Computational Aspects in Econometrics - Python II module

Pieter Kleer

Table of contents

Chapter 1

About

1.1 Welcome

Welcome to the online "book" for the Python II module of Computational Aspects in Econometrics. We will follow the content in this book during the lectures and it is the basis of the material that will appear on the exam, so you should read through this book carefully. Because this book is new, it is likely that we will make some edits throughout the course.

Please not that this is an advanced Python module. We assume familiarity with the basics of programming in Python. For students enrolled in the bachelor Econometrics and Operations Research, the topics of the course Programming for EOR is a good example of what I expect you to be familiar with. An online book covering most of these topis can be found here. This book also contains some topics not covered in Programming for EOR.

1.2 Goal

The goal of this module is to teach you the basics of scientific computing with Python. Here you should think mostly of implementing algorithmic tasks that you encounter during your Econometrics and Operations Research courses, such as, linear algebra, optimization, statistics and machine learning. We hope that the skills you are taught here can be useful for, e.g., numerical work in your bachelor or (perhaps later) master thesis. Furthermore, many companies nowadays program in Python, so the topics of this module can also be useful in your professional career later in life.

Next to teaching you how to implement certain mathematical task in a correct manner in Python, we also put emphasis on good coding practices. Especially if you start to write scripts with hundres of lines of code, it is important that you

learn how to do this in a structured fasion using efficient Python functionality. Good coding practices are the topic of the next chapter. The idea is that you use these practices when doing the exercises corresponding to every lecture, as well as the group assignment.

1.3 Use of Spyder

This course document is based on the use of Spyder as integrated development environment (IDE) for creating Python code, i.e., the program that the code is written in. You can also use VS code or any other IDE to do the exercises and/or assignment in. Whenever this book contains screenshots to illustrate something, they will have been taken in Spyder.

To install the Anaconda distribution containing Spyder, follow the steps here. This is the installation that was alos recommended in the Python I module.

Chapter 2

Good coding practices

In this chapter we will discuss the good coding practices that have to applied when writing Python code: efficient computations, no hard coding, don't repeat yourself (DRY), single responsibility, and documentation. Overall, always try to keep your code simple and structured.

2.1 Efficient computations

Perhaps the most important topic of this course is that of efficient computation, i.e., to make efficient use of the mathematical functionality that Python has to offer, in particular the functionality of the NumPy package.

The way to think of this is as follows: Many of the mathematical tasks and exercises that we will see could, in theory, be solved using for- and/or while-loops, as well as if/else-statements. However, there are often more efficient functions programmed in the NumPy package in Python that can carry out these task in a quicker way using less code!

Let us look at an example. Suppose I am given a vector $x = [x_1, \dots, x_n] \in \mathbb{R}^n$, and I want to compute the L^2-norm

$$||x||_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$$

of this vector. One way to solve this directly would be by using a for-loop to compute the sum inside the square root, and then take the square root of this number. We will illustrate this next for the vector $x = [1, 2, 3, \dots, 300.000]$.

```
# Length of vector
n = 300000
```

```
# Compute inner summation of ||x||_2
inner_sum = 0
for i in range(1,n+1):
    inner_sum = inner_sum + i**2

# Take square root
two_norm = (inner_sum)**(0.5)

# Print value of two-norm
print(two_norm)
```

```
94868566.97584295
```

Another way to do this is to define the vector x efficiently making use of the `arange()` function in Numpy, followed by the `linalg.norm()` function that can compute the L^2-norm for us directly. We will see these functions in more detail in a subsequent chapter.

It is standard convention to import the Numpy package under the alias `np`.

```
#Import Numpy package under the alias np
import numpy as np

# Length of vector
n = 300000

# Define vector x as array using arange() function
x = np.arange(1,n+1)

# Compute two-norm with built-in function
two_norm = np.linalg.norm(x)

# Print value of two-norm
print(two_norm)
```

```
94868566.97584295
```

This is a much cleaner, and in fact faster, way to compute the L^2-norm of the vector x. Especially for large values of n, the second piece of code that exploits the Numpy functionality can be much faster! The Python code below (which we will not discuss) illustrates this comparison by timing how long both approaches require to compute the norm for $n = 30.000.000$ (i.e., thirty million). If you run the code below on your own device, you might get different outputs, but, in general, there should be a significant (multiplicative) difference in execution time.

```
import numpy as np
import time
```

```
## Computing norm with for-loop

# We determine the time it takes to input the code between start and end
start = time.time()

# Length of vector
n = 30000000

# Compute inner summation of ||x||_2
inner_sum = 0
for i in range(1,n+1):
    inner_sum = inner_sum + i**2

# Take square root
two_norm = (inner_sum)**(0.5)

end = time.time()

# Time difference
loop_time = end - start
print('%.10f seconds needed for computing norm' % (loop_time),
      'with for-loop')

## Computing norm with Numpy functions

#We determine the time it takes to input the code between start and end
start = time.time()

# Length of vector
n = 30000000

# Define vector x as array using arange() function
x = np.arange(1,n+1)

# Compute two-norm with built-in function
two_norm = np.linalg.norm(x)

end = time.time()

# Time difference
numpy_time = end - start
print('%.10f seconds needed for computing norm' % (numpy_time),
      'with Numpy functions')


#This shows how many times faster Numpy approach is faster than for-loop solution
```

```
if numpy_time != 0:
    (print('NumPy\'s approach is %i times more efficient than for-loop approach.'
           % ((loop_time)/(numpy_time))))
```

```
9.8354628086 seconds needed for computing norm with for-loop
0.1126644611 seconds needed for computing norm with Numpy functions
NumPy's approach is 87 times more efficient than for-loop approach.
```

One important take-away of the above comparison is the following.

> When performing mathematical tasks, avoid the use of for- and while-loops, as well as if/else-statements, by efficient use of Python functionality.

2.2 No hard coding

Suppose we are given the function $f(x) = a \cdot x^2 + a \cdot b \cdot x - a \cdot b \cdot c$ and the goal is to compute $f(10)$ for $a = 3, b = 4$ and $c = -10$. One way of doing this would be to plug in all the variables and return the resulting function value.

```
# Hard coding
print(3*(10**2) + 3*4*10 + 4*-2*3)
```

```
396
```

However, this is inefficient for the following reason: If we would want to change the number $x = 10$ to, e.g., $x = 20$, we would have to twice replace 10 by 20. The same is true for the variable a, which appears in even three places. In general, in large scripts, variables can appear in more than a hundred places. To overcome this inefficiency issue, it is always better to separate the input data (the x, a, b and c in this case) from the function execution

```
# No hard coding
def f(x, a, b, c):
    return a*x**2 + a*b*x + b*c*a

x = 10
a, b, c = 3, 4, -2
print(f(x, a, b, c))
```

```
396
```

The take-away of the above comparison is the following.

> Separate input data from the execution of functions and commands.

