Unit 6: Advanced NumPy

Richard Foltyn

November 14, 2020

Contents

1 Unit 6: Advanced NumPy **1**
- 1.1 Why NumPy arrays? . . . 1
- 1.2 Creating arrays . . . 2
- 1.3 Array shape . . . 3
- 1.4 Advanced indexing . . . 5
- 1.5 Numerical operations . . . 7
- 1.6 NumPy data types [advanced] . . . 11
- 1.7 Copies and views [advanced] . . . 13
- 1.8 Array storage order [advanced] . . . 15

2 Exercises **16**

3 Solutions **16**

1 Unit 6: Advanced NumPy

We already encountered NumPy arrays and their basic usage throughout this tutorial. In this unit, we will take a more in-depth look at NumPy arrays.

1.1 Why NumPy arrays?

Why don't we just stick with built-in types such as Python lists to store and process data? It turns out that while the built-in objects are quite flexible, this flexibility comes at the cost of decreased performance:

- `list` objects can store arbitrary data types, and the data type of any item can change:

  ```
  items = ['foo']
  items[0] = 1.0      # item was a string, now it's a float!
  ```

- There is no guarantee where in memory the data will be stored. In fact, two consecutive items could be very "far" from each other in memory, which imposes a performance penalty.
- Even primitive data types such as `int` and `float` are not "raw" data, but full-fledged objects. That, again, is bad for performance.

On the other hand, the approach taken by NumPy is to store and process data in a way very similar to low-level languages such as *C* and *Fortran*. This means that

- Arrays contain a *homogenous* data type. *All* elements are either 64-bit integers (`np.int64`), 64-bit floating-point numbers (`np.float64`), or some other of the many data types supported by NumPy.

  It is technically possible to get around this by specifying an array's data type (`dtype`) to be `object`, which is the most generic Python data type. However, we would never want to do this for numerical computations.

- NumPy arrays are usually *contiguous* in memory. This means that adjacent array elements are actually guaranteed to be next to each other in memory, which allows for much more efficient computations.
- NumPy array support numerous operations used in scientific computing. For example, with a NumPy array we can write

  ```
  x = np.array([1, 2, 3])
  y = x + 1       # We would expect this to work
  ```

  With lists, however, we cannot:

  ```
  x = [1, 2, 3]
  y = x + 1       # Does not work!
  ```

  Lists don't implement an addition operator that accepts integer arguments, so this code triggers an error.

You can see performance uplift provided by NumPy arrays in this simple example:

[1]:
```
# Create list 0, 1, 2, ..., 999
lst = list(i for i in range(1000))

# Compute squares, time how long it takes
%timeit [i**2 for i in lst]
```

```
201 µs ± 1.51 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

[2]:
```
# Repeat using NumPy arrays
import numpy as np
arr = np.arange(1000)

%timeit arr**2
```

```
1.07 µs ± 21.2 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

- On my machine (not the fastest laptop), squaring 1000 elements of a `list` takes approximately 200 microseconds, while the equivalent array-based operation takes only abount 1 microsecond. NumPy is therefore approximately 200 times faster!
- Also, as mentioned above, NumPy supports squaring an array directly, while we have to manually loop through the `list` and square each element individually.

Note: `%timeit` is a so-called magic command that only works in notebooks, but not in regular Python files. [See documentation]

1.2 Creating arrays

We have already encountered some of the most frequently used array creation routines:

- `np.array()` creates an array from a given argument, which can be
  - a scalar
  - a collection such as a list or tuple
  - some other iterable object, eg. something created by `range()`
- `np.empty()` allocates memory for a given array shape, but does not overwrite it with initial values.
- `np.zeros()` creates an array of a given shape and initializes it to zeros.
- `np.ones()` creates an array of a given shape and initializes it to ones.
- `np.arange(start, stop, step)` creates an array with evenly spaced elements over the range $[start, stop)$.
  - `start` and `step` can be omitted and then default to `start=0` and `step=1`.
  - Note that `stop` is not included!

- `np.linspace(start, stop, num)` returns a vector of `num` elements which are evenly spaced over the interval $[start, stop]$.
- `np.identity(n)` returns the identity matrix of a size $n \times n$.
- `np.eye()` is a more flexible variant of`identity()` that can, for example, also create non-squared matrices.

