NumPy

November 23, 2020

1 *NumPy* and *SciPy*

1.1 *NumPy*

1.1.1 What is *NumPy*?

NumPy is an open-source package **designed for working with multi-dimensional arrays**.

It includes routines for linear algebra and statistical operations, Fourier transforms, I/O functions, sorting algorithms and much more.

NumPy comes with a multi-dimensional array implementation that is designed for scientific computation, but has **broader applications** as its arrays are **efficient** multi-dimensional **containers of generic data** and, considering the ubiquity of (multi-dimensional) arrays, *NumPy* has applications in a wide range of domains.

NumPy is very fast. In fact, the library itself is mostly an interface for the underling C and Fortran algorithms that are very optimized. For instance, they take advantage of **locality of reference** and thanks to the memory contiguity of the array implementation.

Import convention *NumPy* is conventionally imported as *np* in the code.

[1]:
```
import numpy as np
```

1.1.2 Data structure

The core object is the **homogeneous multidimensional array**, called *ndarray*, which consists of:

- The raw array data (***data buffer***), a **contiguous** and **fixed-size block** of memory
- A set of attributes about the memory layout of the *data buffer* that include, for instance:
  - Data type (called *dtype*)
  - Byte-order
  - Size of each element (and, therefore, the stride)
  - In memory order (i.e., elements are serialized in memory column or row-wise?)

Note that **memory can be shared** with other objects!

- More *ndarrays* can share the same memory location
- *ndarray* items can also be references

1.1.3 Creating arrays

NumPy allows the creation of arrays in many ways.

From iterables For instance, lists or tuples.

```
[2]: a = list(range(9))

     np.array(a)
```

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

From generators This creation method requires specifying the data-type (dtype).

```
[3]: generator = range(9)

     np.fromiter(generator, dtype=int)
```

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

From built-in functions

arange *arange* is similar to *python*'s built-in *range*.

numpy.arange([start,]stop, [step,]dtype=None)

Given a *start* and an *end* values, it will produce a sequence of evenly spaced numbers with distance *step* between two continous values.

Note that the *end* value is not included (i.e., the considere interval is $[start, end)$)

```
[4]: np.arange(9)
     np.arange(1, 9, 1)
```

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

arange also supports float steps!

```
[5]: np.arange(0, 1, 0.1)
```

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

linspace *linspace* creates an array with a specified number of elements that are spaced equally between the specified beginning and end values.

```
[6]: np.linspace(0, 1, 5)
```

```
[6]: array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

From a given shape This is useful, for instance, if elements are unknown a priori.

Arrays of zeros

```
[7]: np.zeros(shape=(3,3), dtype=np.int)
```

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

Arrays of ones

```
[8]: np.ones(shape=(3,3), dtype=np.float)
```

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

Uninitialized arrays That is, memory location is *taken as is* and is not filled with any value.

This is faster if our algorithm is going to fill every location anyway.

```
[9]: np.empty(shape=(3,3), dtype=int)
```

```
[9]: array([[4607182418800017408, 4607182418800017408, 4607182418800017408],
            [4607182418800017408, 4607182418800017408, 4607182418800017408],
            [4607182418800017408, 4607182418800017408, 4607182418800017408]])
```

Identity matrices

```
[10]: np.eye(3, dtype=int)
```

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

Arrays of random values

```
[11]: np.random.rand(3,3)
```

```
[11]: array([[0.35093176, 0.1355403 , 0.54983159],
             [0.13184183, 0.94931174, 0.26727931],
             [0.22697441, 0.1688725 , 0.10444577]])
```

```
[12]: np.random.randint(10, size=(3,3))
```

```
[12]: array([[1, 1, 6],
             [4, 0, 8],
             [9, 7, 0]])
```

From files *NumPy* can load arrays from files. For instance:

- Text files can be parsed into arrays using the *loadtxt* function
- Text and binary files can be loaded using *fromfile*
- *ndarrays* can be stored in loaded using *save* and *load* respectively

1.2 *ndarray* attributes

Let's have a look to the *ndarray* main attributes.

[13]:
```
# Let's consider a generic array _a_
a = np.random.rand(3,3)

a
```

[13]:
```
array([[0.15574837, 0.33004759, 0.54087412],
       [0.83172004, 0.06515084, 0.4215084 ],
       [0.06689276, 0.48353281, 0.71274245]])
```

1.2.1 Info about the array

These attributes return some info about the array itself.

- *.ndim* : number of array dimensions (each called *axis*)
- *.shape* : tuple of array dimensions (elements in each axis)
- *.size* : number of elements in the array
- *.dtype* : data-type of the array's elements

[14]:
```
print("Number of dimensions {}".format(a.ndim))
print("Shape {}".format(a.shape))
print("Size {}".format(a.size))
print("Data type {}".format(a.dtype))
```

