Programming for Engineers

Lecture 4 - NumPy

Radoslav Škoviera

Table of contents

Lecture 4 - NumPy

Introduction to NumPy

The `NumPy` library enhances the Python with support for numerical operations. It is especially useful for large (multi-dimensional) arrays and matrices. Much of it is actually programmed in C and uses vectorization (parallelization) for efficient computations. The arrays are typed, which puts a restriction on what can be stored in them (unlike Python lists). On the other hand, they are memory efficient and many operations run faster on them. They also allow special manipulation, like **reshaping** and **broadcasting**.

This lecture covers only the basic operations and concepts. Numpy offers many other functions, operations, tricks and convenience methods. You can find more information in the NumPy documentation.

Installation: `pip install numpy`

Import:

```
import numpy as np
```

Array Creation

1D Arrays

From Python Lists:

```
a = np.array([1, 2, 3, 4, 5])
print("1D numpy array:", a)
```

```
1D numpy array: [1 2 3 4 5]
```

Direct Creation with `np.r_`:

```
a = np.r_[1, 2, 3, 4, 5]
print("1D numpy array:", a)
```

```
1D numpy array: [1 2 3 4 5]
```

The `np.r_` function (notice, it's actually not a function) creates a 1D array from a list of values. It can also be used to create more complex arrays (reader is encouraged to check the NumPy documentation for more details).

Using Ranges and Linspace:

```
a_range = np.arange(0, 10, 2)  # same as np.array(range(0, 10, 2))
a_range_float = np.arange(0.5, 11.5, 2.5)  # also support floats
a_lin = np.linspace(0, 1, 5)  # 5 values, evenly spaced between 0 to 1
print("arange:", a_range)
print("arange with floats:", a_range_float)
print("linspace:", a_lin)
```

```
arange: [0 2 4 6 8]
arange with floats: [ 0.5  3.   5.5  8.  10.5]
linspace: [0.   0.25 0.5  0.75 1.  ]
```

The function `np.arange` has the same signature as the Python `range` function: `np.arange(start, stop, step)`. The `np.linspace` function creates a sequence of evenly-spaced values, between start and stop. The signature is `np.linspace(start, stop, num=100, endpoint=True)`, where `num` is the number of samples to generate and `endpoint` is a boolean that indicates whether to include the stop value in the sequence (`False` for similar behavior to `range`).

`np.r_` can also be used to generate sequences, if instead of a sequence of numbers, a slice is passed as an argument. This slice has the same format as in the Python `range` function: `start:stop:step`. However, unlike for `range` the values do not have integers and they can even be **imaginary numbers**, which makes this operation behave like `linspace`.

```
a = np.r_[1:10:2]
b = np.r_[0:1:0.1]
c = np.r_[0:1:5j]
print("values from 1 to 10, with step 2:", a)
print("values from 0 to 1, with step 0.1:", b)
print("5 values, evenly spaced between 0 and 1:", c)
```

```
values from 1 to 10, with step 2: [1 3 5 7 9]
values from 0 to 1, with step 0.1: [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]
5 values, evenly spaced between 0 and 1: [0.   0.25 0.5  0.75 1.  ]
```

`np.r_` can also be used to quickly combine multiple lists and arrays into a 1D array - by default, it "flattens" all the input arrays and lists.

```
a = [1, 2, 3]  # a Python list
b = np.array([4, 5, 6])  # a 1D numpy array
c = np.r_[7, 8, 9]  # a 1D numpy array

d = np.r_[a, b, c]
print("combined array:", d)
```

```
combined array: [1 2 3 4 5 6 7 8 9]
```

You can even combine both the range creation and concatenation of lists and arrays:

```
e = np.r_[a, b, c, 10, 11, 12:20:2, 20:30:6j]
print("combined array:", e)
```

```
combined array: [ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 14. 16. 18. 20. 22. 24.
 26. 28. 30.]
```

Pre-filled Arrays:

NumPy supports several array *pre-allocation* methods.

```
zeros = np.zeros(5)
ones = np.ones(5)
sevens = np.full(5, 7)  # the same as np.ones(5) * 7
print("Zeros:", zeros)
print("Ones:", ones)
print("Sevens:", sevens)
```

```
Zeros: [0. 0. 0. 0. 0.]
Ones: [1. 1. 1. 1. 1.]
Sevens: [7 7 7 7 7]
```

Random Arrays:

```
rand = np.random.rand(5)
randn = np.random.randn(5)
randint = np.random.randint(0, 10, 5)
print("5 random values in range [0, 1):", rand)
print("5 random normally distributed values:", randn)
print("5 random integers in range [0, 10):", randint)
```

```
5 random values in range [0, 1): [0.88766782 0.687989   0.02337783 0.33334182 0.9142708 ]
5 random normally distributed values: [ 1.59592662 -0.41243968 -0.23824128  2.66023854  0.28
5 random integers in range [0, 10): [2 6 9 2 5]
```

Newest NumPy versions actually encourage the use of random generators from the `numpy.random` module. However, for the purpose of this course, we will not cover this in detail.

2D Arrays (Matrices)

From Nested Lists:

```python
array_2d = np.array(
    [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
    ]
)
print("2D Array:\n", array_2d)
```

```
2D Array:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
```

The `numpy.matrix` Class

There is also a specific class for matrices in numpy - `numpy.matrix`. It contains some matrix-specific syntax sugar but it is also more "cumbersome" and thus normally not recommended (unless you need nicer syntax and outputs).

```python
matrix = np.matrix(array_2d)
print("Matrix:\n", matrix)
```

```
Matrix:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
```

One of the 'advantages' is that standard operations on matrix are automatically treated as matrix operations. For example, multiplication and division for arrays are elementwise, while in matrices they are matrix multiplication and matrix division.

```
print("Array multiplication:\n", array_2d * array_2d)
print("Matrix multiplication:\n", matrix * matrix)
print("Element-wise matrix multiplication:\n", np.multiply(matrix, matrix))
```

```
Array multiplication:
 [[ 1  4  9]
 [16 25 36]
 [49 64 81]]
Matrix multiplication:
 [[ 30  36  42]
 [ 66  81  96]
 [102 126 150]]
Element-wise matrix multiplication:
 [[ 1  4  9]
 [16 25 36]
 [49 64 81]]
```