There are many more array creation functions for more exotic use-cases, see the NumPy documentation for details.

Examples:

[3]:
```
import numpy as np

# Create array from list
lst = [1, 2, 3]
np.array(lst)
```

[3]:
```
array([1, 2, 3])
```

[4]:
```
# Create array from tuple
tpl = 1.0, 2.0, 3.0
np.array(tpl)
```

[4]:
```
array([1., 2., 3.])
```

[5]:
```
# arange: end point is not included!
np.arange(5)
```

[5]:
```
array([0, 1, 2, 3, 4])
```

[6]:
```
# arange: increments can be negative too!
np.arange(5, 1, -1)
```

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

[7]:
```
# arange also works on floats
np.arange(1.0, 3.0, 0.5678)
```

[7]:
```
array([1.    , 1.5678, 2.1356, 2.7034])
```

[8]:
```
# linspace DOES include the end point
np.linspace(0.0, 1.0, 11)
```

[8]:
```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

1.3 Array shape

Many of the array creation take the desired shape of the array as their first argument. Array shapes are usually specified as tuples:

- A vector with 5 elements has shape `(5, )`.

  Note the comma `,` : we need to specify a tuple with a single element using this comma, since `(5)` is just the integer 5, not a tuple.

- A 2×2 matrix has shape `(2, 2)`.
- A higher-dimensional array has shape `(k, l, m, n, ...)`.
- A *scalar* NumPy array has shape `()`, an empty tuple.

  While "scalar array" sounds like an oxymoron, it does exist.

We can query the shape of an array using the `shape` attribute, and the number of dimensions is stored in the `ndim` attribute.

Examples:

[9]:
```
import numpy as np

# Scalar array
x = np.array(0.0)
print('Scalar array with shape={} and ndim={}'.format(x.shape, x.ndim))
```

```
Scalar array with shape=() and ndim=0
```

Note that the a scalar NumPy array is not the same as a Python scalar. The built-in type `float` has neither a `shape`, nor an `ndim`, nor any other of the NumPy array attributes.

[10]:
```
scalar = 1.0
scalar.shape
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-1-cebbc052f34b> in <module>
      1 scalar = 1.0
----> 2 scalar.shape

AttributeError: 'float' object has no attribute 'shape'
```

[11]:
```
# 1-dimensional array (vector), values not initialised
x = np.empty((5,))
x       # could contain arbitrary garbage
```

[11]:
```
array([0.0e+000, 4.9e-324, 9.9e-324, 1.5e-323, 2.0e-323])
```

An array created with `empty()` will contain arbitrary garbage since the memory block assigned to the array is not initialised. The result will potentially differ on each invocation and across computers.

[12]:
```
# 1-dimensional array
x = np.empty(5)      # equivalent to np.empty((5,))
```

Most function accept an integer value instead of a `tuple` when creating 1-dimensional arrays, which is interpreted as the number of elements.

[13]:
```
# 3d-array
x = np.ones((1, 2, 3))
x
```

[13]:
```
array([[[1., 1., 1.],
        [1., 1., 1.]]])
```

Recall from unit 2 that we can use the `reshape()` method to convert arrays to a different shape:

- The resulting number of elements must remain unchanged!
- *One* dimension can be specified using `-1`, which will prompt NumPy to compute the implied dimension size itself.

[14]:
```
x = np.zeros((2, 1, 3))
x = x.reshape((3, -1))       # Infer number of columns
x
```

[14]:
```
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
```

1.4 Advanced indexing

We previously covered single element indexing and slicing, which works the same way for both Python `list` and `tuple` objects as well as NumPy arrays.

NumPy additionally implements more sophisticated indexing mechanisms which we cover now.

- You might also want to consult the NumPy indexing tutorial and the detailed indexing reference.

1.4.1 Boolean or "mask" indexing

We can pass logical arrays as indices:

- Logical (or boolean) arrays consist of elements that can only take on values `True` and `False`
- We usually don't create logical arrays manually, but apply an operation that results in `True/False` values, such as a comparison.
- The boolean index array usually has the *same* shape as the indexed array.

Examples:

[15]:
```
import numpy as np