```
Number of dimensions 2
Shape (3, 3)
Size 9
Data type float64
```

1.2.2 Transformed array

These attributes return some transformation on the original array.

For instance:

- *.T* : the transposed array
- *.real* : the real part of the array
- *.imag* : the imaginary part of the array

[15]:
```
print("Transposed array\n{}\n".format(a.T))
print("Real part of the array\n{}\n".format(a.real))
print("Imaginary part of the array\n{}\n".format(a.imag))
```

```
Transposed array
[[0.15574837 0.83172004 0.06689276]
 [0.33004759 0.06515084 0.48353281]
 [0.54087412 0.4215084  0.71274245]]

Real part of the array
[[0.15574837 0.33004759 0.54087412]
 [0.83172004 0.06515084 0.4215084 ]
 [0.06689276 0.48353281 0.71274245]]

Imaginary part of the array
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

1.2.3 Memory attributes

These attributes return information about the *data buffer* and its memory allocation.

- *.flags* : information about the memory layout of the array
- *.itemsize* : length of one array element in bytes
- *.nbytes* : bytes occupied by the elements of the array
- *.base* : base object if memory is from some other object

[16]:
```
print("Array flags\n{}".format(a.flags))

print("Item size {}\n".format(a.itemsize))

print("Number of bytes {}\n".format(a.nbytes))

print("Base object (if any) {}\n".format(a.base))
```

```
Array flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False

Item size 8

Number of bytes 72

Base object (if any) None
```

Base object example Given a view, the view and the original array will share memory!

[17]:
```
# Let's assign a slice of a to 'b'
b = a[2:]
print(b, "\n")

# Base object will be not null!
print("Base object (if any)\n{}\n".format(b.base))
```

```
[[0.06689276 0.48353281 0.71274245]]

Base object (if any)
[[0.15574837 0.33004759 0.54087412]
 [0.83172004 0.06515084 0.4215084 ]
 [0.06689276 0.48353281 0.71274245]]
```

[18]:
```
# What happens to a if we assign to b?
b[0] = 0

print(a)
```

```
[[0.15574837 0.33004759 0.54087412]
 [0.83172004 0.06515084 0.4215084 ]
 [0.         0.         0.        ]]
```

1.2.4 Miscellaneous

Other attributes worth mentioning include *flat*:

- *.flat* : returns a 1-D iterator over the array

[19]:
```
print("1-D iterator over the array\n{}".format(a.flat))
```

```
1-D iterator over the array
<numpy.flatiter object at 0x55567461f450>
```

1.3 *ndarray* methods and functions

As mentioned in the introduction, *NumPy* comes with a number of algorithms that operate on the *ndarray*.

Some of them are in form of functions, others of *ndarray* methods, and some methods are just equivalent to invoking the functions on the array.

In this lecture we will focus on the most important, divided by type.

[20]:
```
# Let's init an array
a = np.arange(0, 9, 1, dtype=np.int64)
```

1.3.1 Casting

The *.astype(dtype)* method casts the array to the given data type.

Note that it returns **a copy** of the array unless:

- the *copy* parameter is passed as false **and**…
- … the memory allocation is compatible with the new type

The *order* parameter allows to change the memory serialization order, i.e., Fortran's column first (F) or C's row first (C) style.

[21]:
```
b = a.astype(int, copy=False)
print(b)
print(b.flags)
```

```
[0 1 2 3 4 5 6 7 8]
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

[22]:
```
b = a.astype(np.int64, order="F", copy=False)
print(b)
print(b.flags)
print(b.base)
```

```
[0 1 2 3 4 5 6 7 8]
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False

None
```

[23]:
```
print(a.astype(complex))

b = a.astype(np.int64, order="F", copy=False)
print(b)
print(b.flags)
print(b.base)
```

```
[0.+0.j 1.+0.j 2.+0.j 3.+0.j 4.+0.j 5.+0.j 6.+0.j 7.+0.j 8.+0.j]
```

```
[0 1 2 3 4 5 6 7 8]
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False

None
```

1.3.2 Manipulations

Conversion to higher dimensional arrays *NumPy* has helpers to convert the array to a multidimensional array with at least 1, 2 or 3 dimensions or to a matrix.