Pre-filled 2D Arrays

Similarly to 1D arrays, you can create pre-filled and random 2D arrays:

```
# all matrices have 3 rows and 4 columns
zeros = np.zeros((3, 4))
ones = np.ones((3, 4))
sevens = np.full((3, 4), 7)
rand = np.random.rand(3,4)
randn = np.random.randn(3,4)
randint = np.random.randint(0, 10, (3, 4))
print("Zeros:\n", zeros)
print("Ones:\n", ones)
print("Sevens:\n", sevens)
print("Random:\n", rand)
print("Random normally distributed:\n", randn)
print("Random integers:\n", randint)
```

```
Zeros:
 [[0. 0. 0. 0.]
```

```
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
Ones:
 [[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
Sevens:
 [[7 7 7 7]
 [7 7 7 7]
 [7 7 7 7]]
Random:
 [[0.49963911 0.59995266 0.00703725 0.01548836]
 [0.01435928 0.41755333 0.46231065 0.05872243]
 [0.38411893 0.53294835 0.97284708 0.35052676]]
Random normally distributed:
 [[ 0.67528451  1.34398953 -0.20041165 -0.9722687 ]
 [-0.39727817 -0.78475693  0.69718927  0.7500202 ]
 [ 1.19377205 -0.7074143  -0.8283151  -0.38471126]]
Random integers:
 [[5 3 6 2]
 [9 3 9 1]
 [8 1 9 1]]
```

Array size

The default function `len` will only reveal the number of rows of the array, not the number of elements (this is because of how iterating over the arrays work). To get the total number of elements, use the `size` attribute:

```
a = np.random.rand(2, 3, 4)
print("Array size:", a.size)  # = 2 * 3 * 4
```

```
Array size: 24
```

To see the sizes of the array along each dimension, you can use the `shape` attribute:

```
a = np.random.rand(2, 3, 4)
print("Array shape:", a.shape)
```

```
Array shape: (2, 3, 4)
```

Finally, to get the number of dimensions of the array, use the `ndim` attribute:

```
a = np.random.rand(2, 3, 4)
print("Array dimensions:", a.ndim)
```

```
Array dimensions: 3
```

Reshaping 1D Arrays

You can also take a 1D array and **reshape** it into a 2D array with the `reshape` array method. The reshape takes the height and width (number of rows and columns) as arguments. And example of use case is getting a 1D list of byte values and reshaping it into a 2D array of pixel intensities for an image. This is how some image formats store images. Reshape organizes the values row-by-row. You have to make sure that there is enough values to fill the array, i.e., `height * width == len(array_1d)`.

```
a = np.arange(12).reshape(3, 4)
print("Reshaped array:\n", a)
```

```
Reshaped array:
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

The following would throw an error, because there is not enough values to create a 2D array of the shape (4, 5):

```
a = np.arange(12).reshape(4, 5)
```

You can also **omit** one of the dimensions when reshaping, in which case the other dimension will be calculated automatically. In this case, the length of the input (1D) array must be divisible by the specified value.

```
a = np.arange(12)
print("Reshaped array (3, -1):\n", a.reshape(3, -1))
print("Reshaped array (-1, 4):\n", a.reshape(-1, 4))
```

```
Reshaped array (3, -1):
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
Reshaped array (-1, 4):
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

The following would throw an error, because the length of the input array length is not divisible by 5:

```
a = np.arange(12).reshape(-1, 5)
a = np.arange(12).reshape(5, -1)
```

```
# Column-wise reshape
a = np.arange(12).reshape(3, 4, order="F")
print("Column-wise reshaped array:\n", a)
print("...or using transposition 'a.T':\n", np.arange(12).reshape(4, 3, order="C").T)
# notice swapped shape dimensions
```

```
Column-wise reshaped array:
 [[ 0  3  6  9]
 [ 1  4  7 10]
 [ 2  5  8 11]]
...or using transposition 'a.T':
 [[ 0  3  6  9]
 [ 1  4  7 10]
 [ 2  5  8 11]]
```

There are some performance nuances about **contiguous** arrays but that is beyond the scope of this course.

Indexing trick `np.r_`

The `np.r_` operator can be used to create a 2D array as well. If all inputs are 2D, they can be directly concatenated. The concatenation is row-wise by default.

```
a = np.arange(6).reshape(2, -1)
b = np.arange(6, 12).reshape(2, -1)
print(f"a =\n{a}")
print(f"b =\n{b}")
print("Concatenated:\n", np.r_[a, b])
```

```
a =
[[0 1 2]
 [3 4 5]]
b =
[[ 6  7  8]
 [ 9 10 11]]
Concatenated:
 [[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```

The inputs can be also 1D arrays or a mix of 1D and 2D arrays. In this case, a string **directive** is specified as the first "argument". It determines how the arrays are concatenated. The format of the directive is as follows:

```
'axis,min_dim,transpose_spec'
```

where: * `axis` - the **axis** along which the arrays (remaining arguments) are **concatenated** * `min_dim` is the **minimum dimension**. Inputs with lower dimension are first "upgraded" to this dimension * `transpose_spec` specifies how lower-dimensional inputs are transposed before concatenation. It can be omitted, in which case it is set to `-1`.

For example, a row-wise stacking into a 2D array would be specified as:

```
np.r_['0,2', a, b, ...]
```

where `a` and `b` are 1D arrays. Note that the concatenated arrays must have the same length.

For the `axis`, 0 means row-wise and 1 means column-wise (can be higher for multi-dimensional arrays). For 2D arrays, the `min_dim` must be set to 2. This means that all 1D inputs are first "upgraded" to 2D. Already 2D inputs are not changed. The last argument specifies "where to put" the upgraded 1D inputs. The default value is `-1`, which means that for 2D case. the 1D inputs are upgraded to have the dimension of [1, N], where N is the length of the 1D input. If we change it to `0`, the 1D inputs are upgraded to have the dimension of [N, 1]. This happens before concatenation and this it will affect how the result will be shaped.

For example, for K inputs, each being 1D with length N, the spec “0,2,-1” will result in a 2D array of shape (K, N): - row-wise stacking - upgrade 1D inputs to 2D with dimension [1, N] (the last “-1” means “put the 1D inputs into the last dimension”)

The spec “1,2,0” will result in a 2D array of shape (N, K): - column-wise stacking - upgrade 1D inputs to 2D with dimension [N, 1] (the last “0” means “put the 1D inputs into the first dimension”)

To make it perhaps more clear, for 3D case, the spec “0,3,-1” (the same as writing “0,3”) will upgrade 1D input of length N to a 3D array of shape [1, 1, N]. It will also upgrade 2D inputs of shape [M, N] to a 3D array of shape [1, M, N]. The spec “0,3,0” will upgrade 1D input of length N to a 3D array of shape [N, 1, 1]. It will also upgrade 2D inputs of shape [M, N] to a 3D array of shape [M, N, 1]. Lastly, the spec “0,3,1” will upgrade 1D input of length N to a 3D array of shape [1, N, 1]. For 2D inputs, it is the same as “0,3,-1”.

Additionally, you can also “append” 2D arrays or lists. Although, these must also have each row (or column) of the same length.

