6 Introduction to NumPy

Vectors, matrices and higher-dimensional arrays are essential for numerical computing. **Vectorised computing**, computations that are formulated in terms of array operations, eliminate the need for explicit loops over the array elements. The result is a more concise and readable code. Plus, vectorised operations are usually much faster than sequential element-by-element operations. However, built-in Python types do not support mathematical operations in arrays very well. An additional module called **NumPy** provides efficient data structures for scientific computing.

The `NumPy` module is not a part of the standard Python installation, however it is included in the Anaconda environment. `NumPy` introduces a new data object, an `ndarray` – similar to lists, but more easily manipulated by mathematical functions included in the module. `Ndarrays arrays` are a much better choice to implement matrices or simply do vector operations than standard Python objects. Unlike lists, `arrays` are homogenous – **each array can store only one type of data**. The size of the `array` is immutable and empty elements (with 0 dimensions) are not allowed. Apart from the `array`, NumPy provides many operators and functions that act on these data structures, as well as submodules implementing algorithms, e.g linear algebra or Fast Fourier Transformation.

To start using the module functionality, an import of the library is necessary. By convention, `NumPy` is imported under an alias `np` (example `[1]`). After that, any function or object can be accessed with a `np` namespace. In order to check all the functions available in `NumPy`, try using a `dir` command (example `[1]`). Using `dir` on an object will provide a list of methods available for the object.

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

#print(dir(np)) # check all the functions in np module
arr = np.array([1, 2, 3])
print(dir(arr)) # check methods for arr object
```

```
['T', ..., 'all', 'any', 'argmax', 'argmin', 'argpartition', 'argsort', 'astype', '
    base', 'byteswap', 'choose', 'clip', 'compress', 'conj', 'conjugate', 'copy', '
    ctypes', 'cumprod', 'cumsum', 'data', 'diagonal', 'dot', 'dtype', 'dump', 'dumps
    ', 'fill', 'flags', 'flat', 'flatten', 'getfield', 'imag', 'item', 'itemset', '
    itemsize', 'max', 'mean', 'min', 'nbytes', 'ndim', 'newbyteorder', 'nonzero', '
    partition', 'prod', 'ptp', 'put', 'ravel', 'real', 'repeat', 'reshape', 'resize'
    , 'round', 'searchsorted', 'setfield', 'setflags', 'shape', 'size', 'sort', '
    squeeze', 'std', 'strides', 'sum', 'swapaxes', 'take', 'tobytes', 'tofile', '
    tolist', 'tostring', 'trace', 'transpose', 'var', 'view']
```

6.1 Arrays

Probably the most important feature of the `NumPy` library is the data structure used for representing multi-dimensional arrays of **homogenous** data. Homogenous refers to elements of the array all having the same data type. The main data structure for arrays in `NumPy` is the `ndarray` class. Apart from the data in an array, the data structure also contains basic metadata, such as its size, shape, data type and number of dimensions (example `[2]`). A full list of attributes is listed in documentation and can be accessed by calling `help(np.ndarray)`. An `ndarray` instance is created by calling the function `np.array` with a nested list as an argument.

In [2]:
```
data = np.array([[1, 2, 3], [4, 5, 6]])
print(type(data))
print(data.size)
print(data.shape)
print(data.dtype)
```

```
<class 'numpy.ndarray'>
6
(2, 3)
int64
```

The `dtype` attribute describes the data type of each element in the array. Basic numerical data types (integer, unsigned integer, float, complex and boolean) are supported. Each of the data type come in different sizes (32 or 64 bits). It is important to pay attention which data type is used in the array – once it is defined, it will be used for all the subsequent operations, unless a new copy of the data is created with a type-casted array values (example `[3]`). Modyfing an item in an integer array with a floating point value will result in truncating the decimal part.

In [3]:
```
a = np.array([1, 2, 3], dtype=np.int)
b = np.array([1, 2, 3], dtype=np.float)
c = np.array(b, dtype=np.int)
print(c.dtype)