2.3 Don't repeat yourself (DRY)

If you have to carry out a piece of code for multiple sets of input data, always avoid copy-pasting the code. As a first step, try to to the repeated execution using a for-loop. For example, suppose we want to print the function values $f(x, 3, 4, -2)$ for $x = 1, 2, 3, 4$, with f as in the previous section. We can do this as follows.

```
def f(x, a, b, c):
    return a*x**2 + a*b*x + b*c*a

a, b, c = 3, 4, -2

# With repetition
x = 1
print('%.2f' % f(x,a,b,c))
x = 2
print('%.2f' % f(x,a,b,c))
x = 3
print('%.2f' % f(x,a,b,c))
x = 4
print('%.2f' % f(x,a,b,c))
```

```
-9.00
12.00
39.00
72.00
```

A more efficient way of typing this, is by using a for-loop that repeatedly executes the print statement.

```
import numpy as np

a, b, c = 3, 4, -2
x = [1,2,3,4]
# Determine all function values with list comprehension
y = [f(i,a,b,c) for i in x]

# Print values in y (could also print the whole vector y right away)
for j in y:
    print(j)
```

```
-9
12
39
72
```

The take-away of the above comparison is the following.

> Don't carry out the same task on different inputs twice by copy-pasting code, but use, e.g., a for-loop in which you iterate over the inputs.

In fact, later on in the course we will explain the concept of vectorizing a function, to make sure it can handle multiple input simultaneously. This is another way to avoid unnecessary repetition and in fact also the use of loops.

2.4 Single responsibility

When you write larger pieces of codes, it is often useful to split it up in smaller parts that all serve their own purpose. Suppose we want to implement Newton's method for finding a root x of a function $f : \mathbb{R} \to \mathbb{R}$, i.e., a point x that satisfies $f(x) = 0$.

Newton's methods starts with a (suitably chosen) initial guess $x_0 \in \mathbb{R}$ and repeatedly computes better approximations using the recursive formula

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

The goal is to implement this formula for the function $f(x) = (x-1)^2 - 1$, whose derivative is $f'(x) = 2(x-1)$. The roots of this function are $x = 0$ and $x = 2$.

```
# We implement the function y = x - f(x)/f'(x)
def y(x):
    return x - ((x-1)**2 - 1)/(2*(x-1))

x0 = 3
x1 = y(x0)

print(x1)
```

```
2.25
```

A clearer way to implement the formula $x_{i+1} = x_i - f(x_i)/f'(x_i)$ is to define separate functions for the evaluation of the functions f and f', and then combine these to create the recursive formula. In this way, we get three functions that are each responsible for one aspect of Newton's formula, namely implementing the function f, implementing the function f' and computing the recursive formula. This also makes it easier for a user of the code to understand what is happening.

```
# Single responsibility version

# Define f
def f(x):
    return (x-1)**2 - 1
```

```
# Define f'
def fprime(x):
    return 2*(x-1)

# Implement function y = x - f(x)/f'(x)
def y(x):
    return x - f(x)/fprime(x)

x0 = 3
x1 = y(x0)

print(x1)
```

```
2.25
```

The take-away of the above comparison is the following.

> If you are implementing a complex mathematical task involving multiple aspects, try to separate these aspects in different functions.

2.5 Documentation

The final good coding practice that we discuss is documentation. Ideally, you should always explain inside a function what the inputs and outputs are, and for larger scripts it is good to indicate what every part of the script does. For smaller functions and scripts this is not always necessary. We will next give an example for Newton's method as in the previous section

```
## Function that implements Newton's method
def newton(f,fprime,x0,iters):
    """
    This function implements Newton's iterative method
    x_{i+1} = x_i - f(x_i)/f'(x_i) for finding root of f.

    Input parameters
    ----------
    f : Function f
    fprime : Derivative of the function f
    x0 : Initial estimate for root x of f.
    iters : Number of iterations that we run Newton's method.

    Returns
    -------
    Approximation for x satisfying f(x) = 0.
    """

    # Initial guess
```

```
    x = x0

    # Repeatedly compute the recursive formula
    # by overwriting x for 'iters' iterations
    for i in range(iters):
        x_new = x - f(x)/fprime(x)
        x = x_new
    return x

## An example of Newton's method as implemented above

# Define f
def f(x):
    return (x-1)**2 - 1

# Define f'
def fprime(x):
    return 2*(x-1)

# Define intial guess and number of iterations
x0 = 10
iters = 6

# Run Newton's method
root = newton(f,fprime,x0,iters)

# Print output and explain what has been computed
print('Running Newton\'s method for %.i iterations with initial' % iters,
       'estimate %.2f' % x0,'\n','gives (estimated) root x = %.7f' % root,
       'with f(x) = %.7f' % f(root))
```

```
Running Newton's method for 6 iterations with initial estimate 10.00
 gives (estimated) root x = 2.0000013 with f(x) = 0.0000025
```

The following is a set of guidelines regarding how to add documentation to a function.

> Try to adhere to the following documention rules when writing complex functions: 1. Function documentation between triple double-quote characters. 2. Clearly describe what a function does and what its input and output arguments are. 3. Choose descriptive variable names, lines not longer than 80 characters. 4. Don't add comments for every line. Add comments for main ideas and complex parts. 5. A comment should not repeat the code as text (e.g. "time = time + 1 # increase time by one).

Chapter 3

NumPy arrays

3.1 Introduction

The NumPy package (module) is used in almost all numerical computations using Python. It is a package that provides high-performance vector, matrix and higher-dimensional data structures for Python. High-performance here refers to the fact that Python can perform computations on such data structures very quickly if appropriate functions are used for this.

To use NumPy you need to import the `numpy` module. This is typically done under the alias `np` so that you don't have to type `numpy` all the time when using a function from the module.

```
import numpy as np
```

We emphasize at this point that there is often not a unique way or command to achieve a certain outcome. When doing the exercises corresponding to the theory given in this chapter, it is, however, recommended to find a solution using the presented functionality.

3.2 Creating arrays

In the NumPy package the data type used for vectors, matrices and higher-dimensional data sets is an array. There are a number of ways to initialize new arrays, for example from

- a Python list or tuples;
- using functions that are dedicated to generating numpy arrays, such as `arange()` and `linspace()` (we will see those later);
- reading data from files.

3.2.1 Lists

For example, to create new vector and matrix arrays from Python lists we can use the `numpy.array()` function. Since we imported NumPy under the alias `np`, we use `np.array()` for this.

To create a vector, the argument to the array function is a Python list

```
v = np.array([1,2,3,4]) #Array creation from list [1,2,3,4]
print(v)
```

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

To create a matrix, the argument to the array function is a nested Python list. Every element of the outer list is a list corresponding to a row of the matrix. For example, the matrix

$$M = \begin{bmatrix} 1 & 2 & 7 \\ 3 & -4 & 4 \end{bmatrix}$$

is created as follows.

```
M = np.array([[1, 2, 7], [3, -4, 4]])
print(M)
```

```
[[ 1  2  7]
 [ 3 -4  4]]
```

You can access the shape (number of rows and columns) , size (number of elements) and number of dimensions (number of axes in matrix) of the array with the `shape`, `size` and `ndim` attributes, respectively. Note that the size is simply the product of the numbers in the shape tuple, and the number of dimensions is the size of the shape tuple.

```
# Shape of matrix M
shape_M = M.shape  #np.shape(M) also works
print(shape_M)
```

```
(2, 3)
```

```
# Size of matrix M
size_M = M.size  #np.size(M) also works
print(size_M)
```

```
6
```

```
# Number of diemenions
ndim_M = M.ndim  #np.ndim(M) also works
print(ndim_M)
```

```
2
```

NumPy arrays are of the type `ndarray` (short for n-dimensional array). You can access this type through the `type()` function.

```
# Type of matrix M
type_M = type(M)
print(type_M)
```

```
<class 'numpy.ndarray'>
```

So far a NumPy array looks awfully much like a Python list (or nested list). Why not simply use Python lists for computations instead of creating a new array type?