vec = np.arange(5)
mask = (vec > 1)          # apply comparison to create boolean array
mask
```

[15]:
```
array([False, False,  True,  True,  True])
```

[16]:
```
vec[mask]                 # use mask to retrieve only elements greater than 1
```

[16]:
```
array([2, 3, 4])
```

We can even apply boolean indexing to multi-dimensional arrays. The result will be flatted to a 1-dimensional array, though.

[17]:
```
mat = np.arange(6).reshape((2,3))
mat
```

[17]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

[18]:
```
mask = (mat > 1)          # create boolean array
mask
```

[18]:
```
array([[False, False,  True],
       [ True,  True,  True]])
```

[19]:
```
mat[mask]              # collapses result to 1-d array
```

[19]:
```
array([2, 3, 4, 5])
```

Logical indexing does **not** work with `tuple` and `list`

[20]:
```
tpl = (1, 2, 3)
mask = (True, False, True)
tpl[mask]                 # error
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-19bb25c629fe> in <module>
```

```
      1 tpl = (1, 2, 3)
      2 mask = (True, False, True)
----> 3 tpl[mask]                # error

TypeError: tuple indices must be integers or slices, not tuple
```

1.4.2 Integer index arrays

We can also use index arrays of *integer* type to select specific elements on each axis. These are straightforward to use for 1-dimensional arrays, but can get fairly complex with multiple dimensions.

[21]:
```
import numpy as np

data = np.arange(10)
index = [1, 2, 9]        # select second, third and 10th element
data[index]
```

[21]:
```
array([1, 2, 9])
```

As you see, the index array does not have to be a NumPy array, but can also be a list (not a tuple, though!).

In general, if we are using an index array to select elements along an axis of length n, the index

- must only contain integers between 0 and $n-1$, or negative integers from $-n$ to -1 (which, as usual, count from the end of the axis).
- can be of arbitrary length. We can therefore select the same element multiple times.

[22]:
```
data = np.arange(5, 10)      # array with 5 elements, [5,...,9]
data
```

[22]:
```
array([5, 6, 7, 8, 9])
```

[23]:
```
index = [0, 1, 1, 2, 2, 3, 3, 4, 4]          # select elements multiple times
data[index]
```

[23]:
```
array([5, 6, 6, 7, 7, 8, 8, 9, 9])
```

With multi-dimensional data arrays, the same restrictions apply, and additionally - if more than one axis is indexed using index arrays, the index arrays have to be of equal length.

[24]:
```
data = np.arange(12).reshape((3, 4))
data
```

[24]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

[25]:
```
index1 = [0, 2]      # row indices
index2 = [1, 3]      # column indices
data[index1, index2]
```

[25]:
```
array([ 1, 11])
```

The code above selects to elements, the first at position `(0,1)`, the second at position `(2,3)`.

We can combine index arrays on one axis with other indexing methods used for another axis:

[26]:
```
data[index1, 2]      # return elements in 3rd column from rows given in index1
```

[26]:
```
array([ 2, 10])
```

Using different indexing methods, in particular index arrays, on higher-dimensional data can quickly become a mess, and you should be extra careful to see if the results make sense.

1.5 Numerical operations

1.5.1 Element-wise operations

Element-wise operations are performed on each element individually and leave the resulting array's shape unchanged.

There are three types of such operations:

1. One operand is an array and one is a scalar.
2. Both operands are arrays, either of identical shape, or broadcastable to an identical shape (we discuss broadcasting below)
3. A function is applied to each array element.

Case 1: Array-scalar operations. These intuitively behave as you would expect:

[27]:
```
import numpy as np

x = np.arange(10)
scalar = 1

# The resulting array y has the same shape as x:
y = x + scalar      # addition
y = x - scalar      # subtraction
y = x * scalar      # multiplication
y = x / scalar      # division
y = x // scalar     # division with integer truncation
y = x % scalar      # modulo operator
y = x ** scalar     # power function
y = x == scalar     # comparison: also >, >=, <=, <
```

Note that unlike in Matlab, the "standard" operators work element-wise, so `x * y` is **not** matrix multiplication!

Case 2: Both operands are arrays of equal shape:

[28]:
```
x = np.arange(10)
y = np.arange(10, 20)       # has same shape as x