```
# Given our previous example:
c = np.r_[a, b]
print("Concatenated along axis '0':", c, c.shape, sep="\n")
c = np.r_["1", a, b]
print("Concatenated along axis '1':", c, c.shape, sep="\n")
c = np.r_["2,3", a, b]
print("Concatenated along axis '2':", c, c.shape, sep="\n")
c = np.r_["0,3", a, b]
print("Concatenated along axis '0' but *upgraded* to 3 dims:", c, c.shape, sep="\n")
print()
d, e = np.ones((1, 3)), np.zeros((3, ))
print("d shape:", d.shape)
print("e shape:", e.shape)
c = np.r_["1,2,1", d, e]  # just "1,2" wouldn't work because zeros will have bad shape
print("Concatenation of arrays with different dims", c, c.shape, sep="\n")
```

```
Concatenated along axis '0':
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
(4, 3)
Concatenated along axis '1':
[[ 0  1  2  6  7  8]
 [ 3  4  5  9 10 11]]
(2, 6)
```

```
Concatenated along axis '2':
[[[ 0  1  2  6  7  8]
  [ 3  4  5  9 10 11]]]
(1, 2, 6)
Concatenated along axis '0' but *upgraded* to 3 dims:
[[[ 0  1  2]
  [ 3  4  5]]

 [[ 6  7  8]
  [ 9 10 11]]]
(2, 2, 3)

d shape: (1, 3)
e shape: (3,)
Concatenation of arrays with different dims
[[1. 1. 1. 0. 0. 0.]]
(1, 6)
```

Here, we combine multiple items into one 2D array.

```
a = np.array([1, 2, 3])
c = [7, 8, 9]
d = np.r_[
    '0,2',  # row-wise 2D array
    a,      # 1D array
    4:7,    # range(4, 7)
    c,      # list
    [10, 11, 12],  # list, directly specified
    [[13, 14, 15], [16, 17, 18]],  # 2D list
    np.array([[19, 20, 21], [22, 23, 24]]),  # 2D array
    25:30:3j  # range with 3 evenly spaced values
]
print("Row-wise stacking:\n", d)
```

```
Row-wise stacking:
 [[ 1.   2.   3. ]
 [ 4.   5.   6. ]
 [ 7.   8.   9. ]
 [10.  11.  12. ]
 [13.  14.  15. ]
 [16.  17.  18. ]
 [19.  20.  21. ]
```

```
 [22.  23.  24. ]
 [25.  27.5 30. ]]
```

You can also stack the items column-wise:

```
e = np.r_[
    '1,2,0',  # column-wise 2D array
    a,      # 1D array
    4:7,    # range(4, 7)
    c,      # list
    [10, 11, 12],  # list, directly specified
    [[13, 14], [15, 16], [17, 18]],  # 2D list
    np.array([[19, 20], [21, 22], [23, 24]]),  # 2D array
    25:30:3j  # range with 3 evenly spaced values
]
print("Column-wise stacking:\n", e)
```

```
Column-wise stacking:
 [[ 1.   4.   7.  10.  13.  14.  19.  20.  25. ]
 [ 2.   5.   8.  11.  15.  16.  21.  22.  27.5]
 [ 3.   6.   9.  12.  17.  18.  23.  24.  30. ]]
```

Note that the 2D arrays must have the same number of columns, in the column.wise stacking case.

More explanation of `np.r_` can be found at https://rmoralesdelgado.com/all/numpy-concatenate-r_-c_/.

3D Arrays

All the previously shown methods for 1 an 2D arrays also work for 3D arrays. Actually, you can create an array of any dimension with similar techniques.

From Nested Lists:

```
array_3d = np.array(
    [
        [  # first 'slice' of the 3D matrix
            [1, 2, 3], # first row of the first slice
            [4, 5, 6],
```

```
            [7, 8, 9]
        ],
        [  # second 'slice' of the 3D matrix
            [10, 11, 12], # first row of the second slice
            [13, 14, 15],
            [16, 17, 18]
        ]
    ]
)
print("3D Array:\n", array_3d)
```

```
3D Array:
 [[[ 1  2  3]
  [ 4  5  6]
  [ 7  8  9]]

 [[10 11 12]
  [13 14 15]
  [16 17 18]]]
```

From 1D arrays using reshape

Similarly to 2D, make sure that there is enough values to create a 3D array of the shape (height, width, depth). Also, you can omit one of the dimensions when reshaping, in which case the other dimension will be calculated automatically.

```
a = np.arange(24).reshape(2, 3, 4)
print("3D Array shape:", a.shape)
print(a)
```

```
3D Array shape: (2, 3, 4)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

Pre-filled 3D Arrays

Each of these arrays has a shape of (2, 3, 4)

```
a = np.zeros((2, 3, 4))
b = np.ones((2, 3, 4))
c = np.full((2, 3, 4), 5)
d = np.random.rand(2, 3, 4)
e = np.random.randn(2, 3, 4)
f = np.random.randint(0, 10, (2, 3, 4))
print("Zeros:\n", a)
print("Ones:\n", b)
print("Fives:\n", c)
print("Random:\n", d)
print("Random normal:\n", e)
print("Random integers:\n", f)
```

```
Zeros:
 [[[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]

 [[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]]
Ones:
 [[[1. 1. 1. 1.]
  [1. 1. 1. 1.]
  [1. 1. 1. 1.]]

 [[1. 1. 1. 1.]
  [1. 1. 1. 1.]
  [1. 1. 1. 1.]]]
Fives:
 [[[5 5 5 5]
  [5 5 5 5]
  [5 5 5 5]]

 [[5 5 5 5]
  [5 5 5 5]
  [5 5 5 5]]]
Random:
 [[[0.50752861 0.95789564 0.56725651 0.53874782]
```

```
  [0.02022099 0.5292411  0.06961163 0.50449079]
  [0.56732059 0.84283408 0.62421889 0.31080542]]

 [[0.95731648 0.42606245 0.70647531 0.41905399]
  [0.6233089  0.66256545 0.94973562 0.35410947]
  [0.15091625 0.82618572 0.25477008 0.21681509]]]
Random normal:
 [[[ 0.9856035  -0.67212661  0.15864263  0.26373472]
  [-0.86768396  0.1821877   0.95726832 -0.81939135]
  [ 1.02944177 -1.60498661 -0.23760953 -1.70455597]]

 [[-0.93891731  0.15175773 -1.82658148 -0.30051663]
  [-0.39320658 -1.3229441   1.30537824  0.79100702]
  [-1.60754607 -0.25439804 -0.01395379 -0.42215926]]]
Random integers:
 [[[8 0 8 0]
  [1 9 8 7]
  [5 6 9 0]]

 [[9 1 0 9]
  [3 5 9 9]
  [8 0 1 7]]]
```

Using `np.r_`

Similarly to previous cases, you can use `np.r_` to create 3D arrays.

```python
array_3d = np.r_[
    "0,3",
    np.arange(12).reshape(3, 4),
    np.arange(12, 24).reshape(3, 4),
]
print("3D Array:\n", array_3d)
```

```
3D Array:
 [[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

Data type

By default, Numpy automatically determines the "best" **data type** for a given array, based on the provided values. E.g., if all the values are integers, it will be an integer array. If any of the values is a float, it will be a float array. Strings and chars are also supported. You can also create mixed arrays, in which case the data type will be `object` (though, this is not recommended).

Often, however, you will need to specify the data type explicitly. There are three main reasons for this:

- You want to avoid automatic type conversion.
- You want to control the memory usage (64-bit ints and floats are used by default).
- Specific data type is required for your specific application (e.g., if you need to store binary or image data).

Automatic data type conversion might happen, for example, when you append a numeric value to a string array.