There are several reasons:

- Python lists are very general. They can contain any kind of object. They are dynamically typed. They do not support mathematical functions such as matrix and dot multiplications.
- Numpy arrays are statically typed and homogeneous. The type of the elements is determined when the array is created.
- Numpy arrays are memory efficient.
- Because of the static typing, fast implementation of mathematical functions such as multiplication and addition of numpy arrays can be implemented in a compiled language (C and Fortran are used).

Using the `dtype` (data type) attribute of an array, we can see what type the data inside an array has.

```
# Data type of elements in array
dtype_M = M.dtype
print(dtype_M)
```

```
int32
```

If we want, we can explicitly define the type of the array data when we create it, using the `dtype` keyword argument:

```
# Define data as integers
M = np.array([[1, 2], [3, 4]], dtype=int)
print('M = \n', M)
```

```
M =
 [[1 2]
 [3 4]]
```

```
# Define data as floats
N = np.array([[1, 2], [3, 4]], dtype=float)
print('N = \n', N)
```

```
N =
 [[1. 2.]
 [3. 4.]]
```

```
# Define data as complex floats
O = np.array([[1, 2], [3, 4]], dtype=complex)
```

```
print('O = \n', O)
```

```
O =
 [[1.+0.j 2.+0.j]
 [3.+0.j 4.+0.j]]
```

Common data types that can be used with dtype are: `int`, `float`, `complex`, `bool`, `object`, etc.

We can also explicitly define the bit size of the data types, such as: `int64`, `int16`, `float128`, `complex128`. For example, `int64` allows us to define an integer variable in the range [−264,...,264].

You can also change the data type of the elements using the `astype()` method.

```
M = np.array([[1,2], [3,4]])
print(M.dtype)
```

```
int32
```

```
# Define M_float as matrix whose elements are those of
# the matrix M, but then as floats.
M_float = M.astype(float)
print(M_float)
```

```
[[1. 2.]
 [3. 4.]]
```

```
print(M_float.dtype)
```

```
float64
```

3.2.2 Arrays froms functions

There are various useful arrays that can be automatically created using functions from the NumPy package. These arrays are typically hard to implement directly as a list.

`arange(n)`: This function creates the array $[0, 1, 2, \ldots, n-1]$ whose elements range from 0 to $n-1$.

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

```
print(x)
```

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

If you want to explicitly define the data type of the elements, you can add the `dtype` keyword argument (the same applies for all functions that are given below).

```
n = 10
x = np.arange(n, dtype=float)

print(x)
```

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

`arange(a,b)`: This function creates the array $[a, a+1, a+2, \ldots, b-2, b-1]$.

```
a, b = 5,11
x = np.arange(a,b)

print(x)
```

```
[ 5  6  7  8  9 10]
```

`arange(a,b,step)`: This function creates the array $[a, a+step, a+2\cdot step, \ldots, b-2\cdot step, b-step]$. That is, the array ranges from a to b (but not including b itself), in steps of size `step`.

```
a, b, step = 5, 11, 0.3
x = np.arange(a,b,step)

print(x)
```

```
[ 5.   5.3  5.6  5.9  6.2  6.5  6.8  7.1  7.4  7.7  8.   8.3  8.6  8.9
  9.2  9.5  9.8 10.1 10.4 10.7]
```

`linspace(a,b,k)`: Create a discretization of the interval $[a, b]$ containing k evenly spaced points, including a and b as the first and last element of the array.

```
a,b,k = 5,10,20
x = np.linspace(a,b,k)

print(x)
```

```
[ 5.          5.26315789  5.52631579  5.78947368  6.05263158  6.31578947
  6.57894737  6.84210526  7.10526316  7.36842105  7.63157895  7.89473684
  8.15789474  8.42105263  8.68421053  8.94736842  9.21052632  9.47368421
  9.73684211 10.        ]
```

`diag(x)`: This function creates a matrix whose diagonal contains the list/vector/array `x`.

```
x = np.array([1,2,3])
D = np.diag(x)

print(D)
```

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

```
 [0 0 3]]
```

`np.zeros(n)`: This function create a vector of length n with zeros.

```
n = 5
x = np.zeros(n)

print(x)
```

```
[0. 0. 0. 0. 0.]
```

`np.zeros((m,n))`: This function create a matrix of size $m \times n$ with zeros. Note that we have to input the size of the matrix as a tuple `(m,n)`; using `np.zero(m,n)`

```
m, n = 2, 5
M = np.zeros((m,n))

print(M)
```

```
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

`np.ones(n)` and `np.ones((m,n))`: These functions create a vector of length n with ones, and a matrix of size $m \times n$ with ones, respectively.

```
m, n = 2, 5
x = np.ones(n)

print(x)
```

```
[1. 1. 1. 1. 1.]
```

```
M = np.ones((m,n))

print(M)
```

```
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

`mgrid[a:b,c:d]`: This function can be used to generate a "grid" of integer points in $[a, b-1] \times [c, d-1]$ for $a, b, c, d, \in \mathbb{N}$.

In the execution below, the idea is that the function generates two matrices `x` and `y`, so that points $(x[i][j], y[i][j])$ correponds to a grid point $(a+i, c+j)$ for $i = 0, \ldots, a-(b-1)$ and $j = 0, \ldots, c-(d-1)$. More generally, for a given grid point $(a+i, c+j)$, the matrix `x` stores the first coordinate $a+i$, and `y` the second coordinate $c+j$.

```
a,b = 0,5
c,d = 2,6
x, y = np.mgrid[a:b, c:d]
```

```
print(x)
```

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

```
print(y)
```

```
[[2 3 4 5]
 [2 3 4 5]
 [2 3 4 5]
 [2 3 4 5]
 [2 3 4 5]]
```

3.2.3 Reading data from files

The third option, which you might use most often in a professional context, is to read in data from a file directly into a NumPy array. You can do this using the `loadtxt()` function.

If you want to try this yourself, download the file numerical_data.dat here and store it in the same folder as where you are storing the Python script in which you execute the code snippet below.

```
# Load data into NumPy array
data_dat = np.loadtxt('numerical_data.dat')

# Print the data
print(data_dat)
```

```
[[ 1.   5.   4.  -9.   1. ]
 [ 3.   5.   6.   7.   7. ]
 [ 4.   3.   2.   1.   0.5]]
```

Python puts every row in the data (DAT) file into a separate row of the NumPy array; note that the numbers in the data file are separated by a whitespace character.