# Resulting array z has the same shape as x and y:
z = x + y           # addition
z = x - y           # subtraction
z = x * y           # multiplication
z = x / y           # division
z = x // y          # division with integer truncation
z = x % y           # modulo operator
z = x ** y          # power function
y = x == y          # comparison: also >, >=, <=, <
```

Case 3: Applying element-wise functions This case covers numerous functions defined by NumPy, such as - `np.sqrt`: square root - `np.exp`, `np.log`, `np.log10`: exponential and logarithmic functions - `np.sin`, `np.cos`, etc.: trigonometric functions

You can find a complete list of mathematical functions in the NumPy documentation (not all functions listed there operate element-wise!).

[29]:
```
# element-wise functions
x = np.arange(1, 11)
y = np.exp(x)       # apply exponential function
```

```
y = np.log(x)        # apply natural logarithm
```

1.5.2 Matrix operations

Transpose You can transpose a matrix using the `T` attribute:

[30]:
```
mat = np.arange(6).reshape((2,3))
mat
```

[30]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

[31]:
```
mat.T
```

[31]:
```
array([[0, 3],
       [1, 4],
       [2, 5]])
```

Array multiplication Matrix multiplication is performed using the `np.dot()` function ("dot product"). The operands need not be matrices, but can be vectors as well, or even high-dimensional arrays (the result is then not entirely obvious and one should check the docs).

Every newer version of Python and NumPy additionally interprets `@` as the matrix multiplication operator.

[32]:
```
import numpy as np

mat = np.arange(9).reshape((3, 3))
vec = np.arange(3)

# matrix-matrix multiplication
np.dot(mat, mat)     # or: mat @ mat
```

[32]:
```
array([[ 15,  18,  21],
       [ 42,  54,  66],
       [ 69,  90, 111]])
```

[33]:
```
# vector dot product (returns a scalar)
np.dot(vec, vec)     # or: vec @ vec
```

[33]:
```
5
```

[34]:
```
# matrix-vector product (returns vector)
np.dot(mat, vec)     # or: mat @ vec
```

[34]:
```
array([ 5, 14, 23])
```

We must of course make sure that matrices and vector have conformable dimensions!

[35]:
```
mat = np.arange(6).reshape((2, 3))
mat
```

[35]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

[36]:
```
np.dot(mat, mat)        # raises error, cannot multiply 2x3 matrix with 2x3 matrix
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-1-5a8b21790e76> in <module>
```

```
----> 1 np.dot(mat, mat)         # raises error, cannot multiply 2x3 matrix with␣
 ↪2x3 matrix

<__array_function__ internals> in dot(*args, **kwargs)

ValueError: shapes (2,3) and (2,3) not aligned: 3 (dim 1) != 2 (dim 0)
```

[37]:
```
np.dot(mat, mat.T)       # transpose second operand, works
```

[37]:
```
array([[ 5, 14],
       [14, 50]])
```

1.5.3 Reductions

Reductions are operations that reduce the dimensionality of the data. For example, computing the mean of an array reduces a collection of data points to a single scalar, its mean.

Basic reduction operations include:

- `np.sum()`: sum of array elements
- `np.prod()`: product of array elements
- `np.amin()`, `np.amax()`: minimum and maximum element
- `np.argmin()`, `np.argmax()`: location of minimum and maximum element
- `np.mean()`, `np.average()`: mean of array elements
- `np.median()`: median of array elements
- `np.std()`, `np.var()`: standard deviation and variance of array elements
- `np.percentile()`: percentiles of array elements

Most if not all reductions accept an `axis` argument which restricts the operation to a specific axis.

- The resulting array will have one dimension less than the input.
- If no axis is specified, the operation is applied to the flattened array

Examples:

[38]:
```
import numpy as np

# 1-dimensional input data
data = np.linspace(0.0, 1.0, 11)

# Compute mean and std. of input data
m = np.mean(data)
s = np.std(data)
print('Mean: {:.2f}, std. dev.: {:.2f}'.format(m, s))
```