# alternative
d = c.astype(np.int)
```

```
int64
```

Computing with `NumPy` arrays can result in changing one data type to another, for example, adding an integer vector to a floating point vector returns a floating point vector.

The default data type is `float`. However, in some cases, it is necessary to set the data type to `int` or `complex`. Example `[4]` presents different results obtained with different types of data. Only `complex` values can give results for a square root computed from a negative value. In other cases a warning will appear.

In [4]:
```
# a = np.sqrt(np.array([-1, 0, 1])) # RuntimeWarning: invalid value encountered in
    sqrt array([ nan, 0.,1.])
b = np.sqrt(np.array([-1, 0, 1], dtype=complex))
```

```
[0.+1.j 0.+0.j 1.+0.j]
```

6.1.1 Creating arrays

One way to initilise an array is using `np.array` function used in previous section. But other functions may be more convenient, depending on the intended use and required properties:

`np.array` – creates an array from a given array-like object, for example, a (nested) list, a tuple, an iterable sequence or another `ndarray`,

`np.zeros` – creates an array filled with zeros in a specified shape (given as an integer or a tuple),

`np.ones` – creates an array filled with ones in a specified shape (given as an integer or a tuple), `np.empty` – creates an empty array in a specified shape (given as an integer or a tuple); the array is filled with uninitialised values (usually close to zero); all values should be explicitly assigned before the array is used to avoid unpredictable errors,

`np.full(shape, value)` – creates an array filled with a given value in a specified shape (given as an integer or a tuple),

`np.arange` – creates an array with evenly spaced values between specified start and end with a defined increment (similar to range, but a step can be a floating point number),

`np.diag` – creates a diagonal array with specified values along the diagonal (and zeros elsewhere),

`np.linspace` – creates an array with evenly spaced values between start and end using a specified number of elements; it is recommended to use `np.linspace` over `np.arange` wherever the increment is non-integer,

`np.logspace` – creates an array with logarithmically spaced values between the specified start and end values,

`np.fromfunction(function, size)` – creates an array and fills it with values specified by a given function evaluated for each combination of indices for the given array size,

`np.fromfile` – creates an array with the data from a text or binary file stored with a function `np.tofile`,

`np.genfromtxt, np.loadtxt` – creates an array from data read from a text file, for example, comma-separated value files (CSV); `np.genfromtxt` supports data with missing values,

`np.random.rand` – creates an array filled with random numbers which are uniformly distributed between 0 and 1 (other types of distribution also available),

`np.meshgrid` – generates coordinate matrices (or higher-dimensional coordinate arrays) from one dimensional coordinate vectors.

An existing array can be filled with a specified value with `fill` method, as in example `[5]`.

In [5]:
```python
a = np.empty(5)
print("empty a:", a)
a.fill(2.0)
print("filled a:", a)
```