We can also save data from a Numpy array into a DAT-file using the `savetxt()` function. The first argument of this function is the name of the file in which you want to store the array, and the second argument is the array to be stored.

```
# Matrix M
M = np.array([[1,2,3],[5,6,7],[10,11,12],[14,15,16]])

# Save matrix to DAT file
np.savetxt('matrix.dat', M)
```

This should have created the file matrix.dat in the same folder as where you stored the Python script that ran the code above. You might notice that the numbers are stored using the scientific notation. For example, the number 1 appears as `1.00000000000000000e+00` in the CSV-file.

You can suprress this behaviour by explicitly specifying the data type in which you want the numbers in the matrix to be stored using the `fmt` keyword argument. For example, `fmt = '%i'` stores the numbers as integers.

```
# Matrix M
M = np.array([[1,2,3],[5,6,7],[10,11,12],[14,15,16]])

# Save matrix to DAT file
np.savetxt('matrix_int.dat', M, fmt='%i')
```

This should have created the file matrix_int.dat in the same folder as where you stored the Python script that ran the code above.

3.3 Accessing

In this section we will describe how you can access, or index, the data in a NumPy array.
We can index elements in an array using square brackets and indices, just like as with lists. In NumPy indexing starts at 0, just like with a Python list.

```
v = np.array([12,4,1,9])

# Element in position 0
print(v[0])

# Element in position 2
print(v[2])

# Element in position -1 (last element)
print(v[-1]) # Same as v[3]

# Element in position -3 (counted backwards)
print(v[-3]) # Same as v[1]
```

```
12
1
9
4
```

3.3.1 Basic indexing

If you want to access the element at position (i, j) from a two-dimensional array, you can use the double bracket notation `[i][j]`, but with arrays you can also

use the more compact syntax `[i,j]`.

```
M = np.array([[10,2,6,7], [-15,6,7,-8], [9,10,11,12],[3,10,6,1]])

# Element at position (1,1)
print('List syntax:',M[1][1])

# Element at position (1,1)
print('Array syntax', M[1,1])
```

```
List syntax: 6
Array syntax 6
```

If you want to access row i you can use `M[i]` or `M[i,:]`.

```
print(M[2]) # Gives last row

print(M[2,:]) # Gives last row
```

```
[ 9 10 11 12]
[ 9 10 11 12]
```

If you want to access column j you can use `M[:,j]`. Both here and in the previous command, the colon `:` is used to indicate that we want all the elements in the respective dimension. So `M[:,j]` should be interpreted as, we want the elements from all rows in the j-th column.

3.3.2 Index slicing

Index slicing is the technical name for the index syntax that returns a slice, a consecutive part of an array.

```
v = np.array([12,4,1,9,11,14,17,98])

print(v)
```

```
[12  4  1  9 11 14 17 98]
```

`v[lower:upper]`: This return the elements in `v` at positions `lower, lower+1,...,upper-1`. Note that the element at position `upper` is not included.

```
# Returns v[1], v[2], v[3], v[4], v[5]
print(v[1:6])
```

```
[ 4  1  9 11 14]
```

You can also omit the `lower` or `upper` value, in which case it is set to be position 0 or the last position -1, respectively.

```
# Returns v[3],...,v[8]
print(v[3:])
```

```
# Returns v[0],...,v[4]
print(v[:5])
```

```
[ 9 11 14 17 98]
[12  4  1  9 11]
```

`v[lower:upper:step]`: This returns the elements in `v` at position `lower,lower+step,lower+2*step,...(upper-1)-step, (upper-1)`. It does the same as `[lower:upper]`, but now in steps of size `step`.

```
v = np.array([12,4,1,9,11,14,17,98])

# Returns v[1], v[3], v[5]
print(v[1:6:2])
```

```
[ 4  9 14]
```

You can omit any of the three parameters `lower`,`upper` and `step`

```
# lower, upper, step all take the default values
print(v[::])

# Index in step is 2 with lower and upper defaults
print(v[::2])

# Index in steps of size 2 starting at position 3
print(v[3::2])
```

```
[12  4  1  9 11 14 17 98]
[12  1 11 17]
[ 9 14 98]
```

You can also use slicing with negative index values.

```
# The last three elements of v
print(v[-3:])
```

```
[14 17 98]
```

Furthermore, the same principles apply to two-dimensional arrays, where you can specify the desired indices for both dimensions

```
M = np.array([[10,2,6,7], [-15,6,7,-8], [9,10,11,12],[3,10,6,1]])

print(M)
```

```
[[ 10   2   6   7]
 [-15   6   7  -8]
 [  9  10  11  12]
 [  3  10   6   1]]
```

`[a:b, c:d]`: This returns the submatrix consisting of the rows `a,a+1,...,b-1` and rows `c,c+1,...,d`. You can also combine this with a step argument, i.e., use `[a:b:step1, c:d:step2]`.

```
# Returns elements in submatrix formed by rows 2,3 (excluding 4)
# and columns 1,2 (excluding 3)
print(M[2:4,1:3])
```

```
[[10 11]
 [10  6]]
```

If you want to obtain a submatrix whose rows and/or columns do not form a consecutive range, or if you want to specify these list manually, you can use the `ix_()` function from NumPy. Its arguments should be a list of row indices, and a list of column indices specifying the indices of the desired submatrix.

```
i = [0,2,3]
j = [0,3]

# Returns submatrix formed by rows 0,2,3 and columns 0,3
print(M[np.ix_(i,j)])
```

```
[[10  7]
 [ 9 12]
 [ 3  1]]
```

3.3.3 Fancy indexing

Fancy indexing is the name for when an array or list is used instead of indices, to access part of an array. For example, if you want to access elements in the locations $(0,3),(1,2)$ and $(1,3)$, you can define a list of row indices `[0,1,1]` and columns indices `[3,2,3]` and access the matrix with these lists.

```
i = [0,1,1]
j = [3,2,3]

# Returns M[0,3] = 7, M[1,2] = 7, M[1,3] = -8
print(M[i,j])
```

```
[ 7  7 -8]
```

Another way of fance indexing is by using a Boolean list, that indicates for every element whether it should be index (True) or not (False). Such a list is sometimes called a mask.

```
v = np.array([1,6,2,3,9,3,6])

# Tell for every element whether is should be index
mask = [False, True, True, True, False, True, False]
```

```
print(v[mask])
```

```
[6 2 3 3]
```

Typically, the mask is generated from a Boolean statement. For example, suppose we want to select all elements strictly smaller than 3 and greater or equal than 7 from the array `v`.