```
string_array = np.array("a b c d".split() + [5.0])
string_array = np.concat((string_array, np.r_[7]))
print("String array:", string_array)
print("Data type:", string_array.dtype)
```

```
String array: ['a' 'b' 'c' 'd' '5.0' '7']
Data type: <U32
```

Notice that both `5.0` and `7` where converted to strings. `U32` means fixed-length unicode string of 32 characters - that is 32x4 bytes for each item. Numpy arrays store string data in strings of equal length - the length is determined and expanded automatically based on the maximum length of the string in the array.

The memory issue comes into place when working with very large arrays, for example, images or large data sets.

First, create a helper function to print the data type and memory usage:

```
import sys

def print_memory_usage(array):
    print(
        f"Memory usage for data type '{array.dtype}':",
        f"{sys.getsizeof(array) / 1024**2:0.3f} MB"
    )
```

The `sys.getsizeof` function returns the size in bytes that the object takes up in the memory.

Let's create a matrix with *three million* elements (1000x1000x3) of byte values (0-255, e.g., typical image).

```
import sys

integer_array = np.random.randint(0, 256, (1000, 1000, 3))
print_memory_usage(integer_array)
# convert the integer array to bytes
byte_array = integer_array.astype(np.uint8)
print_memory_usage(byte_array)
print(np.allclose(integer_array, byte_array))  # make sure there is now data loss
```

```
Memory usage for data type 'int64': 22.888 MB
Memory usage for data type 'uint8': 2.861 MB
True
```

The default string data type - unicode - is also less memory efficient - 4 bytes are needed for each character. It might be useful to change if you need to store large text data with only ASCII characters.

```
import string  # import the string module and get the list of ASCII letters
letters = np.array(list(string.ascii_letters))

random_text = np.random.choice(letters, int(1e6))  # 1 million characters
print_memory_usage(random_text)
random_bytes = random_text.astype(np.bytes_)  # 1-byte characters
print_memory_usage(random_bytes)
random_bytes = random_text.astype('U32')  # Unicode, up to 32 chars
print_memory_usage(random_bytes)
```

```
Memory usage for data type '<U1': 3.815 MB
Memory usage for data type '|S1': 0.954 MB
Memory usage for data type '<U32': 122.070 MB
```

Setting the data type

You can set data type manually using the `dtype` parameter, when creating the array.

```
short_int_array = np.array([1, 2, 3], dtype=np.int16)
print_memory_usage(short_int_array)
```

```
Memory usage for data type 'int16': 0.000 MB
```

Some other Numpy functions also support the `dtype` parameter. For example, `np.random.randint`:

```
random_byte_array = np.random.randint(0, 256, (1000, 1000, 3), dtype=np.uint8)
print_memory_usage(random_byte_array)
```

```
Memory usage for data type 'uint8': 2.861 MB
```

Converting data types

Existing arrays can also be converted to different data types with the `astype` array method. For example:

```
int_array = np.array([1, 2, 3])
float_array = int_array.astype(np.float32)
print("Integer array:", int_array)
print("Float array:", float_array)
```

```
Integer array: [1 2 3]
Float array: [1. 2. 3.]
```

Defining the data type

You have already seen a few data types. Basic data types are simply provided by their Python *type* name (e.g., `int`, `float`, `str`, `bool`). For more specific types, such as integer with specific number of bits, you can use the types defined by Numpy (e.g., `np.int32`, `np.float64`). You can also specify a data type using `np.dtype` function. It takes the data type code as an argument (e.g., `np.dtype('int32')`). You have seen some of these codes when we were printing the size of arrays of different types.

For reference, here is a list of **base codes**:

Character Code:
- `i`: Signed integer.
- `u`: Unsigned integer.
- `f`: Float.
- `c`: Complex.

- `b`: Boolean.
- `S`: Bytes (ASCII string).
- `U`: Unicode string.
- `M`: Datetime.
- `m`: Timedelta.
- `O`: Python object.
- `V`: Void (raw data).

Of these, you will most likely only use 'i', 'u', 'f' and maybe 'U', which can be defined directly: `np.int<bits>`, `np.uint<bits>`, `np.float<bits>`, `np.str_` (substitute `<bits>` with the number of bits used by the specific data type).

Note that for *direct* type definition (using the single letter code) you don't specify the number of *bits*, but **bytes**. For example, `np.int32` is "i4" (32 bits is 4 bytes) and `np.float64` is "f8" (64 bits is 8 bytes).

Nonetheless, it is useful to know the other types in case your array "get converted to" one of those and you will see that code when you inspect the `dtype` attribute of the array.

For completeness, the code might also include `<`, `>`, `|`. This indicates the bit ordering (little or big endian).

Indexing

Indexing works in similar way to the Python lists, i.e., the "usual" indexing and slicing. The difference is that for multi-dimensional lists the indexing is:

```
list_2d[row][column]
```

while for multi-dimensional Numpy arrays the indexing is:

```
array_2d[row, column]
```

That is, packed into single square brackets.

Simple indexing

```
a = np.array([1, 2, 3, 4, 5])
print("Element at index 0:", a[0])
print("Element at index -1:", a[-1])
```

```
b = np.array([[1, 2, 3], [4, 5, 6]])
print("Element at row 1, col 2:", b[1, 2])
print("Element at row -1, col 0:", b[-1, 0])
print("Element at row -1 (last row), col -2 (second to last):", b[-1, -2])

c = np.random.rand(3, 4, 5)
print("Element at row 1, col 2, depth 3:", c[1, 2, 3])
print("Element at row 0, col 0, depth 0:", c[0, 0, 0])
print("Element at row -1, col -1, depth -1:", c[-1, -1, -1])
```

```
Element at index 0: 1
Element at index -1: 5
Element at row 1, col 2: 6
Element at row -1, col 0: 4
Element at row -1 (last row), col -2 (second to last): 5
Element at row 1, col 2, depth 3: 0.4565380719574368
Element at row 0, col 0, depth 0: 0.33203702519902345
Element at row -1, col -1, depth -1: 0.7325154059513793
```

Slicing

One bid advantage of Numpy arrays is that you can create multi-dimensional slices of multi-dimensional arrays. For example:

```
m = np.random.rand(3, 4, 5)
print("First row of m:\n", m[0, ...])  # same as m[0, :, :] or m[0]
print("First column of m:\n", m[:, 0])  # same as m[:, 0, :]
print("First depth slice of m:\n", m[..., 0])  # same as m[:, :, 0]
```

```
First row of m:
 [[0.5879116  0.2703893  0.49131424 0.09045312 0.70979462]
 [0.90112622 0.88989293 0.93462929 0.74921512 0.90138669]
 [0.56429463 0.29098177 0.72185279 0.8826334  0.46783102]
 [0.91768049 0.82745265 0.70549574 0.41438552 0.12089209]]
First column of m:
 [[0.5879116  0.2703893  0.49131424 0.09045312 0.70979462]
 [0.00568129 0.11017769 0.47870619 0.60162688 0.84491397]
 [0.9017782  0.2581337  0.34177009 0.77566051 0.80572765]]
First depth slice of m:
 [[0.5879116  0.90112622 0.56429463 0.91768049]
```

```
 [0.00568129 0.70814726 0.52981191 0.66722104]
 [0.9017782  0.72802722 0.16871078 0.7888741 ]]
```

All of these are 2D matrices - slices from a 3D array.

The following will create a 3D slice of the matrix `m`:

```
print(m[:, :2, -2:])
```

```
[[[0.09045312 0.70979462]
  [0.74921512 0.90138669]]

 [[0.60162688 0.84491397]
  [0.75941928 0.36913458]]

 [[0.77566051 0.80572765]
  [0.45462676 0.42988106]]]
```

In this case, we took all of the rows (indicated by `:`), the first two columns (`:2`), and the last two depth slices (`-2:`).

If this seems a little confusing - each element in the brackets (separated by comma), specifies slice for the specific dimension. Otherwise, the slicing principle is the same as for Python lists (i.e., [start, stop, step], with each argument being optional).

The only small difference is that you can provide colon `:` for each dimension, which will indicate "use all of this dimension". E.g., `a[:, 2]` means "from each row, take the element from the column 2" (third column, since its 0-based index). Or in other words, create a slice of the whole second column. For more dimensional arrays, you can also use ellipsis `...`, which will count as putting colon `:` for each of the corresponding dimensions. For example, for a 3D array, you can either specify `a[:, :, 3]`, which means "take the third depth slice (last dimension) for each row and column (first two dimensions)". An equivalent form would be `a[..., 3]`, which does exactly the same.

Lastly, omitting dimension specification at the end is the same as using colon or ellipsis for them. E.g., for a 3D array, this `a[0, :, :]` will take each column and depth slice for the first row of the 3D array (thus, creating a 2D sub-matrix located at row 0 in the original matrix). This is the same as `a[0, ...]` or `a[0]`.

Assignment

The assignment works in the same way as in Python lists. However, you can directly assign to multi-dimensional slices of multi-dimensional arrays. What's more, you can either assign scalars (each value of the slice will be set to this value) or multi-dimensional arrays (of the same dimension as the slice).

Scalar assignment to a slice along a single dimension:

```
a = np.arange(24).reshape(2, 3, 4)
print(a, end="\n---\n")
a[0, ...] = 1  # same as a[0, :, :] or a[0]
print(a, end="\n---\n")
a[:, 0] = 2  # same as a[:, 0, :]
print(a, end="\n---\n")
a[..., 0] = 3  # same as a[:, :, 0]
print(a, end="\n---\n")
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
---
[[[ 1  1  1  1]
  [ 1  1  1  1]
  [ 1  1  1  1]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
---
[[[ 2  2  2  2]
  [ 1  1  1  1]
  [ 1  1  1  1]]

 [[ 2  2  2  2]
  [16 17 18 19]
  [20 21 22 23]]]
---
```

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

 [[ 3  2  2  2]
  [ 3 17 18 19]
  [ 3 21 22 23]]]
---
```

Scalar assignment to a 2D slice:

```
a = np.arange(24).reshape(2, 3, 4)
a[:, :2, -2:] = 4
print(a)
```

```
[[[ 0  1  4  4]
  [ 4  5  4  4]
  [ 8  9 10 11]]

 [[12 13  4  4]
  [16 17  4  4]
  [20 21 22 23]]]
```

Assigning a 2D array to a 2D slice:

```
a = np.arange(24).reshape(2, 3, 4).astype(np.float16)
b = np.random.rand(2, 2)
print("b:\n", b, end="\n---\n")
a[0, :2, -2:] = b
print("a (modif):\n", a)
```

```
b:
 [[0.22793463 0.56677131]
 [0.40437459 0.23692991]]
---
a (modif):
 [[[ 0.      1.      0.2279  0.567 ]
  [ 4.      5.      0.4043  0.2369]
  [ 8.      9.     10.     11.    ]]

 [[12.     13.     14.     15.    ]
```

```
  [16.     17.     18.     19.    ]
  [20.     21.     22.     23.    ]]]
```

Assigning a 3D array to a 3D slice:

```
a = np.arange(24).reshape(2, 3, 4).astype(np.float16)
b = np.random.rand(2, 2, 2)
print("b:\n", b, end="\n---\n")
a[:, :2, -2:] = b
print("a (modif):\n", a)
```

```
b:
 [[[0.17033391 0.55406923]
  [0.6179509  0.08872283]]

 [[0.74399683 0.38245909]
  [0.68451162 0.51904386]]]
---
a (modif):
 [[[ 0.       1.       0.1703   0.554  ]
  [ 4.       5.       0.618    0.08875]
  [ 8.       9.      10.      11.     ]]

 [[12.      13.       0.744    0.3826 ]
  [16.      17.       0.6846   0.519  ]
  [20.      21.      22.      23.     ]]]
```

Assigning a 1D array to a 3D slice:

```
a = np.arange(24).reshape(2, 3, 4).astype(np.float16)
b = np.random.rand(2)
print("b:\n", b, end="\n---\n")
a[:, :2, -2:] = b
print("a (modif):\n", a)
```

```
b:
 [0.1757673 0.7144757]
---
a (modif):
 [[[ 0.      1.      0.1758  0.7144]
  [ 4.      5.      0.1758  0.7144]
```

```
  [ 8.      9.     10.     11.    ]]

 [[12.     13.      0.1758  0.7144]
  [16.     17.      0.1758  0.7144]
  [20.     21.     22.     23.    ]]]
```

The cases where we assign a lower-dimensional array to a higher-dimensional slice showcases the **broadcasting** ability of Numpy arrays. See below for more details.