```
Mean: 0.50, std. dev.: 0.32
```

[39]:
```
# 2-dimensional input data
data = np.linspace(0.0, 1.0, 21).reshape((3, 7))
data
```

[39]:
```
array([[0.  , 0.05, 0.1 , 0.15, 0.2 , 0.25, 0.3 ],
       [0.35, 0.4 , 0.45, 0.5 , 0.55, 0.6 , 0.65],
       [0.7 , 0.75, 0.8 , 0.85, 0.9 , 0.95, 1.  ]])
```

[40]:
```
# Compute mean of each row, ie along the column axis
m = np.mean(data, axis=1)
m           # Result is a vector of 3 elements, one for each row
```

[40]:
```
array([0.15, 0.5 , 0.85])
```

1.5.4 Broadcasting [advanced]

Element-wise operations in most programming languages require input arrays to have identical shapes. NumPy relaxes this constraint and allows us to use arrays with different shares that can be "broadcast" to identical shapes.

What do we mean by "broadcast"? - Imagine we want to add a 2×3 matrix to a length-2 vector. - This operation does not make sense, unless we interpret the (column) vector as a 2×1 matrix, and replicate it 3 times to obtain a 2×3 matrix. This is exactly what NumPy does.

Examples:

[41]:
```
import numpy as np

# Create 3x2 matrix
mat = np.arange(6).reshape((2, 3))
mat
```

[41]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

[42]:
```
# Create 2-element vector
vec = np.arange(2)
vec
```

[42]:
```
array([0, 1])
```

[43]:
```
# Trying to add matrix to vector fails
mat + vec
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-1-d51c7a88afa4> in <module>
      1 # Trying to add matrix to vector fails
----> 2 mat + vec

ValueError: operands could not be broadcast together with shapes (2,3) (2,)
```

[44]:
```
# However, we can explicitly reshape the vector to a 2x1 column vector
colvec = vec.reshape((-1, 1))
colvec
```

[44]:
```
array([[0],
       [1]])
```

[45]:
```
# Now, broadcasting replicates column vector to match matrix columns
mat + colvec
```

[45]:
```
array([[0, 1, 2],
       [4, 5, 6]])
```

We do not need to `reshape()` data, but can instead use a feature of NumPy that allows us to increase the number of dimensions on the spot:

[46]:
```
# use vec[:, None] to append an additional dimension to vec
mat + vec[:, None]
```

[46]:
```
array([[0, 1, 2],
       [4, 5, 6]])
```

For more examples, see the official NumPy tutorial on broadcasting.

1.6 NumPy data types [advanced]

1.6.1 Default data types

We have already touched upon the numerical data types used by NumPy. If we do not explicitly request a data type using the `dtype` keyword argument, NumPy by default behaves as follows:

1. The default data type for most array creation routines which create arrays of a given shape or size, such as `np.empty()`, `np.ones()` and `np.zeros()`, is a 64-bit floating-point number (`np.float64`).
2. Array creation routines that accept numerical input data will use the data type of this input data to determine the array data type.

```
Examples of such functions are `np.arange()` and
`np.array()`.
```

3. Arrays that are implicitly created as a result of an operation (addition, etc.) are assigned the most suitable type to represent the result.

```
For example, when adding a floating-point and an
integer array, the result will be a floating-point array.
```

Examples:

Case 1: default data type is `np.float64`:

[47]:
```
import numpy as np

x = np.ones(1)      # length-1 vector of ones
x.dtype             # default type: float64
```

[47]:
```
dtype('float64')
```

Case 2: data type depends on input data:

[48]:
```
# Argument is an integer
x = np.arange(5)
x.dtype     # data type is np.int64 (on most platforms)
```

[48]:
```
dtype('int64')
```

[49]:
```
# Argument is a float
x = np.arange(5.0)
x.dtype     # data type is np.float64
```

[49]:
```
dtype('float64')
```

Case 3: data type determined to accommodate result

[50]:
```
# Add two integer arrays
arr1 = np.arange(3)
arr2 = np.arange(3, 0, -1)      # creates [3, 2, 1]
result = arr1 + arr2
result.dtype        # data type is np.int64
```

[50]:
```
dtype('int64')
```

[51]:
```
# Add integer to floating-point array
arr1 = np.arange(3)
arr2 = np.arange(3.0, 0.0, -1.0)    # creates [3.0, 2.0, 1.0]
result = arr1 + arr2
```