The following statements achieve this. Recall that you can use `&` if you want the first AND the second statement to be satisfied, and `|` if either the first OR the second has to be satisfied (or both).

```
mask_37 = (v < 3) | (v >= 7)

# Boolean vector indiciating for ever element in v
# whether the conditions v < 3 and v >= 7 are satisfied
print(mask_37)
```

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

We can now access the elements satisfying these conditions by indexing `v` with this mask

```
print(v[mask_37])
```

```
[1 2 9]
```

3.4 Modifying

3.4.1 Elements, rows or columns

Using similar ways of indexing as in the previous section, we can also modify the elements of an array

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

print(M)
```

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

```
# Modify individual element
M[0,1] = -1

print(M)
```

```
[[ 1 -1  1  1]
 [ 2  2  2  2]
```

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

```
# Modify (part of a) row
M[1,[1,2,3]] = [-2,-2,-2]

print(M)
```

```
[[ 1 -1  1  1]
 [ 2 -2 -2 -2]
 [ 3  3  3  3]
 [ 4  4  4  4]]
```

```
# Modify third column to ones
M[:,3] = np.ones(4)

print(M)
```

```
[[ 1 -1  1  1]
 [ 2 -2 -2  1]
 [ 3  3  3  1]
 [ 4  4  4  1]]
```

3.4.2 Broadcasting

There does not necessarily have to be a match between the part of the matrix that we index, and the dimensions of the data that we want to overwrite that part with.

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

print(M)
```

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

For example, in order to replace the third column of M by ones, we can also do the command below, instead of using `np.ones(4)`.

```
# Modify third column to ones
M[:,3] = 1

print(M)
```

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

Although there is a mismatch between the indexed part on the left (a column) and the data on the right (single number), Python broadcasts the data to an appopriate format by copying it to the correct size. That is, it copies the 1 to an array `[1,1,1,1]` of ones, which it then places in the third column.

This works similar in higher dimensions. Suppose we want to overwrite the second and third row with `[1,6,2,3]`. Then the indexed part is a 2×4 array, but the data a 1×4 array.

```
# Modify second and third row
M[2:4,:] = [1,6,2,3]

print(M)
```

```
[[1 1 1 1]
 [2 2 2 1]
 [1 6 2 3]
 [1 6 2 3]]
```

Python here first copies the data to `[[1,6,2,3],[1,6,2,3]]` and then modifies M with this array.

3.4.3 Transpose

Another useful function, in the context of linear algebra, is to take the transpose of a two-dimensional array `M`, which modifies the entries along the diagonal.

```
M = np.array([[1,2,3],[3,4,-1]])

print(M)
```

```
[[ 1  2  3]
 [ 3  4 -1]]
```

```
transpose_M = M.T #np.transpose(M) also works
print(transpose_M)
```

```
[[ 1  3]
 [ 2  4]
 [ 3 -1]]
```

3.5 Repeating and stacking

We can also use existing matrices and build new ones from it by stacking them either horizontally or vertically.

`tile(M,(k,r))`: This function takes an array `M` and copies it k times vertically and r times horizontally, resulting in a "tiling" of the original array `M`.

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

M_tile = np.tile(M,(2,3))
print(M_tile)
```

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

If you do not input a tuples with two arguments, but only a number, then `tile()` does the tiling only horizontally.

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

M_tile = np.tile(M,4)
print(M_tile)
```

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

`repeat(M,k)`: This function takes every element of `M`, repeats it k times, and puts all these numbers in a one-dimension array.

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

M_repeat = np.tile(M,3)
print(M_repeat)
```

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

`vstack(a,b)`: This stacks two arrays `a` and `b` vertically, provided they have the correct dimensions to do this.

```
a = np.array([7,8])
M = np.array([[1,2],[3,4]])

M_a = np.vstack((M,a))
print(M_a)
```

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

`hstack(a,b)`: This stacks two arrays `a` and `b` horizontally, provided they have the correct dimensions to do this.

Note that in the example below we define `a` as a 1×2 array, i.e., a column array, to make sure we can stack it right of `M`. If we would have kept `a = np.array([7,8])` then Python will give an error, because it cannot stack a row vector next to a two-dimensional array.

```
a = np.array([[7],[8]])
M = np.array([[1,2],[3,4]])

M_a = np.hstack((M,a))
print(M_a)
```

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

3.6 Reshaping

It is possible to adjust the shape of an array, while keeping the data of the array the same. For example, consider the $x = [1, 2, 3, \dots, 12]$.

```
x = np.arange(1,13)

print(x)
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12]
```

We can reshape it into the 3×4 matrix

$$M = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 11 & 10 & 11 & 12 \end{bmatrix}$$

by using the `reshape(a,b)` method. It reshapes x to an $a \times b$ array provided that $a \cdot b$ equal the size (i.e., number of elements) of x.

```
# Reshape x to a 3-by-4 matrix
M = x.reshape(3,4)

print(M)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

We can also reshape two-dimensional arrays, for example, we can reshape M again to a 2×6 matrix.

```
# Reshape M to a 2-by-6 matrix
N = M.reshape(2,6)

print(N)
```

```
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]]
```

You should observe that Python does the reshaping in a very specific way: When we transform x to M above, Python fills the matrix M in a row-by-row fashion

(instead of column-by-column). This is because of what is called the largest (axis) index changest fastest principle.

To understand this idea, recall that we can access the element at position (i, j) of a matrix M with `M[i,j]`. Here i is the row-index at position 0 of the index list `[i,j]`, and j is the column index at position 1 of the index list `[i,j]`. We said that the row indices form the 0-axis of the matrix, and the column indices the 1-axis.

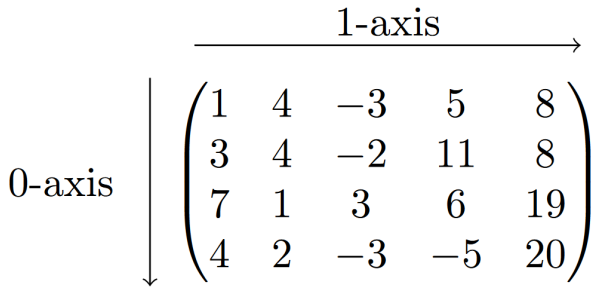

Figure 3.1: Axes of a two-dimensional array

Largest (axis) index changing fastest means that an $m \times n$ matrix gets filled first along the 1-axis, i.e., it fills the positions $(0, 0), (0, 1), ..., (0, n)$ while keeping the row index 0 fixed. It then moves up one row index, i.e., one position along the 0-axis and fills the elements $(1, 0), (1, 1), ..., (1, n)$, i.e., the elements along the 1-axis. It continues in this fashion until the complete matrix is full.

Another convenient method for reshaping is `flatten()`, which turns a matrix of any size into a one-dimensional array.

```
# Define 2-by-3 matrix
M = np.array([[9,1,3],[2,4,3]])

# Turn into one-dimensional array
x = M.flatten()
print(x)
```

```
[9 1 3 2 4 3]
```

If you want to turn a one-dimensional array $x = [x_0, \dots, x_{n-1}]$ into a column array of shape $(n, 1)$, you can do this as follows.

```
x = np.array([1,2,4,3,8])
n = np.size(x)

x = x.reshape(n,1)
print(x)
```

```
[[1]
 [2]
 [4]
 [3]
 [8]]
```

A more direct way of doing this, is by using `x[:,None]`.

```
x = np.array([1,2,4,3,8])
x = x[:,None] # Turns x into column array of shape (n,1)

print(x)
```

```
[[1]
 [2]
 [4]
 [3]
 [8]]
```

3.7 Copy vs. view

In the last sections we have seen various ways of using arrays to create other arrays. One point of caution here is whether or not the new array is a view or a copy of the original array.