(Re)shaping arrays

Reshape

As you've seen, the `reshape` method allows you to change the shape of the array. There, you simply specify the desired dimensions of the new array. One thing to keep in mind is that by default, this creates a view into the original array, not a copy. You can, however, specify that you want a copy.

```
a = np.arange(24).reshape(2, 3, 4)
print(a)
b = a.reshape(3, 8)
c = a.reshape(3, 8, copy=True)
print(b)
b[0, 0] = 100
print("a:\n", a)
c[1, 1] = 200
print("a:\n", a)
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
[[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]
 [16 17 18 19 20 21 22 23]]
a:
 [[[100   1   2   3]
  [  4   5   6   7]
  [  8   9  10  11]]
```

```
 [[ 12  13  14  15]
  [ 16  17  18  19]
  [ 20  21  22  23]]]
a:
 [[[100   1   2   3]
  [  4   5   6   7]
  [  8   9  10  11]]

 [[ 12  13  14  15]
  [ 16  17  18  19]
  [ 20  21  22  23]]]
```

Note that modifying `b` does change the original array `a`, while modifying `c` does not.

Increasing dimensions

Sometimes, you might need to increase the dimensionality of an array by creating **singleton dimensions** (one-element dimension). This can be done either by `reshape` or **inserting new axis** into the index. Typical usage for this is if we create a 1D vector, e.g., using `arange`, and want to convert it to a 2D array.

For the reshape approach, we simply specify `1` for the new dimension:

```
a = np.arange(6)
b = a.reshape(6, 1)
c = a.reshape(1, 6)
print(a)
print(b)
print(c)
```

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

Notice that `b` is a 2D vector with 6 rows and 1 column and `c` is a 2D vector with 1 row and 6 columns.

You can also supply `-1` for the non-singleton dimension, to make things simpler and more general. And you can create more than 2D arrays.

```
a = np.arange(6)
b = a.reshape(-1, 1, 1)
c = a.reshape(1, 1, -1)
print(a)
print(b)
print(c)
```

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

 [[1]]

 [[2]]

 [[3]]

 [[4]]

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

New axis

Alternatively, you can use `np.newaxis` with indexing trick. The following will yield the same result as using `reshape`:

```
a = np.arange(6)
b = a[np.newaxis, :]
c = a[:, np.newaxis]
print(a)
print(b)
print(c)
```

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

```
 [3]
 [4]
 [5]]
```

Squeezing out singleton dimensions

Sometimes, you need to do the inverse, i.e., remove singleton dimensions, Numpy offers the `squeeze` method:

```
a = np.arange(6)
b = a[np.newaxis, :]
c = a[:, np.newaxis]
print("b")
print(b, b.shape)
print("b squeezed")
print(np.squeeze(b), np.squeeze(b).shape)
print("c")
print(c, c.shape)
print("c squeezed")
print(np.squeeze(c), np.squeeze(c).shape)
```

```
b
[[0 1 2 3 4 5]] (1, 6)
b squeezed
[0 1 2 3 4 5] (6,)
c
[[0]
 [1]
 [2]
 [3]
 [4]
 [5]] (6, 1)
c squeezed
[0 1 2 3 4 5] (6,)
```

Iteration over arrays

Even though vectorized operations are preferred (see next section), sometimes you may want to use a loop with arrays (e.g., for debugging or custom operations). Like the Python lists, you can use `for` loop to iterate over the indices or the **for each** *version*. In the later case, the iteration will happen along the first dimension of the array (row).

Iterating over 1D arrays

For 1D arrays, the iteration is fairly simple:

```
a = np.array([10, 20, 30])
for element in a:  # for each
    print("Element:", element)
# OR
for i in range(len(a)):
    print(f"Element {i}:", a[i])
```

```
Element: 10
Element: 20
Element: 30
Element 0: 10
Element 1: 20
Element 2: 30
```

Iterating over 2D arrays

For higher dimensional arrays, you can iterate over the rows. For example:

```
a = np.arange(12).reshape(3, 4)
for row in a:  # for each row
    print(row)
```

```
[0 1 2 3]
[4 5 6 7]
[ 8  9 10 11]
```

Subsequent iteration will iterate over the first dimension of the "slice":

```
for row in a:
    for element in row:
        print(element, end=", ")
    print("", end=chr(8)*2 + "; ")  # chr(8) == backspace
```

```
0, 1, 2, 3, ; 4, 5, 6, 7, ; 8, 9, 10, 11, ;
```

Iterating over 3D arrays

```
a = np.arange(24).reshape(2, 3, 4)
for depth in a:
    for row in depth:
        for element in row:
            print(element, end=", ")
        print("", end=chr(8)*2 + "; ")
    print("", end=chr(8) + " -- ")
```

```
0, 1, 2, 3, ; 4, 5, 6, 7, ; 8, 9, 10, 11, ;  -- 12, 13, 14, 15, ; 16, 17, 18, 19, ; 20, 21, 2
```

Linear iteration over multi-dimensional arrays

You can directly iterate over individual elements of a multi-dimensional array (called linear iteration). You can either **ravel** the array or use the `nditer` iterator.

The `ravel` method will simply "stretch" or flatten the array into a 1D array, row-by-row. It has (almost) the same effect as `reshape(-1)`, although, it is the preferred method. Both of these two methods will return a flattened **view** of the array (not a copy). The difference is that `ravel` returns a **contiguous** view, if possible. This means, certain operations will be faster. There is also a `flatten` method, which also returns a 1D array, but it returns a **copy** of the original array instead of a view.

```
M = np.arange(24).reshape(2, 3, 4)
print("Original array:\n", M)
print("Flattened (raveled) array:\n", M.ravel())
```

```
Original array:
 [[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
Flattened (raveled) array:
 [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
```

Try uncommenting any one of the "flattening" lines to see the difference:

```
M = np.arange(24).reshape(2, 3, 4)

flat_array = M.ravel()
# flat_array = M.reshape(-1)
# flat_array = M.flatten()

for i, x in enumerate(flat_array):
    flat_array[i] = x * 2
print("Original array:\n", M)
print("Modified flattened array:\n", flat_array)
```

```
Original array:
 [[[ 0  2  4  6]
  [ 8 10 12 14]
  [16 18 20 22]]

 [[24 26 28 30]
  [32 34 36 38]
  [40 42 44 46]]]
Modified flattened array:
 [ 0  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46]
```

Linear iteration with `np.nditer`. This function returns a special **iterator** object that can be used by the for loop. It is actually a very powerful method that allows all sorts of special iterations but that is beyond the scope of this course. You can read more about it here.

```
M = np.arange(24).reshape(2, 3, 4)
for x in np.nditer(M):
    print(x, end=" ")
print()
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
```

Arithmetic operations and vectorization

A big advantage of Numpy arrays is that you can perform arithmetic operations on them directly. Unlike Python lists, for which we had to implement functions, iterating over individual elements, you can compute certain operations directly on the array.