```
empty a: [4.0e-323 9.9e-324 2.0e-323 2.5e-323 3.0e-323]
filled a: [2. 2. 2. 2. 2.]
```

It is not uncommon to create arrays based on the values or shapes of existing arrays. `NumPy` provides functions to perform such operations:

`np.ones_like(x)` – creates an array of the same properties as `x`, filled with ones,

`np.zeros_like(x)` – creates an empty array of the same properties as `x`, filled with zeros,

`np.full_like(x, n)` – creates an empty array of the same properties as `x`, filled with value `n`,

`np.empty_like(x)` – creates an empty array of the same properties as `x`.

A typical use-case involves taking an array of unspecified shape and type as argument and requiring array with similar properties.

Matrices are two-dimensional arrays commonly used for numerical computing. NumPy defines several functions used to define frequently used matrices:

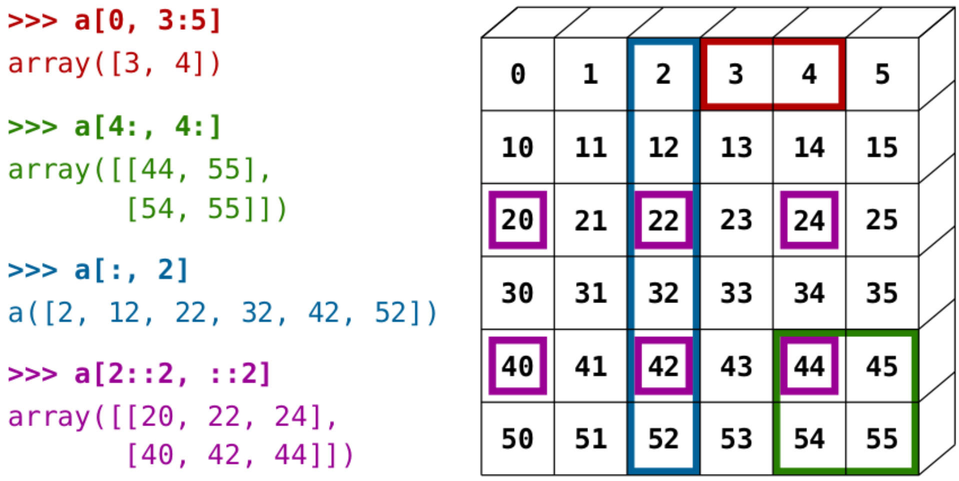

Figure 6.1. Examples of slicing an array (Source: `http://scipy-lectures.org/intro/numpy/array_object.html#indexing-and-slicing`)

`np.identity(n)` – generates an n×n identity matrix with ones on the diagonal and zeros elsewhere.

`np.eye(n, k=i)` – generates an n×n matrix with ones on a diagonal, optionally offset if k different than 0 is defined.

6.2 Indexing

Elements of arrays are indexed using the square bracket operator (known from indexing sequences: lists, tuples and dictionaries) and sliced according **to the same rules** pertaining to sequences. An element within the bracket is a tuple, each item in a tuple refers to a different dimension (**axis**) of the array (Fig. 6.1). To select every second element in array `a`, starting from a third element (with an index 2) to penultimate element (second to last), we can use index slice `a[2:-1:2]`.

In multidimensional arrays, a single column or row can be selected by defining a colon (`:`) operator for the remaining dimensions, as in example `[6]`.

In [6]:
```
a = np.array([range(4), range(4, 8)])
print(a[:, 1])  # second column of a
print(a[0, :])  # first row of a
```

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

Subarrays that are extracted in slicing operations are **views** of the same source array data – they refer **to the same data in memory** as the original array. When an element in a view is reassigned, the value **in an original array is also updated** (example `[7]`).

In [7]:
```
a = np.array([range(4), range(4, 8)])
b = a[1:4, 1:4]
b[0, 0] = 10
print(a)
```

```
[[ 0  1  2  3]
 [ 4 10  6  7]]
```

When a copy is needed, use a `copy` method instead (example `[8]`) or `np.array` function with the keyword argument `copy=True`. Using `copy` creates an independent set of data in memory, a modification on the copied values do not interact with the original data.

In [8]:
```
a = np.array([range(4), range(4, 8)])
b = a[1:4, 1:4].copy()
b[0, 0] = 10    # does not affect a
print(a)
```

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

6.2.1 Fancy indexing and boolean indexing

`NumPy` has another way of indexing, not available for built-in sequences, called **fancy indexing**. With fancy indexing, arrays can be indexed using another array, a list or a sequence of integers (example `[9]`). This method can be used along any axis in a multi-dimensional array.

In [9]:
```
a = np.linspace(0, 21, 10)
b = np.array([1, 3, 5])
print(a[b]) #indexing using an array
print(a[[1, 1, 5, 3, 5]]) # indexing using a list
```

```
[ 2.33333333 7.         11.66666667]
[ 2.33333333 2.33333333 11.66666667 7.         11.66666667]
```

Another method uses **boolean values to index arrays**. In this case, each element (with `True` or `False` value) indicates whether or not to select the element from the array with a corresponding index: if an element with index `n` in boolean-valued array is `True`, then the element `n` is selected from the indexed array, otherwise it is not selected. This method is very convenient in filtering out the elements of the array, for example, choosing the elements that meet a specific condition (example `[10]`).

In [10]:
```
a = np.linspace(0, 21, 10)
mask = a > 10
print(mask)
print(a[mask]))
```