3.7.1 View

A view y of an array x is another array that simply displays the elements of the array x in a different array, but the elements will always be the same. This means that if we would change an element in the array x, the same element will change in y and vice versa.

```
x = np.array([[4,2,6],[7,11,0]])
y = x # This create a view of x

print('y = \n', y)
```

```
y =
 [[ 4  2  6]
 [ 7 11  0]]
```

We next change an element in x. Note that the same element changes in y.

```
# Change element in x
x[0,2] = -30

# y now also changes in that position
print('y = \n',y)
```

```
y =
 [[  4   2 -30]
 [  7  11   0]]
```

The same happens the other way around: If we change an element in y, then the corresponding element in x also changes.

```
# Change element in y
y[1,1] = 100

# x now also changes in that position
print('x = \n', x)
```

```
x =
 [[  4   2 -30]
 [  7 100   0]]
```

Note that the same behaviour occurs in we apply the `reshape()` method.

```
# Define x = [1,2,...,12]
x = np.arange(1,13)

# Reshape x to a 3-by-4 matrix
M = x.reshape(3,4) # Creates view of x

print(M)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

If we now change an element in M, then the corresponding element changes in x. This mean that M is a view of the original array x.

```
# Change element in M
M[1,3] = 50

# x now also changes in that position
print(x)
```

```
[ 1  2  3  4  5  6  7 50  9 10 11 12]
```

3.7.2 Copy

A copy of an array x is an array z that is completely new and independent of x, meaning that if we change an element in x, then the corresponding element in z does not change, and vice versa. To obtain a copy of x, we can simply apply the `copy()` method to it.

```
# Define x = [1,2,...,12]
x = np.arange(1,13)
```

```
z = x.copy() # Create copy of x
z[0] = -10 # Change element of z

print('z = \n', z)
print('x = \n', x) # x has not changed
```

```
z =
 [-10   2   3   4   5   6   7   8   9  10  11  12]
x =
 [ 1  2  3  4  5  6  7  8  9 10 11 12]
```

Note that in the above example, x remains unchanged when we modify the element of z at position 0.

Similarly, to turn a reshaped array into a copy, we can apply the `copy()` method to it.

```
# Define x = [1,2,...,12]
x = np.arange(1,13)

# Reshape x to a 3-by-4 matrix
M = x.reshape(3,4).copy() # Create copy
M[0,0] = -10 # Change element of x

print('M = \n', M)
print('x = \n', x) # x has not changed
```

```
M =
 [[-10   2   3   4]
 [  5   6   7   8]
 [  9  10  11  12]]
x =
 [ 1  2  3  4  5  6  7  8  9 10 11 12]
```

The `flatten()` method actually directly creates a copy of the original array.

```
# Define 2-by-3 matrix
M = np.array([[9,1,3],[2,4,3]])

# Turn into one-dimensional array
x = M.flatten() # Creates copy of M
x[0] = 100 # Change element in x

print('x = \n', x)
print('M = \n', M) # M has not changed
```

```
x =
 [100   1   3   2   4   3]
M =
```

```
[[9 1 3]
[2 4 3]]
```

It is important to know whether a Python function or command creates a copy or a view of the original array. You can typically look this up in the documentation of Python. Otherwise, experiment with the function or command to be sure how it behaves.

Chapter 4

Vectorization

In this chapter we will explore the power of NumPy arrays by studying the concept of vectorization. Let us first import the NumPy package.

```
import numpy as np
```

The idea of vectorization is that functions are designed so that they can efficiently handle multiple inputs simultaneously.

As an example, let's implement the Heavyside function H, which is defined by

$$H(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}.$$

That is, the function returns 1 if x is nonnegative, and 0 otherwise. If we are only interested in computing $H(x)$ for a single value x, then the following function suffices.

```
# Heavyside function
def Heavyside(x):
    if x >= 0:
        return 1
    else:
        return 0

print('H(2) =', Heavyside(2))
print('H(-3) =', Heavyside(-3))
```

```
H(2) = 1
H(-3) = 0
```

Suppose now that we would want to compute the value $H(x)$ for many values given in an array $x = [x_0, \ldots, x_{n-1}]$; this you would need to do, e.g., if you want

to visualize a function. One way to do this is to use a for-loop and append the value of $H(x_i)$ to an (initially empty) list in iteration i.

```
n = 8
x = np.arange(-n,n)

h_values = []
for i in x:
    h_values.append(Heavyside(i)) # append H(i) to list h_values

print(h_values)
```

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

As mentioned earlier, for-loops are typically very time-inefficient and should be avoided whenever possible. It would be better if we could instead use another approach that can compute the function values more efficiently.

One approach to avoid the for-loop is with a Boolean statement. Recall from Chapter 2 that for an array x, the command `x >= 0` will return a Boolean array containing True at position i if $x_i \geq 0$, and False if not. This command is an example of a vectorized operation: Although, mathematically speaking, it is defined to compare a number x with 0, the command also works if x is an array, in which case each of its elements get compared to 0.

```
comparison = (x >= 0) # Compare each element in x with 0
print(comparison)
```

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

The statement `x >= 0` returns a Boolean array containing True and False, but the Heavyside function should ouput 1 and 0, respectively. To achieve this, we can convert the Boolean array to an array with ones and zeros.

This can be done with the `astype()` method, or by a clever multiplication: In Python, multiplying True with 1 gives 1, and False with 1 gives 0. Note that these are also examples of vectorized operations. For example, in the latter case, we multiply an array with one number, which Python executes by multiplying every element in the array with that number.

```
# Convert Boolean values to integers (True becomes 1, False becomes 0)
H = comparison.astype('int')
print(H)

# Multiply Boolean array with 1 (True*1 = 1, and False*1 = 0)
H2 = comparison*1
print(H2)
```

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

This means we can define the vectorized Heavyside function as follows, using the "multiplication with one" approach. We actually need less code for the vectorized version than in the original approach.

```
# Vectorized Heavyside function
def Heavyside(x):
    return (x >= 0)*1

print(x)
print(Heavyside(x))
```

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

> A vectorized function can handle higher-dimensional inputs. For example, a function that is (mathematically speaking) defined for a single number can also handle one- or two-dimensional arrays as input, in which case the function computes the function value for every number in the array. Or a function that is defined for a one-dimensional array, can also handle two-dimensional arrays as input, in which case it computes the function value for every (one-dimensional) row (inner list) of the two-dimensional array.

We remark that vectorizing a function should not result in additional for-loops, but should exploit the functionality that Python, and in particular NumPy, have to offer. We will see many examples of this later in this chapter.

In this course, you will typically be told which types of input data your function should be able to handle, so, e.g., it could be that you have to write a function that performs mathematical operations on a one-dimensional array, but that it should be able to handle two-dimensional arrays as well (again, in which case your function should compute the function value for every row of that array).

As an example of what you should not do, the following definition of the Heavyside function is also able to handle "higher dimensional" inputs, but it does so by looping over the elements of the input, which makes it again a slow function. To summarize, you should not "hide" additional for-loops in the function itself.