```
print(result)
result.dtype        # data type is np.float64
```

```
[3. 3. 3.]
```

```
[51]: dtype('float64')
```

Observe that even though the resulting array is `[3.0, 3.0, 3.0]` and can thus be represented as integers without loss of data, NumPy only takes into account that one of the operands is floating-point, and thus the result has to be of floating-point type!

1.6.2 Explicit data types

We can almost always explicitly request an array to be of a particular data type by passing the `dtype` keyword argument. The most common types are:

- `np.float64`: a 64-bit floating-point number, also called *double precision* in other languages.

  This is the most commonly used floating-point data type. It can represent numbers with up to 16 decimal digits, and covers a range of approximately $\pm 10^{308}$.

- `np.int64`: a 64-bit integer which can represent integer values on the interval of (approximately) $\pm 10^{19}$.

  Unlike with floating-point, the integer representation is *exact*, but covers a much smaller range (and, obviously, no fractional numbers)

- `np.float32`, `np.float16`: single-precision and half-precision floating-point numbers. These occupy only 32 and 16 bits of memory, respectively.

  They thus trade off storage requirements for a loss of precision and range.

- `np.int32`, `np.int16`, `np.int8` represent integers using 32, 16 and 8 bits, respectively.

  They require less memory, but can represent only a smaller range of integers. For example, `np.int8` can only store integer values from $-$128 to 127.

- NumPy also supports complex numerical types to represent imaginary numbers. We will not be using those in this tutorial.

Would we ever want to use anything other than the default data types, which in most cases are either `np.float64` and `np.int64`? These, after all, support the largest range and highest precision. This is true in general, but there are special cases where other data types need to be used:

1. *Storage requirements:* if you work with large amounts of data, for example arrays with many dimensions, you can run out of memory or storage space (when saving results to files).

   In this case, you can store data as `np.float32` instead of `np.float64`, which halves the storage requirement.

   Similarly, if you know that your integer data only takes on values between $-$128 and 127, you can store them as `np.int8` which consumes only 1/8 of the space compared to `np.int64`!

2. *Performance:* Some tasks simply don't require high precision or range. For example, some machine learning tasks can be performed using only 8-bit integers, and companies like Google have developed dedicated processors to considerably speed up workloads using 8-bit integers.

   Even if you are not using any dedicated CPUs or GPUs, data has to be transferred from memory to the processor, and this is a major performance bottleneck. The less data needs to be transferred, the better!

   In general, this is nothing you need to worry about at this point, but might become relevant in the future.

Examples:

[52]:
```
import numpy as np

# Explicitly specify data type
x = np.ones(1, dtype=np.float16)
x       # prints np.float16
```

[52]:
```
array([1.], dtype=float16)
```

We can use `dtype` to override the data type inferred from input data:

[53]:
```
lst = [1, 2, 3]
x = np.array(lst)       # given list of integers, creates integer array
x.dtype                 # prints np.int64
```

[53]:
```
dtype('int64')
```

[54]:
```
# override inferred data type:
# created floating-point array even if integers were given
x = np.array(lst, dtype=np.float64)
x.dtype                 # prints np.float64
```

[54]:
```
dtype('float64')
```

[55]:
```
# override inferred data type:
# created integer array even if floats were given,
# thus truncating input data!
lst = [1.234, 4.567, 6.789]
x = np.array(lst, dtype=np.int)
print(x)                # prints [1, 4, 6]
x.dtype                 # prints np.int64
```