```
[24]: print("1D", np.atleast_1d(a), "\n")
print("2D", np.atleast_2d(a), "\n")
print("3D", np.atleast_3d(a), "\n")
print("Matrix", np.mat(a))
```

```
1D [0 1 2 3 4 5 6 7 8]

2D [[0 1 2 3 4 5 6 7 8]]

3D [[[0]
  [1]
  [2]
  [3]
  [4]
  [5]
  [6]
  [7]
  [8]]]

Matrix [[0 1 2 3 4 5 6 7 8]]
```

Concatenation The *concatenate* function allows to concatenate arrays along a given axis.

```
[25]: b = a.copy()

print("Concatenating 1D arrays\n", np.concatenate((a, b), axis=0), "\n")

print("Concatenating the rows\n", np.concatenate((np.atleast_2d(a), np.
 ↪atleast_2d(b)), axis=0), "\n")

print("Concatenating the columns\n", np.concatenate((np.atleast_2d(a), np.
 ↪atleast_2d(b)), axis=1), "\n")
```

```
Concatenating 1D arrays
 [0 1 2 3 4 5 6 7 8 0 1 2 3 4 5 6 7 8]

Concatenating the rows
 [[0 1 2 3 4 5 6 7 8]
 [0 1 2 3 4 5 6 7 8]]

Concatenating the columns
 [[0 1 2 3 4 5 6 7 8 0 1 2 3 4 5 6 7 8]]
```

Reshape You can also reshape an array without changing its data (and memory allocation).

[26]:
```
c = a.reshape((3,3))

print("3x3 shape\n", c, "\n")
print("Flags\n", c.flags, "\n" "C is a view!", "\n")
```

```
3x3 shape
 [[0 1 2]
 [3 4 5]
 [6 7 8]]

Flags
   C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False

C is a view!
```

Resize Resizes the memory allocation of the array.

Different behaviour if you invoke the *.resize* method or the *resize* function.

The *.resize* method works in-place and will create a new **a new array** if needed, while the latter will always allocate a new memory chunk.

[27]:
```
# Inplace
a.resize(3,3)
print(a, "\n")

# Copy
d = np.resize(b, (4,3))
print(d, "\n")
```

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

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

Diagonals You can get the diagonals of the array (main, but also upper, lower, etc), using the *diagonal(offset=0)* method.

[28]:
```
# Main
print("Main diagonal", a.diagonal())
# and with offset
print("Upper diagonal", a.diagonal(1))
print("Lower diagonal", a.diagonal(-1))
```

```
Main diagonal [0 4 8]
Upper diagonal [1 5]
Lower diagonal [3 7]
```

Squeeze Removes the single-entry dimentions from the array.

[29]:
```
# Removes
print("3D matrix\n", np.atleast_3d(a))
print("Squeezed 3D matrix\n", np.atleast_3d(a).squeeze())
```

```
3D matrix
 [[[0]
  [1]
  [2]]

 [[3]
  [4]
  [5]]

 [[6]
  [7]
  [8]]]
Squeezed 3D matrix
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
```

1.3.3 Querying

ndarray can be queried by simply comparing them to other objects.

Queries return boolean arrays that can be used as masks to filter the elements, in boolean expressions, etc.

For instance:

[30]:
```
a > 0
```

[30]:
```
array([[False,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]])
```

For now, let's init a boolean array and assume that it comes from some query.

[31]:
```
boolean_array = np.random.randint(low=0,
                                  high=2,
                                  size=(3,3),
                                  dtype=np.bool_)
print(boolean_array)
```

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

.all() and .any() These methods allow to check whether *all* or *any* elements in the specified axis are *True*.

That is, they perform an *AND* or *OR* along the provided axis.

If *axis* is *None*, the all dimensions are taken into account.

[32]:
```
print("Are the array elements ALL true? {}\n".format(boolean_array.all()))
print("Are the rows ALL true? {}\n".format(boolean_array.all(axis=0)))
print("Are the columns ALL true? {}\n".format(boolean_array.all(axis=1)))
```

```
Are the array elements ALL true? False

Are the rows ALL true? [False False False]