```
# Heavyside function
def Heavyside(x):
    h = []
    for i in x:
        if i >= 0:
            h.append(0)
        else:
            h.append(1)
    return h

x = np.array([-2,-1,0,1,2])
```

```
print(Heavyside(x))
```

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

Also, it is not allowed to use functions like `vectorize()` from NumPy, because these are implemented essentially as a loop (as stated in the documentation of that function).

We continue with exploring vectorized functions within Python and NumPy.

4.1 Arithmetic operations

All basic arithmetic operations (addition, substraction, division, multiplication and comparison) are vectorized in Python. We will illustrate this with the addition operation `+`, but the same commands can be applied to the other arithmetic operations `-`, `/`, `*`, and `>=, ==, <=, !=`.

The addition operation `+` can be used to add two numbers together, as you well know. It can also add two arrays, i.e., it works as well for one- and two-dimensional arrays if they have the same shape. This is the usual addition operation you learn about when studying linear algebra.

```
x = np.array([1,4,7])
y = np.array([2,4,3])

print('x + y =\n',x+y)
```

```
x + y =
 [ 3  8 10]
```

```
A = np.array([[1,2],[1,4]])
N = np.array([[7,9],[3,4]])

print('A + N = \n',A+N)
```

```
A + N =
 [[ 8 11]
 [ 4  8]]
```

Python is also able to handle addition of arrays of different shapes in certain cases, using the concept of broadcasting that we have seen before. For example, we can add a single number to any array, in which case Python adds this number to every element in the array. This can be seen as an instance of vectorization.

```
c = 5

print('A + c =\n', A+c)
```

```
A + c =
 [[6 7]
```

```
 [6 9]]
```

We can also add either a one-dimensional array x of size n to an $m \times n$ matrix A. In this case, the array x gets added to every row of A:

$$A + x = \begin{bmatrix} a_{00} & a_{01} & \dots & a_{0(n-1)} \\ a_{10} & a_{11} & \dots & a_{1(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(m-1)0} & a_{(m-1)1} & \dots & a_{(m-1)(n-1)} \end{bmatrix} + \begin{bmatrix} x_0 & x_1 & \dots & x_{(n-1)} \end{bmatrix}$$
$$= \begin{bmatrix} a_{00} + x_0 & a_{01} + x_1 & \dots & a_{0(n-1)} + x_{n-1} \\ a_{10} + x_0 & a_{11} + x_1 & \dots & a_{1(n-1)} + x_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(m-1)0} + x_0 & a_{m1} + x_1 & \dots & a_{(m-1)(n-1)} + x_{n-1} \end{bmatrix}$$

In the example below, we have $m = 3$ and $n = 2$.

```
x = np.array([10,12])
print('Shape of x:', np.shape(x))

A = np.array([[1,2],[1,4],[3,1]])
print('Shape of A:', np.shape(A))

print('A + x = \n', A + x)
```

```
Shape of x: (2,)
Shape of A: (3, 2)
A + x =
 [[11 14]
 [11 16]
 [13 13]]
```

Again, this can be seen as an instance of vectorization, since Python automatically adds x to every row of the matrix A.

> Note that the shape of x is $(n,)$ which is the syntax that Python uses to denote that x only has one dimension. You can define $x = [x_0, \dots, x_{n-1}]$ explicitly as a row vector of shape $(1, n)$ by defining `x = np.array([[x_0,...,x_{n-1}]])`, that is, with double brackets. It is sometimes needed to change the shape of an array from $(n,)$ to $(1, n)$ or $(n, 1)$ to be able to use a function from NumPy.

Addition works in the same way if we define x explicitly as an array of shape $(1, n)$.

```
x = np.array([[10,12]])
print('Shape of x:', np.shape(x))
```

```
A = np.array([[1,2],[1,4],[3,1]])
print('Shape of A:', np.shape(A))

print('A + x =\n', A + x)
```

```
Shape of x: (1, 2)
Shape of A: (3, 2)
A + x =
 [[11 14]
 [11 16]
 [13 13]]
```

The same works if we define $x = [x_0, \dots, x_{m-1}]^T$ as a column array of shape $(m, 1)$, in which case it gets added to every column of the matrix A of shape (m, n):

$$A + x = \begin{bmatrix} a_{00} & a_{01} & \dots & a_{0(n-1)} \\ a_{10} & a_{11} & \dots & a_{1(n-1)} \\ \vdots & \ddots & \vdots & \\ a_{(m-1)0} & a_{(m-1)1} & \dots & a_{(m-1)(n-1)} \end{bmatrix} + \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{(m-1)} \end{bmatrix}$$

$$= \begin{bmatrix} a_{00} + x_0 & a_{01} + x_0 & \dots & a_{0(n-1)} + x_0 \\ a_{10} + x_1 & a_{11} + x_1 & \dots & a_{1(n-1)} + x_1 \\ \vdots & \ddots & \vdots & \\ a_{(m-1)0} + x_{m-1} & a_{(m-1)1} + x_{m-1} & \dots & a_{(m-1)(n-1)} + x_{m-1} \end{bmatrix}$$

```
x = np.array([[10],[12],[14]])
print('Shape of x:', np.shape(x))

A = np.array([[1,2],[1,4],[3,1]])
print('Shape of A:', np.shape(A))

print('A + x =\n', A + x)
```

```
Shape of x: (3, 1)
Shape of A: (3, 2)
A + x =
 [[11 12]
 [13 16]
 [17 15]]
```

We cannot add arrays of any dimensions to each other. For example, if we would try to add a a 2×2 array to a 4×2 array, then Python will return `ValueError: operands could not be broadcast together with shapes (4,2) (2,2)`, i.e., Python cannot perform this addition.

4.1.1 Multiplication broadcasting

We emphasize that the broadcasting concepts above also apply to the multiplication operator `*`. That is, if x is a column array then `A*x` multiplies every column of A in a pointwise fashion with the array x. Similarly, for two matrix A and B, the syntax `A*B` returns a matrix in which all elements of A and B are pointwise multiplied with each other, that is, entry (i, j) contains $a_{ij} \cdot b_{ij}$.

This is not the same as, e.g., the matrix-vector multiplication Ax in the linear algebra sense, i.e,

$$
\begin{aligned}
Ax &= \begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0(n-1)} \\ a_{10} & a_{11} & \cdots & a_{1(n-1)} \\ \vdots & \ddots & \vdots & \\ a_{(m-1)0} & a_{(m-1)1} & \cdots & a_{(m-1)(n-1)} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{(m-1)} \end{bmatrix} \\
&= \begin{bmatrix} a_{00}x_0 + a_{01}x_1 + \cdots + a_{0(n-1)}x_{n-1} \\ a_{10}x_0 + a_{11}x_1 + \cdots + a_{1(n-1)}x_{n-1} \\ \vdots \\ a_{(m-1)0}x_0 + a_{(m-1)1}x_1 + \cdots + a_{(m-1)(n-1)}x_{n-1} \end{bmatrix}
\end{aligned}
$$

We will see matrix-vector and matrix-matrix multiplications in the linear algebra sense later in this book.

4.2 Mathematical functions

Many mathematical functions in NumPy are also vectorized by default. Here you should think of functions like

- Trigonometry: `sin()`, `cos()`, `tan()`
- Exponentation and logarithms: `exp()`, `log()`, `log10()`, `log2()`
- Rounding: `around()`, `floor()`, `ceil()`
- Division with remainder: `mod()`, `divmod()`
- Power computation: `sqrt()`, `abs()`, `power()`

You access them using `np.function_name()`. Let us look at some examples; you can check out the documentation of the other functions yourself.