```
[False False False False False True True True True True]
[11.66666667 14.         16.33333333 18.66666667 21.        ]
```

Unlike arrays created using slices, the arrays returned using fancy indexing and boolean indexing are **independent arrays**, not views. Changes in arrays obtained that way do not affect the original array.

6.3 Reshaping and resizing

`NumPy` provides a collection of functions useful to rearrange and manipulate the shape of an array:

`np.reshape, np.ndarray.reshape` – reshapes an N-dimensional array (creates a new view); the total number of elements must remain the same,

`np.transpose, np.ndarray.transpose, np.ndarray.T` – transposes the array (transpose operation reverses the axes of the array), `np.resize` – resizes an array – creates a new copy of the array with the given size; if necessary, the original array will repeat to fill up the new array,

`np.append` – appends an element to an array (creates a new copy of the array),

`np.insert` – inserts a new element at a given position (creates a new copy of the array),

`np.delete` – deletes an element at a given position (creates a new copy of the array),

`np.hstack` – stacks a list of arrays horizontally (along axis 1): for example, given a list of column vectors, appends the columns to form a matrix (example **??**),

`np.vstack` – stacks a list of arrays vertically (along axis 0): for example, given a list of row vectors, appends the rows to form a matrix,

`np.dstack` – stacks arrays depth-wise (along axis 2),

`np.concatenate` – creates a new array by appending arrays after each other, along a given axis,

`np.ndarray.flatten` – creates a copy of an N-dimensional array and reinterprets it as a one-dimensional array (all dimensions are collapsed into one),

`np.ravel, np.ndarray.ravel` – creates a view (if possible, otherwise a copy) of an N-dimensional array as a flattened, one-dimensional array; `np.ndarray.flatten` performs the same operation, but returns a copy of an array,

`np.squeeze` – removes axes with length 1,

`np.expand_dims, np.newaxis` – adds a new axis of length 1 to an array, where `np.newaxis` is used with array indexing.

Reshaping an array does not change the data in the memory, only a way in which the data is arranged. It is important to take into account the shape of the array while stacking matrices horizontally and vertically. One-dimensional results may not give intended results (example `[11]` and `[12]`). Stacked arrays produced by `np.hstack`, `np.vstack` and `np.concatenate` have the same number of dimensions as the input arrays.

In [11]:
```python
data = np.arange(5) # one-dimensional vector (5, )
result = np.hstack((data, data, data))
print(result)
```

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

In [12]:
```python
# two-dimensional vectors (1, 5)
data = data[:, np.newaxis] # insert new axis
result2 = np.hstack((data, data, data))
print(result2)
```

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

The number of elements in an array cannot be changed, therefore `np.insert`, `np.append` and `np.delete` create new array and copy the data. It is usually a good idea to preallocate arrays with size to avoid resizing them.