Are the columns ALL true? [False False False]
```

[33]:
```
print("Is ANY of the array elements true? {}\n".format(boolean_array.any()))
print("Is there a true in the rows? {}\n".format(boolean_array.any(axis=0)))
print("Is there a true in the columns? {}\n".format(boolean_array.any(axis=1)))
```

```
Is ANY of the array elements true? True
Is there a true in the rows? [False  True  True]

Is there a true in the columns? [ True  True False]
```

Chose values: ***.where()*** You can choose values according to some condition using the *.where(condition[, x, y])* method.

An element is returned if the condition is met, otherwise you can provide a fallback value.

Note that you must provide *condition*, but you cannot omit x or y if you provide the other.

If you provide *condition* only, *where* will return all the indices that meet the condition.

[34]:
```
np.where(boolean_array != False)
```

[34]:
```
(array([0, 1, 1]), array([2, 1, 2]))
```

If you provide x and y, for each position it will get the element from x if the *condition* is satisfied, otherwise the element from y.

[35]:
```
g = np.where(boolean_array != False, 4, 10)
print(g)
```

```
[[10 10  4]
 [10  4  4]
 [10 10 10]]
```

[36]:
```
h = np.where(g > 5, g * 10, np.sqrt(g))
print(h)
```

```
[[100. 100.   2.]
 [100.   2.   2.]
 [100. 100. 100.]]
```

Get non zero elements *nonzero* returns a tuple of arrays, one for each dimension of the array, containing the indices of the non-zero elements in that dimension (in C-style order!)

[37]:
```
print(np.nonzero(boolean_array))
```

```
(array([0, 1, 1]), array([2, 1, 2]))
```

1.3.4 Statistics

Here we get an overview of the statistics functions that come with *NumPy*.

The *mean* method returns the mean, *std* returns the standard deviation and *var* returns the variance.

[38]:
```
print("Array a:\n", a, "\n")
print("OVERALL: Mean {}, STD {}, VAR {}\n".format(a.mean(), a.std(), a.var()))
print("ROWS ONLY: Mean {}, STD {}, VAR {}\n".format(a.mean(axis=0), a.
↪std(axis=0), a.var(axis=0)))
print("COLUMNS ONLY: Mean {}, STD {}, VAR {}\n".format(a.mean(axis=1), a.
↪std(axis=1), a.var(axis=1)))
```

```
Array a:
 [[0 1 2]
 [3 4 5]
 [6 7 8]]

OVERALL: Mean 4.0, STD 2.581988897471611, VAR 6.666666666666667

ROWS ONLY: Mean [3. 4. 5.], STD [2.44948974 2.44948974 2.44948974], VAR [6. 6.
6.]

COLUMNS ONLY: Mean [1. 4. 7.], STD [0.81649658 0.81649658 0.81649658], VAR
[0.66666667 0.66666667 0.66666667]
```

1.3.5 Linear algebra

Linear algebra functions are included but we won't cover them here for sake of time.

1.4 Data types

Data types are stored in the *.dtype* attribute, which **describes** how **the** *data buffer* **memory** bytes should be interpreted.

It includes: * The **type** of the data (integer, float, (reference to) Python object, etc.) * The **size** of the data (how many bytes each element is long) * The byte order of the data (little-endian or big-endian) * If the data type is **structured** data type, it is an **aggregate of** other **data types** * If the data type is a **sub-array**, it specifies what its **shape** and **data type** are

It is worth noting that arbitrary data-types can be defined.

1.4.1 Built-in data types

NumPy comes with a greater variety of built-in numerical types and also knows how to interpret *python*'s.

Some examples are:

- Platform dependent *dtypes*
  - *bool_*, *int_*, *np.half*, *float_*, *ushort*, *uint*, *single*, *double*, *csingle*, *cdouble*, *clongdouble*, …
- Fixed size (platform independent) *dtypes*
  - *(u)int8*, *(u)int16*, *(u)int32*, *(u)int64*, *float32*, *float64*, *complex128* / *complex_*, …

Since some *dtypes* share names with *python*'s, their name has "_" appended to avoid conflicts.

Built-in *dtypes* can be accessed as *np.<DTYPE>*.

1.5 Array indexing

Accessing array's elements using indexes.

1.5.1 Monodimensional arrays

Indexing monodimensional arrays is as simple as accessing any *python* 's list.

[39]:
```
monodimensional_array = np.arange(0, 9, 1)
print(monodimensional_array)
print("Index: 0 ->", monodimensional_array[0])
print("Index: -1 ->", monodimensional_array[-1])
```