Here, we go through some basic functions. A more complete list of mathematical functions that Numpy provides can be found in its documentation.

Basic arithmetic operations

```
A = np.arange(12).reshape(3, 4)
B = A * 2 + 3
print("A multiplied by 2, then added 3:\n", B)
print("A^2", A**2)
```

```
A multiplied by 2, then added 3:
 [[ 3  5  7  9]
 [11 13 15 17]
 [19 21 23 25]]
A^2 [[  0   1   4   9]
 [ 16  25  36  49]
 [ 64  81 100 121]]
```

Mathematical functions

The Numpy library also specifies a handful of mathematical operations:

```
print(f"square root of A:\n{np.sqrt(A)}")
print(f"exponential of A:\n{np.exp(A)}")
print(f"logarithm of A:\n{np.log(A + 1e-9)}")
print(f"sine of A:\n{np.sin(A)}")
print(f"cosine of A:\n{np.cos(A)}")
```

```
square root of A:
[[0.         1.         1.41421356 1.73205081]
 [2.         2.23606798 2.44948974 2.64575131]
 [2.82842712 3.         3.16227766 3.31662479]]
exponential of A:
[[1.00000000e+00 2.71828183e+00 7.38905610e+00 2.00855369e+01]
 [5.45981500e+01 1.48413159e+02 4.03428793e+02 1.09663316e+03]
 [2.98095799e+03 8.10308393e+03 2.20264658e+04 5.98741417e+04]]
logarithm of A:
[[-2.07232658e+01  1.00000008e-09  6.93147181e-01  1.09861229e+00]
 [ 1.38629436e+00  1.60943791e+00  1.79175947e+00  1.94591015e+00]
 [ 2.07944154e+00  2.19722458e+00  2.30258509e+00  2.39789527e+00]]
sine of A:
[[ 0.          0.84147098  0.90929743  0.14112001]
 [-0.7568025  -0.95892427 -0.2794155   0.6569866 ]
```

```
 [ 0.98935825  0.41211849 -0.54402111 -0.99999021]]
cosine of A:
[[ 1.          0.54030231 -0.41614684 -0.9899925 ]
 [-0.65364362  0.28366219  0.96017029  0.75390225]
 [-0.14550003 -0.91113026 -0.83907153  0.0044257 ]]
```

Aggregation functions

```
A = np.arange(12).reshape(3, 4)
print("A:\n", A)
print()
print(f"sum of A:\n{np.sum(A)}")
print(f"mean of A:\n{np.mean(A)}")
print(f"minimum of A:\n{np.min(A)}")
print(f"maximum of A:\n{np.max(A)}")
print(f"standard deviation of A:\n{np.std(A)}")
print(f"variance of A:\n{np.var(A)}")
print(f"median of A:\n{np.median(A)}")
print(f"cumulative sum of A:\n{np.cumsum(A)}")
print(f"cumulative product of A:\n{np.cumprod(A)}")
print(f"norm of A:\n{np.linalg.norm(A)}")
```

```
A:
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

sum of A:
66
mean of A:
5.5
minimum of A:
0
maximum of A:
11
standard deviation of A:
3.452052529534663
variance of A:
11.916666666666666
median of A:
```

```
5.5
cumulative sum of A:
[ 0  1  3  6 10 15 21 28 36 45 55 66]
cumulative product of A:
[0 0 0 0 0 0 0 0 0 0 0 0]
norm of A:
22.494443758403985
```

For the aggregation functions, the dimension along which the aggregation should be done, can be specified. Some of the aggregation functions are also array **methods**. For example:

```
print(f"sum of A along axis 0 (row-wise):\n{A.sum(axis=0)}")
print(f"sum of A along axis 1 (column-wise):\n{A.sum(axis=1)}")
print(f"mean of A along axis 0:\n{A.mean(axis=0)}")
print(f"mean of A along axis 1:\n{A.mean(axis=1)}")
print(f"norm of A along axis 0:\n{np.linalg.norm(A, axis=0)}")
print(f"norm of A along axis 1:\n{np.linalg.norm(A, axis=1)}")
```

```
sum of A along axis 0 (row-wise):
[12 15 18 21]
sum of A along axis 1 (column-wise):
[ 6 22 38]
mean of A along axis 0:
[4. 5. 6. 7.]
mean of A along axis 1:
[1.5 5.5 9.5]
norm of A along axis 0:
[ 8.94427191 10.34408043 11.83215957 13.37908816]
norm of A along axis 1:
[ 3.74165739 11.22497216 19.13112647]
```

Operations between arrays

Mathematical operations between arrays

Much like arithmetics with scalars, many operations can be performed between arrays.

All of the following operations are element-wise:

```
a = np.arange(12).reshape(3, 4)
b = np.arange(12, 24).reshape(3, 4)
print("a + b = \n", a + b)
print("a - b = \n", a - b)
print("a * b = \n", a * b)
print("a / b = \n", a / b)
print("a ** b = \n", a ** b)
```

```
a + b =
 [[12 14 16 18]
 [20 22 24 26]
 [28 30 32 34]]
a - b =
 [[-12 -12 -12 -12]
 [-12 -12 -12 -12]
 [-12 -12 -12 -12]]
a * b =
 [[  0  13  28  45]
 [ 64  85 108 133]
 [160 189 220 253]]
a / b =
 [[0.         0.07692308 0.14285714 0.2       ]
 [0.25       0.29411765 0.33333333 0.36842105]
 [0.4        0.42857143 0.45454545 0.47826087]]
a ** b =
 [[                   0                    1                16384
             14348907]
 [          4294967296         762939453125      101559956668416
     11398895185373143]
 [ 1152921504606846976 -1261475310744950487  1864712049423024128
   6839173302027254275]]
```

There are also matrix versions of sum of the operations, for example, the matrix dot product (for vectors, it will be the inner product):

```
a = np.arange(12).reshape(3, 4)
b = np.arange(12, 24).reshape(4, 3)
print("a . b = \n", a.dot(b))
```

```
a . b =
 [[114 120 126]
```

```
 [378 400 422]
 [642 680 718]]
```

Alternatively, as mentioned before, for 2D arrays, you can convert them directly into the `matrix` class and all operations on them will be matrix operations (where applicable).

Broadcasting

Another big advantage of Numpy is the **broadcasting** ability. Broadcasting means expanding a lower-dimensional array to allow arithmetic operation with higher-dimensional arrays. In the simplest case, you can see this when multiplying a scalar with a 1 or higher dimensional array. The scalar is "repeated" over all elements of the array, and the result is a new array with the same shape as the original array.

