse operation of Joining.

Joining merges multiple arrays into one and Splitting breaks one array into multiple.

We use array_split() for splitting arrays, we pass it the array we want to split and the number of splits.

[21]:
```
# Split the array in 3 parts:

import numpy as np

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

newarr = np.array_split(arr, 3)

print(newarr)
print(type(newarr))
print(newarr[0])
print(newarr[1])
print(newarr[2])
```

```
[array([1, 2]), array([3, 4]), array([5, 6])]
<class 'list'>
[1 2]
[3 4]
[5 6]
```

[22]:
```
# Split the 2-D array into three 2-D arrays.

import numpy as np

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

newarr = np.array_split(arr, 3)

print(newarr)
print(type(newarr))
print(newarr[0])
print(newarr[1])
print(newarr[2])
```

```
[array([[1, 2],
       [3, 4]]), array([[5, 6],
       [7, 8]]), array([[ 9, 10],
       [11, 12]])]
<class 'list'>
[[1 2]
 [3 4]]
[[5 6]
 [7 8]]
[[ 9 10]
 [11 12]]
```

### 1.0.9 NumPy Searching Arrays

Searching Arrays You can search an array for a certain value, and return the indexes that get a match.

To search an array, use the where() method.

[23]:
```
# Find the indexes where the value is 4:

import numpy as np

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

x = np.where(arr == 4)

print(x)
```

```
# The example above will return a tuple: (array([3, 5, 6],)
# Which means that the value 4 is present at index 3, 5, and 6.
```

```
(array([3, 5, 6], dtype=int64),)
```

[24]:
```
import numpy as np

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

x = np.where(arr % 2 == 0)

print(x)
```

```
(array([1, 3, 5, 7], dtype=int64),)
```

### 1.0.10 Sorting Arrays

Sorting means putting elements in an ordered sequence.

Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.

The NumPy ndarray object has a function called sort(), that will sort a specified array.

[25]:
```
# Sort the array:

import numpy as np

arr = np.array([3, 2, 0, 1])

print(np.sort(arr))
```

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

[26]:
```
# You can also sort arrays of strings, or any other data type:
# Sort the array alphabetically:

import numpy as np

arr = np.array(['banana', 'cherry', 'apple'])

print(np.sort(arr))
```

```
['apple' 'banana' 'cherry']
```

[27]:
```
# Sort a 2-D array:

import numpy as np

arr = np.array([[3, 2, 4], [5, 0, 1]])
```

```
print(np.sort(arr))
```

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