```
[0 1 2 3 4 5 6 7 8]
Index: 0 -> 0
Index: -1 -> 8
```

1.5.2 N-dimensional arrays

In case of **multidimensional arrays** , indexes become **tuples** of integers.

The *colon* (:) — meaning "select all indices along this axis" — can be used to select entire dimensions.

An *Ellipsis* — three dots (...) — expands the indexing tuple with needed colons (:) to index all dimensions, and they are automatically appended to the tuple if less dimensions than the ones in the array are provided.

As an example, we can either get the rows, the columns or a single element.

[40]:
```
print("Bidimensional array\n", a, "\n")
print("Index: 0 === (0, ...) === (0, :) ->", a[0])
print("Index: (:, 0) === (..., 0) ->", a[:, 0])
print("Index: (0, 0) ->", a[0, 0])
```

```
Bidimensional array
 [[0 1 2]
 [3 4 5]
 [6 7 8]]

Index: 0 === (0, …) === (0, :) -> [0 1 2]
Index: (:, 0) === (…, 0) -> [0 3 6]
Index: (0, 0) -> 0
```

Negative indices Negative indices are interpreted as $(n - |index|)$, where n is the number of elements in that dimension.

That is, you start counting from the bottom of the array.

[41]:
```
print("What about negative indices?")
print("Index: -1 ->", a[-1])
print("Index: (:, -1) === ..., 0 ->", a[:, -1])
print("Index: (-1, -1) ->", a[-1, -1])
```

```
What about negative indices?
Index: -1 -> [6 7 8]
Index: (:, -1) === …, 0 -> [2 5 8]
Index: (-1, -1) -> 8
```

Dimensionality reduction When indexing the array, the result indexing always has less dimensions than the indexed array. In fact, the indexed dimentions are not returned.

A selection tuple with k integer elements (and all other entries **:**) returns the corresponding subarray with dimension $N - k$.

If $N = 1$ then the returned object is a Scalar

A few examples:

[42]:
```
print("Bidimensional array\n", a.shape, "\n")
print("Index: 0 ->", a[0].shape)
print("Index: (0, 0) ->", a[0, 0].shape)
```

```
Bidimensional array
 (3, 3)

Index: 0 -> (3,)
Index: (0, 0) -> ()
```

[43]:
```
fourd_array = np.arange(0, 81, 1).reshape(3, 3, 3, 3)
print("Four dimensional array\n", fourd_array.shape, "\n")
print("Index: 0 ->", fourd_array[0].shape)
print("Index: (0, 0) ->", fourd_array[0, 0].shape)
print("Index: (0, ..., 0) ->", fourd_array[0, ..., 0].shape)
print("Index: (0, ..., 0, 0) ->", fourd_array[0, ..., 0, 0].shape)
```

```
Four dimensional array
 (3, 3, 3, 3)

Index: 0 -> (3, 3, 3)
Index: (0, 0) -> (3, 3)
Index: (0, …, 0) -> (3, 3)
Index: (0, …, 0, 0) -> (3,)
```

np.newaxis *newaxis* object can be used within array indices to add new dimensions with a size of 1.

[44]:
```
print("Index: (np.newaxis, 0) ->", a[np.newaxis, 0 ])
print("Index: (np.newaxis, :, 0) ->", a[np.newaxis, :, 0 ])
print("Index: (np.newaxis, :, np.newaxis, 0) ->\n", a[np.newaxis, :, np.
↪newaxis, 0 ])
```

```
Index: (np.newaxis, 0) -> [[0 1 2]]
Index: (np.newaxis, :, 0) -> [[0 3 6]]
Index: (np.newaxis, :, np.newaxis, 0) ->
 [[[0]
  [3]
  [6]]]
```

1.6 Slicing

Slicing consists in getting slices (pieces) of arrays, not just single elements/row/dimentions.

Syntax is *start:stop:step* and accessing such slice returns the indices specified by *range(start, stop, step)*.

None of the three components is **required**

- By default, *start* is 0, *end* is the last element index and *step* is 1

The single *colon* (*:*) is just a shortening for *::*

Negative *step* makes stepping go towards smaller indices

[45]:
```
print(monodimensional_array)
print("Slice: 0:4:1 ->", monodimensional_array[0:4:1])
print("Slice: 0:4 ->", monodimensional_array[0:4])
print("Slice: :4 ->", monodimensional_array[:4])
print("Slice: 4: ->", monodimensional_array[4:])
print("Slice: 4:0:-1 ->", monodimensional_array[4:0:-1])