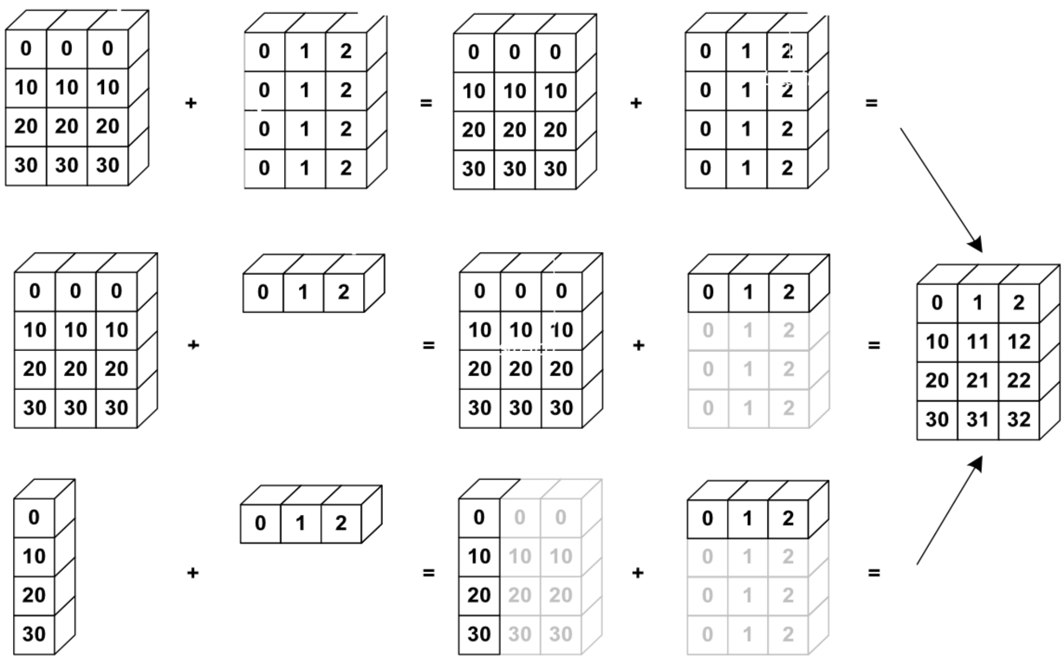

Figure 6.2. Broadcasting arrays (source: `http://scipy-lectures.org/intro/numpy/operations.html#broadcasting`)

6.4 Vectorized operations

Vectorized operations eliminate the explicit need for `for` loops, plus, they are more efficient than loop operations. Many of those require that the arrays are of compatible size, which usually means the same size and shape. More generally, an operation between two arrays is well defined if a result can be broadcasted into the same shape in size. In case of an operation involving a scalar and an array, **broadcasting** refers to applying the effect of the operation and scalar value to each element of the array (Fig. 6.2). An array can be broadcasted to another array if their sizes match at least in one dimension.

The standard arithmetic operations (addition, subtraction, multiplication and division, integer division and exponentiation) on `NumPy` arrays are performed element-wise. In computations involving a scalar and an array, the scalar value is applied to each element in the array. If an operation is performed on arrays with incompatible size, a `ValueError` is raised.

6.4.1 Elementwise functions

Apart from the arithmetic operators, `NumPy` provides vectorized functions for element-wise evaluation of many mathematical functions. Each of the functions listed below[1] takes an array of arbitrary dimension and returns a new array **of the same shape**, where each element is a **result of a function applied** to a corresponding item in the input array. Functions that can be used for element-wise operations include:

- square root: `np.sqrt`,
- exponential: `np.exp`,

[1]The list is by no means complete, the `NumPy` documentation includes a comprehensive list.

- logarithms of base 2, e and 10: `np.log2`, `np.log`, `np.log10`
- trigonometric functions: `np.sin`, `np.cos`, `np.tan`,
- inverse trigonometric functions: `np.arcsin`, `np.arccos`, `np.arctan`, `np.arctan2`.

Mathematical operations that can be applied to arrays are as follows:

- addition, subtraction, multiplication and division of two `NumPy` arrays, equivalents for `+`, `-`, `*` and \, respectively: `np.add`, `np.subtract`, `np.multiply`, `np.divide`,
- raising the input argument to the second input argument: `np.power`,
- the remainder of division: `np.remainder`,
- the inverse (reciprocal) of each element: `np.reciprocal`
- the real and imaginary part and conjugate of a complex number: `np.real`, `np.imaginery`, `np.conj`,
- rounding the values to a given number of decimals: `np.round`
- the absolute value and the sign of a number: `np.abs`, `np.sign`,
- integer approximates: `np.floor`, `np.ceil`, `np.rint`.

A constant value π can be accessed by typing `np.pi` (or `math.pi`).

If a function written for scalar input needs to be applied on an array, and it is not possible to express it in terms of existing vectorized `NumPy` functions, `np.vectorize` can be a convenient tool. This functions takes a non-vectorized function and returns a vectorized one (example `[13]`). However, the vectorized function can be still relatively slow.

In [13]:
```
def even_odd(x):
    return 0 if x%2 == 0 else 1