It is possible to do this with even higher dimensional inputs, e.g., 1D vs 2D or 2D vs 3D arrays. This ability have, of course, certain limitations. It must be possible to do the broadcasting in a meaningful, unambiguous way. For example, you can multiply a column vector of shape [1, M] with a matrix of shape [N, M]. In this case, the vector is "copied" over each row. However, it would not make sense to do the same with a vector of shape [1, K], where `K != M`

1D vs 2D broadcasting

```
N, M = 3, 4
a = np.arange(N * M).reshape(N, M)
b = np.arange(M).reshape(1, M)
c = np.arange(N).reshape(N, 1)
d = np.arange(M)  # this also works but has limited use
print(a.shape, b.shape)
print(a * b)
print(a.shape, c.shape)
print(a * c)
print(a.shape, d.shape)
print(d * a)  # this is always row-wise
```

```
(3, 4) (1, 4)
[[ 0  1  4  9]
 [ 0  5 12 21]
 [ 0  9 20 33]]
(3, 4) (3, 1)
[[ 0  0  0  0]
 [ 4  5  6  7]
 [16 18 20 22]]
(3, 4) (4,)
```

```
[[ 0  1  4  9]
 [ 0  5 12 21]
 [ 0  9 20 33]]
```

2D vs 3D broadcasting

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

```
(2, 2, 3) (2, 3)
[[[ 2  4  6]
  [ 8 10 12]]

 [[ 8 10 12]
  [14 16 18]]]
```

Array filtering / masking (boolean indexing)

Numpy also offers **boolean indexing**. This can be created, for example, by comparing (or applying any other operation with a boolean result) arrays with a scalar or another array.

```
A = np.arange(12).reshape(3, 4)
print("A:\n", A)
print()
# creating boolean index
print("A > 5:\n", A > 5)
# using boolean index to select values from the array
print("A[A > 5]:\n", A[A > 5])
print("A[A % 2 == 0]:\n", A[A % 2 == 0])
print("A[A % 2 == 1]:\n", A[A % 2 == 1])
```

```
A:
 [[ 0  1  2  3]
 [ 4  5  6  7]
```

```
 [ 8  9 10 11]]

A > 5:
 [[False False False False]
 [False False  True  True]
 [ True  True  True  True]]
A[A > 5]:
 [ 6  7  8  9 10 11]
A[A % 2 == 0]:
 [ 0  2  4  6  8 10]
A[A % 2 == 1]:
 [ 1  3  5  7  9 11]
```

`where` function

As an alternative, useful in more complex situations, there is the `where` function. It can be used to create a "flat" index from a boolean condition. It can also be used to create a new array directly.

In the later case, the `where` function is called with the following arguments:

```
np.where(<condition>, <true_values>, <false_values>)
```

If the condition is `True`, the `true_values` will be added to the new array. Otherwise, the `false_values` will be added to the new array.

```
A = np.arange(12).reshape(3, 4)
print("A:\n", A)
print()
print("A where A > 5:\n", np.where(A > 5))
print("A where A > 5:\n", np.where(A > 5, A, 0))
print("A where A % 2 == 0:\n", np.where(A % 2 == 0, A, 0))
print("A where A % 2 == 1:\n", np.where(A % 2 == 1, A, 0))
```

```
A:
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

A where A > 5:
 (array([1, 1, 2, 2, 2, 2]), array([2, 3, 0, 1, 2, 3]))
```

```
A where A > 5:
 [[ 0  0  0  0]
 [ 0  0  6  7]
 [ 8  9 10 11]]
A where A % 2 == 0:
 [[ 0  0  2  0]
 [ 4  0  6  0]
 [ 8  0 10  0]]
A where A % 2 == 1:
 [[ 0  1  0  3]
 [ 0  5  0  7]
 [ 0  9  0 11]]
```

Array concatenation

There are a few functions to combine existing arrays.

Concatenating arrays

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
c = np.concatenate((a, b))
print("Concatenated:\n", c)
d = np.concatenate((a.reshape(-1, 1), b.reshape(-1, 1)), axis=0)
print("Concatenated:\n", d)
e = np.concatenate((a.reshape(-1, 1), b.reshape(-1, 1)), axis=1)
print("Concatenated:\n", e)
```

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

stacking Arrays Vertically and Horizontally

```
A = np.array([[1, 2],
              [3, 4]])
B = np.array([[5, 6]])
v_stack = np.vstack((A, B))    # Add B as new row(s)
h_stack = np.hstack((A, np.array([[5], [6]])) )  # Add B as new column
print("Vertical stack:\n", v_stack)
print("Horizontal stack:\n", h_stack)
```

```
Vertical stack:
 [[1 2]
 [3 4]
 [5 6]]
Horizontal stack:
 [[1 2 5]
 [3 4 6]]
```

There is also a `dstack` function which stacks arrays in the third dimension.

Splitting and stacking arrays

You can also split arrays

```
arr = np.arange(12)
print(arr)
print()
# Split into 3 equal parts
splits = np.array_split(arr, 3)
print(splits)
print(np.stack(splits))
print()
for i, s in enumerate(splits):
    print(f"Split {i}:", s)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]

[array([0, 1, 2, 3]), array([4, 5, 6, 7]), array([ 8,  9, 10, 11])]
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

```
Split 0: [0 1 2 3]
Split 1: [4 5 6 7]
Split 2: [ 8  9 10 11]
```

Data saving and loading

NumPy offers several methods for persisting arrays to disk and reading them back.

Saving and Loading in Binary Format (.npy/.npz)

Using np.save and np.load

```
data = np.array([[1, 2, 3], [4, 5, 6]])
np.save("data.npy", data)  # Saves in binary format
loaded_data = np.load("data.npy")
print("Loaded .npy data:\n", loaded_data)
```

```
Loaded .npy data:
 [[1 2 3]
 [4 5 6]]
```

Saving Multiple Arrays with np.savez:

```
x = np.arange(10)
y = np.linspace(0, 1, 10)
np.savez("multiple.npz", x=x, y=y)
# Load data from .npz file:
data_archive = np.load("multiple.npz")
print("x:", data_archive['x'])
print("y:", data_archive['y'])
```

```
x: [0 1 2 3 4 5 6 7 8 9]
y: [0.         0.11111111 0.22222222 0.33333333 0.44444444 0.55555556
 0.66666667 0.77777778 0.88888889 1.        ]
```

Saving and Loading as Text Files

Using np.savetxt and np.loadtxt:

```
# Save a 2D array to a CSV file.
np.savetxt("data.csv", data, delimiter=",", fmt="%d")
# Load the data back:
loaded_csv = np.loadtxt("data.csv", delimiter=",", dtype=int)
print("Loaded CSV data:\n", loaded_csv)
```

```
Loaded CSV data:
 [[1 2 3]
 [4 5 6]]
```