print("Slice: 0:4:2 ->", monodimensional_array[0:4:2])
```

```
[0 1 2 3 4 5 6 7 8]
Slice: 0:4:1 -> [0 1 2 3]
Slice: 0:4 -> [0 1 2 3]
Slice: :4 -> [0 1 2 3]
Slice: 4: -> [4 5 6 7 8]
Slice: 4:0:-1 -> [4 3 2 1]
Slice: 0:4:2 -> [0 2]
```

Idiomatic way of reversing in python is slicing without specifying start or stop but just negative step.

[46]:
```
print("Slice: ::-1 ->", monodimensional_array[::-1])
```

```
Slice: ::-1 -> [8 7 6 5 4 3 2 1 0]
```

1.6.1 Dimensionality reduction

Since slicing returns pieces of the array, the sliced dimentions are included in the result.

This means that slicing returns an array with the same shape.

As an example, consider indexing at 0 vs slicing 0:1

[47]:
```
print("Index: 0 ->", monodimensional_array[0])
print("Slice: 0:1 ->", monodimensional_array[0:1])
```

```
Index: 0 -> 0
Slice: 0:1 -> [0]
```

[48]:
```
print("Bidimensional array", a.shape, "\n", a)
print("Slice: (:, 1:2) ->", a[:, 1:2].shape, "\n", a[:, 1:2])
```

```
Bidimensional array (3, 3)
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
Slice: (:, 1:2) -> (3, 1)
 [[1]
 [4]
 [7]]
```

However, you must **be careful when you combine slicing with indexing**!

[49]:
```
print("Four dimensional array", fourd_array.shape)
print("Slice: (0, 1:2) ->", fourd_array[0, 1:2, 2, ...].shape, "\n",␣
 ↪fourd_array[0, 1:2, 2, ...])
```

```
Four dimensional array (3, 3, 3, 3)
Slice: (0, 1:2) -> (1, 3)
 [[15 16 17]]
```

1.7 Advanced indexing

This kind of indexing is triggered when one of the indices is an iterable (like a list or an *ndarray*).

There are various options depending on the data type of the iterable.

1.7.1 Iterables of indexes

Iterables of integers, e.g., a list, a tuple or an array of indices to pick.

[50]:
```
print("Monodimensional array", monodimensional_array.shape)
print("Index: [0, 3, 5] ->", monodimensional_array[[0, 3, 5]])
```

```
Monodimensional array (9,)
Index: [0, 3, 5] -> [0 3 5]
```

[51]:
```
print("Bidimensional array\n", a)
print("Index: ([0, 1], [1, 2]) ->", a[[0, 1], [1, 2]])
```

```
Bidimensional array
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
Index: ([0, 1], [1, 2]) -> [1 5]
```

[52]:
```python
print("Four dimensional array", fourd_array.shape)
print("Index: ([0, 3, 5], [1, 2]) ->", fourd_array[[0, 1], [1, 2], [1, 2]].
 ↪shape, "\n", fourd_array[[0, 1], [1, 2], [1, 2]])
print("Index: ([0, 3, 5], [1, 2]) ->", fourd_array[[0, 1], [1, 2], [1, 2], [0,␣
 ↪1]].shape, "\n", fourd_array[[0, 1], [1, 2], [1, 2], [0, 1]])
```

```
Four dimensional array (3, 3, 3, 3)
Index: ([0, 3, 5], [1, 2]) -> (2, 3)
 [[12 13 14]
 [51 52 53]]
Index: ([0, 3, 5], [1, 2]) -> (2,)
 [12 52]
```

1.7.2 Masks

If the iterable is a **boolean array** (*mask*), the result of indexing will include only *True* elements.

The output will have the same shape of the array (one bool per element).

[53]:
```python
print("Monodimensional array", monodimensional_array, "\n")

# Let's create a mask
mask = monodimensional_array > 3

print("Mask", mask, "\n")
print("Masking with: monodimensional_array > 3 \n", monodimensional_array[mask])
```