x = np.arange(10)
even_odd_vec = np.vectorize(even_odd)
print(even_odd_vec(x))
```

```
[0 1 0 1 0 1 0 1 0 1]
```

The rule of thumb while working with numpy arrays: use a NumPy (mathematic) function wherever possible (for example, not a function from `math` module, as they require looping over the whole array).

6.4.2 Aggregate funcions

Aggregate functions are functions that take an array as input and return a scalar value as a result. `NumPy` library includes a set of functions for calculating aggregates:

`np.sum` – returns a sum of all elements,

`np.cumsum` – returns a cumulative sum of all elements (returns an array),

`np.min, np.max` – returns a minimum / maximum value in an array,

`np.argmin, np.argmax` – returns the index of the minimum / maximum value in an array,

`np.mean` – returns an average of the values in an array,

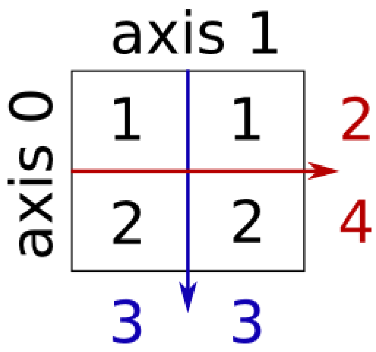

Figure 6.3. Summing values along the axis with `np.ndarray.sum` (Source: `http://scipy-lectures.org/intro/numpy/array_object.html`

`np.std` – returns a standard deviation of the value in an array,

`np.var` – returns a variance of the value in an array,

`np.prod` – returns a product of the value in an array,

`np.cumprod` – returns a cumulative product of the value in an array,

`np.all` – returns `True` if all elements in an array are nonzero,

`np.any` – returns `True` if at least one of the elements in an array is nonzero,

All of the above functions are also available as methods of the `ndarray` class (`np.sum(a)` and `a.sum()` are equivalent). By default those functions aggregate over the **entire array**. Using the `axis` keyword argument enables control over which each axis is the aggregation is carried out (example `[14]`, Fig. 6.3).

In [14]:
```
data = np.random.normal(size=(5, 10, 15))
print(data.sum(axis=0).shape)
print(data.sum(axis=(0, 2)).shape)
print(data.sum())
```