```
[1 4 6]
```

[55]:
```
dtype('int64')
```

1.7 Copies and views [advanced]

Recall that assignment in Python does *not* create a copy (unlike in *C*, *Fortran* or *Matlab*):

[56]:
```
a = [0, 0, 0]
b = a           # b references the same object as a
b[1] = 1        # modify second element of b (and a!)
a == b          # a and b are still the same
```

[56]:
```
True
```

NumPy adds another layer to this sort of data sharing: whenever you perform an assignment or indexing operation, NumPy tries hard *not* to copy the underlying data but instead creates a so-called view which points to the same block of memory. It does this for performance reasons (copying is expensive).

We can illustrate this using array slicing:

[57]:
```
import numpy as np

x = np.arange(10)
x
```

[57]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[58]:
```
y = x[3:8]          # Create array that points to elements 4-8 of x
y
```

[58]:
```
array([3, 4, 5, 6, 7])
```

`x` and `y` are two different Python objects, which we can verify using the built-in `id()` function:

[59]:
```
print(id(x))
print(id(y))
```

```
139975814880992
139975814503360
```

[60]:
```
id(x) == id(y)
```

[60]:
```
False
```

And yet, the NumPy implementation makes sure that they reference the same block of memory!

We can see this easily by modifying `y`:

[61]:
```
y[:] = 0        # overwrite all elements of y with zeros
y
```

[61]:
```
array([0, 0, 0, 0, 0])
```

[62]:
```
x               # elements of x that are also referenced by y
                # are now also zero!
```

[62]:
```
array([0, 1, 2, 0, 0, 0, 0, 0, 8, 9])
```

This behaviour is even triggered when `y` references non-adjacent elements in `x`. For example, we can let `y` be a view on every *second* element in `x`:

[63]:
```
x = np.arange(10)
y = x[::2]      # y now points to every second element of x
y[:] = 0        # overwrite all elements of y with zeros
x               # every second element in x is now zero!
```

[63]:
```
array([0, 1, 0, 3, 0, 5, 0, 7, 0, 9])
```

As a rule of thumb, NumPy will create a view as opposed to copying data if

- An array is created from another array via slicing (ie indexing using the `start:stop:step` triplet)

Conversely, a *copy* is created whenever

- An array is created from another array via boolean (mask) indexing.
- An array is created from another array via integer array indexing.

You can always force NumPy to create a copy by calling `np.copy()`!

Examples:

[64]:
```
# Copies are created with boolean indexing
x = np.arange(10)
mask = (x > 4)      # boolean mask
y = x[mask]         # create y using boolean indexing
y[:] = 0
x                   # x is unmodified
```

[64]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[65]:
```
# Copies are creating with integer array indexing
x = np.arange(10)
index = [3, 4, 5]   # List of indices to include in y
y = x[index]
y[:] = 0
x                   # x is unmodified
```

[65]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[66]:
```
# Forced copy with slicing
x = np.arange(10)
y = np.copy(x[3:8]) # force copy with np.copy()
y[:] = 0
x                   # x is unmodified
```

[66]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

As an alterantive to `np.copy()` we can directly call the `copy()` method of an array:

[67]:
```
y = x[3:8].copy()
```

1.8 Array storage order [advanced]

Computer memory is linear, so a multi-dimensional array is mapped to a one-dimensional block in memory. This can be done in two ways:

1. NumPy uses the so-called *row-major order* (also called *C order*, because its the same as in *C* programming language)
2. This is exactly the opposite of Matlab, which uses *column-major order* (also called *F order*, because its the same as in the *Fortran* programming language)

[68]:
```
import numpy as np

mat = np.arange(6).reshape((2,3))
mat
```

[68]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

[69]:
```
# The matrix mat is stored in memory like this
mat.reshape(-1, order='C')
```

[69]:
```
array([0, 1, 2, 3, 4, 5])
```

[70]:
```
# ... and NOT like this
mat.reshape(-1, order='F')      # use order='F' to convert to column-major␣
↪storage order
```

[70]:
```
array([0, 3, 1, 4, 2, 5])
```

While this is not particularly important initially, as an advanced user you should remember that you never want to perform on non-contiguous blocks of memory. This can have devastating effects on performance!

[71]:
```
# Avoid operations on non-contiguous array sections such as
mat[:, 1]

# Contiguous array sections are fine
mat[1]
```

```
[71]: array([3, 4, 5])
```

2 Exercises

TBA

3 Solutions

TBA