```
Monodimensional array [0 1 2 3 4 5 6 7 8]

Mask [False False False False  True  True  True  True  True]

Masking with: monodimensional_array > 3
 [4 5 6 7 8]
```

… but you can **also specify just the dimensions** to include.

- This is **equivalent to** *array[mask, …]*
- The array is indexed by *mask* followed by as many *:* as are needed

Resulting array's shape is one dimension containing the number of *True* elements of the mask, followed by the remaining dimensions of the array being indexed.

As an example:

```
[54]: mask = np.array(False)
      print("Masking with:\n", mask, "\n", monodimensional_array[mask])
      mask = np.array(True)
      print("Masking with:\n", mask, "\n", monodimensional_array[mask])
```

```
Masking with:
 False
 []
Masking with:
 True
 [[0 1 2 3 4 5 6 7 8]]
```

```
[55]: print("Four dimensional array")

      mask = np.array([True, False, False])
      print("Masking with:\n", mask, mask.shape, "\n", fourd_array[mask], "\n",␣
       ↪fourd_array[mask].shape)
```

```
Four dimensional array
Masking with:
 [ True False False] (3,)
 [[[[ 0  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 26]]]]
 (1, 3, 3, 3)
```

1.8 Views

Array views allow accessing an array restricting the indices and/or virtually changing its data type.

Views will **share memory** with the original array! * Changes to the view will modify the array * Viceversa, changes to the array will affect its views

Views can be created by

- Slicing an array
- Changing its *dtype* (this can be tricky!)
- Both

Note that while slicing **always** returns a view, advanced indexing **never** does.

```
[56]: # Let's create a view by slicing an array
      monodimensional_view = monodimensional_array[0:3]
      print("View of the 1D array:\n", monodimensional_view, "\nFlags:\n",␣
       ↪monodimensional_view.flags)
```

```
View of the 1D array:
 [0 1 2]
Flags:
   C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

```
[57]: # and assign to that view
      monodimensional_view[1] = 50

      # What happens to the array?
      print(monodimensional_array)
```

```
[ 0 50  2  3  4  5  6  7  8]
```

1.9 Numpy functions

Many mathematical functions are built-in in form of "*universal functions*" (*ufunc*).

They operate element-wise on an array producing a **new** array as output.

They apply automatic broadcast (handling of different shapes), casting, etc…

They are implemented in C to improve performance

They may have optional keyword arguments:

- *where*: array mask (apply function at that position or not)
- *casting*: kind of casting to apply, if any

Available *ufuncs* include:

- Math operations
- Trigonometric functions
- Bit-twiddling functions
  - They manipulate the bit-pattern of their integer arguments
- Comparison functions
- Floating functions

A few *ufunc* examples are:

[58]:
```
print("g\n", g)
print("g + 1\n", np.add(g, 1))
print("g + a\n", np.add(g, a))
```

```
g
 [[10 10  4]
 [10  4  4]
 [10 10 10]]
g + 1
 [[11 11  5]
 [11  5  5]
 [11 11 11]]
g + a
 [[10 11  6]
 [13  8  9]
 [16 17 18]]
```

[59]:
```
np.square(monodimensional_array, where=monodimensional_array > 4)
```

[59]:
```
array([  10, 2500,    6,   13,    8,   25,   36,   49,   64])
```

[60]:
```
np.sin(monodimensional_array, where=monodimensional_array < 6.28)
```

[60]:
```
array([ 0.        , 50.        ,  0.90929743,  0.14112001, -0.7568025 ,
       -0.95892427, -0.2794155 ,  7.        ,  8.        ])
```

1.10 *SciPy* brief overview

SciPy is a collection of mathematical algorithms and convenience functions built on top of *NumPy*.

It is written in *C* and *Fortran* and takes the best of the two languages and of their libraries while providing simple access in *python*.

Its main sub-packages include:

- Clustering algorithms
- Physical and mathematical constants
- Fast Fourier Transform routines
- Integration and ordinary differential equation solvers
- Linear algebra
- N-dimensional image processing
- Optimization and root-finding routines
- Signal processing
- Sparse matrices and associated routines
- Spatial data structures and algorithms
- Statistical distributions and functions

1.11 References

- Scipy Lectures
- NumPy docs: quickstart
- NumPy docs: ndarray
- NumPy docs: NumPy internals
- NumPy docs: dtypes
- SciPy: intro