```
(10, 15)
(10,)
-27.572900193776086
```

6.5 Matrix operations

One of the main applications of two-dimensional arrays is to represent matrices and vectors and use them in matrix operations. In `NumPy`, `*` represents **element-wise** multiplication. Hence, in order to use matrix multiplication (multiplication according to linear algebra rules for matrices), two possiblities are available (Examples `[15]`–`[18]`):

- a `np.dot` function or `dot` method,
- `@` operator.

In [15]:
```
A = np.array([[1, 1],
              [0, 1]])
B = np.array([[2, 0],
         [3, 4]])
print("A * B = ")
print(A * B)    # elementwise product
```

```
A * B =
[[2 0]
[0 4]]
```

In [16]:
```
print("A @ B = ")
print(A @ B) # matrix multiplication
```

```
A @ B =
[[5 4]
[3 4]]
```

In [17]:
```
print("A.dot(B)")
print(A.dot(B)) # dot method
```

```
A.dot(B)
[[5 4]
[3 4]]
```

In [18]:
```
print("np.dot(A, B)")
print(np.dot(A, B)) # dot function
```

```
np.dot(A, B)
[[5 4]
[3 4]]
```

An alternative data structure called `matrix` was provided in `NumPy`. For `matrix`, the array multiplication was defined with `*` operator. `Matrix` had also a convenient attribute `matrix.I` returning an inverse of a matrix. However, using this structure had a few disadvantages and was discouraged. Nowadays, **the `matrix` object is discouraged**. Use ndarrays with `@` instead.

6.6 Linear algebra module

Many functions performing linear algebra algorithms and computations were implemented in `NumPy`. They are included in `NumPy.linalg` sub-package. Functions that may be useful during the course:

`multidot` – computes the dot product of two or more arrays (given as a sequence) in one function call; chains several `np.dot` function calls,

`inv(A)` – returns the matrix inverse of the 2D array `A`;

`det(A)` – returns the determinant of an array `A`; the determinant being a product of its singular values,

`norm(x)` – returns a matrix or a vector norm,

`cholesky(A)` – returns `L`, the Cholesky decomposition of `A`,

`qr(A)` – QR decomposition of a matrix `A`, `eigvals(A)` – returns all solutions (λ) to the equation $\mathbf{Ax} = \lambda\mathbf{x}$,

`eig(A)` – returns all solutions, tuples (λ, $\mathbf{x}$), to the equation $\mathbf{Ax} = \lambda\mathbf{x}$,

`solve(A, b)` – finds the solution to the linear equation $\mathbf{Ax} = \mathbf{b}$, where `A` is a 2D array and `b` is 1D or 2D array,

`lstsq(A, b, rcond='warn')` – returns the least-square solution to a linear matrix equation.

6.7 Exercises

Exercise 6.1. Create an 8x8 table (called `A`) and fill it with values. You can use random numbers or any other assignment that provides different values in rows (with at least one value equal to 1). Create a table called `B` that contains a part of `A` – a cross-section of 4 rows and 4 columns.

- assign a new value to `B[1, 1]`. Print table `A`. What happened? Why? What can you do to keep `A` intact?
- change all 1 in `A` to 3.

Exercise 6.2. Let `x = np.arange(12.0)`. Use `shape` and `reshape` to produce 1×12, 2×6, 3×4, 4×3, 6×2 versions of the array. Then, return `x` to its original size.

Exercise 6.3. Let `x = np.reshape(np.arange(12.0), (4, 3))`. Use `ravel`, `flat` and `flatten` to extract elements with indices: 0, 2, 4, . . .

Exercise 6.4. Use `hstack`, `vstack` and `tile` to construct a matrix `A`:

$$A = \begin{bmatrix} & y & y & y \\ x & & & \\ & y & y & y \\ & & z^T & \\ z & & & \\ & y & y & y \end{bmatrix}$$

Exercise 6.5. Given a vector $v = (2, 3, -1)$ and a function $f(X) = x^3 + xe^x + 1$, apply f to each element in v. Then calculate $f(v)$ using vector computing rules. Prove that the results are equal.

Exercise 6.6. Try out different methods to create an array. Next, create an array w with values $0, 0.1, 0.2 \ldots, 3$. What are results of calls: `w[:]`, `w[:-3]`, `w[::4]`, `w[2:-1:6]`.

Exercise 6.7. Solve the linear equation system $\mathbf{Ax} = \mathbf{b}$ using matrix methods and one of the `NumPy.linalg` functions.

Exercise 6.8. The table below contains an epoch (col 1), satellite vehicle number (col 2), the satellite azimuth (col 3) and elevation (col 4); azimuth and elevation are given in degrees.

- Find all the rows in which elevation ¿ azimuth;
- Find all the rows with elevation ¡ 15 degrees. Change those values to `None`;
- Find all the information for satellite 7 and save them to a new array;
- Divide the table into two parts, each of the new table should contain information for one epoch only;

Exercise 6.9. Given the arbitrary array `A` and using a loop, create a new array `B`:

- the first column of `B` should contain row indices;
- the second column of `B` should contain the row sum of `A`;
- the thirs column of `B` should contain the maximum value from a respective row of `A`;
- the fourth column of `B` should contain the sum of col 2 and col 3 in each row in `A`.

Exercise 6.10. Download the file with coordinates of the Polish CORS network ASG-EUPOS (`http://www.asgeupos.pl/webpg/_syst_descr_ref_st/ASGEUPOS_PL-ETRF2000_e2011_20130603.txt`) or use the file provided in the Files section of MS Teams.

- Read the file using `genfromtxt` or `loadtxt`. What happened to columns with text? What other information should be provided to 'genfromtxt' function? What is the separator of the columns? What about latitude and longitude? (*Hint: look at the beginning of the file*)
- Skip the columns with text (*Hint: function documentation: usecols/excludecols). Read the data.
- Create a column vector with IDs starting from 0 and with the same length as data in file. Add the column with ID to column with data.
- Choose all the IDs with heights ¿ 400 m. Save data of the selected points to a text file.

6.8 Useful links

- [Recommended] NumPy documentation. Tutorial for beginners: `https://numpy.org/devdocs/user/absolute_beginners.html`
- NumPy documentation. Quickstart: `https://numpy.org/devdocs/user/quickstart.html`
- SciPy lectures: `https://scipy-lectures.org/intro/numpy/index.html`
- Visual guide to NumPy: `https://betterprogramming.pub/numpy-illustrated-the-visual-guide-to-numpy-3b1d4976de1d`
- General NumPy documentation `https://NumPy.org/devdocs/reference/index.html`
- NumPy for Matlab users: `https://numpy.org/doc/stable/user/numpy-for-matlab-users.html`
- Matlab and Python differences: `https://realpython.com/matlab-vs-python/`
- More NumPy examples: `http://scipy-lectures.org/intro/numpy/index.html`
- Basics of `arrays`: `https://realpython.com/numpy-array-programming/`

References

Beazley, D. M. (2009). *Python essential reference.* Addison-Wesley Professional.

Ceder, N. (2018). *The quick Python book.* Simon and Schuster.

Downey, A. B. (2016). *Think Python. How to Think Like a Computer Scientist.* O'Reilly Media, 2nd edition.

Lubanovic, B. (2014). *Introducing Python: Modern Computing in Simple Packages.* O'Reilly Media, Inc.

Lutz, M. (2013). *Learning Python.* O'Reilly Media, 5th edition.

Python (2021). Python 3.10.0 documentation. `https://docs.python.org/3/` [Accessed 17 October 2021].