```
x = np.array([2,1,6])

# Compute sin(i) for every element i in x
y = np.sin(x)
print(y)
```

```
[ 0.90929743  0.84147098 -0.2794155 ]
```

```
A = np.array([[2,1,6],[1,1,3]])
```

```
# Compute e^i for every element i in A
y = np.exp(A)
print(y)
```

```
[[  7.3890561    2.71828183 403.42879349]
 [  2.71828183   2.71828183  20.08553692]]
```

```
x = np.array([1.249583, 3.110294, 4.51139])

# Round every number in x to two decimals
x = np.around(x, decimals=2)
print(x)
```

```
[1.25 3.11 4.51]
```

```
x = np.array([10,9,4])
y = np.array([2,4,5])

# Compute x_i (mod y_i) for all i
# and output divisor and remainder
z = np.divmod(x,y)
print(z)
```

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

`np.divmod()` outputs two arrays: the divisors and the remainder. For example, looking at `x[1] = 9` and `y[1] = 4`: the number 4 fits twice in 9, after which 1 is left, i.e., $9 = 2 \cdot 4 + 1$. The number 2 appears in the second position of the first array, and the remainder 1 in the second position of the second array.

```
A = np.array([[2,3,6],[4,2,3]])
N = np.array([[1,2,3],[1,2,3]])

# Pointwise compute a_{ij}^n_{ij} for all i,j
P = np.power(A,N)
print(P)
```

```
[[  2   9 216]
 [  4   4  27]]
```

4.3 Operations along array axes

Another efficient way to perform vectorized operations is to exploit the fact that many NumPy functions that perform an operation on a one-dimensional array, can also be used for two-dimensional arrays where the operation is then either performed on every column (i.e., along the 0-axis), or on every row (i.e., along the 1-axis).

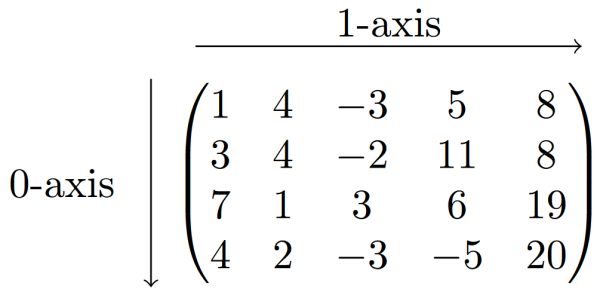

Figure 4.1: Axes of a two-dimensional array

We will look at some examples of this in the next sections.

4.3.1 Sorting and searching

The function `sort()` can be used to sort the elements in a one-dimensional array in ascending order, i.e., smallest to largest.

```
x = np.array([0.89, 0.5,  0.57, 0.34])

# Sort and print the elements in x
x_ascending = np.sort(x)
print(x_ascending)
```

```
[0.34 0.5  0.57 0.89]
```

It is not possible to use `sort()` to sort in descending order, i.e., largest to smallest, but this can be accomplished by reversing the sorted array. We can do this using index slicing with a step size of -1, starting at position -1, meaning that Python goes backwards through the array.

```
# Access x from beginning till end with step size -1
x_descending = x_ascending[-1::-1]

print(x_descending)
```

```
[0.89 0.57 0.5  0.34]
```

Vectorizing the (ascending) sort operation means we want to have a function that can take as input a two-dimensional array, and return for every row (or column) the sorted list of numbers. It turns out that `sort()` can do this right away, by adding an additional keyword argument `axis`.

Adding `axis=0` means that Python will sort the numbers in every column, i.e., along the 0-axis, and `axis=1` will sort numbers in every row, i.e., along the 1-axis.

```
A = np.array([
 [0.89, 0.5,  0.57, 0.34],
 [0.61, 0.12, 0.04, 1.  ],
 [0.27, 0.26, 0.28, 0.25],
 [0.9,  0.84, 0.15, 1.  ]])

# Sort elements in every column
A_col_ordered = np.sort(A,axis=0)
print(A_col_ordered)
```

```
[[0.27 0.12 0.04 0.25]
 [0.61 0.26 0.15 0.34]
 [0.89 0.5  0.28 1.  ]
 [0.9  0.84 0.57 1.  ]]
```

```
# Sort elements in every row
A_row_ordered = np.sort(A,axis=1)
print(A_row_ordered)
```

```
[[0.34 0.5  0.57 0.89]
 [0.04 0.12 0.61 1.  ]
 [0.25 0.26 0.27 0.28]
 [0.15 0.84 0.9  1.  ]]
```

We remark that `sort()` creates a copy, and not a view of the original matrix (see Chapter 3.7).

Another useful sorting function is `argsort()` which outputs an array indicating the relative order of the elements in an array.

```
x = np.array([0.89, 0.5,  0.57, 0.34])

# Relative position of every element
y = np.argsort(x)
print(y)
```

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

The smallest number gets a 0 in `y`, the second smallest number a 1, etc.

This function also works for two-dimensional arrays. For example, determining the relative order of the elements in every column can be done by adding `axis=0` (and similarly in every row by using `axis=1`).

```
A = np.array([
 [0.89, 0.5,  0.57, 0.34],
 [0.61, 0.12, 0.04, 1.  ],
 [0.27, 0.26, 0.28, 0.25],
 [0.9,  0.84, 0.15, 1.  ]])

# Determine relative order in every column
```

```
N = np.argsort(A,axis=0)
print(N)
```

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

4.3.2 Summary statistics

There are various other mathematical functions that can perform operations along axes by adding the `axis` keyword argument. Here we list some common ones from NumPy, that yield so-called summary statistics of a (one-dimensional) array:

- Sum and product: `sum()`, `prod()`,
- Mean and standard deviation: `mean()`, `std()`,
- Maximum and minimum: `max()`, `min()`.

We will illustrate the use of these six functions using `max()`, but the same code applies to all other functions (if the task at hand is mathematically well-defined).

```
A = np.array([
[2,3,6],
[4,2,3]
])
```

If we apply the `max()` function directly to a (two-dimensional) array, it will give the maximum value in the whole array.

```
# Gives maximum of all elements in A
A_max = np.max(A)

print(A_max)
```

```
6
```

If we add the `axis` keyword argument, we can either obtain the maximum of every row, or every column.

```
# Gives maximum of every column
A_column_max = np.max(A,axis=0)

print(A_column_max)
```

```
[4 3 6]
```

```
# Gives max of every row
A_row_max = np.max(A,axis=1)

print(A_row_max)
```

```
[6 4]
```

Another useful function is `argmax()` than can return the index (position) at which the maximum in an array is attained.

```
# Gives position of maximum in every column
A_col_argmax = np.argmax(A,axis=0)

print(A_col_argmax)
```

```
[1 0 0]
```

```
# Gives position of maximum in every row
A_row_argmax = np.argmax(A,axis=1)

print(A_row_argmax)
```

```
[2 0]
```

Note that the array containing the positions of the maxima is given as a row array. If you want to turn this into a column array (because the rows are ordered vertically in a two-dimensional array), recall you can do this as follows.

```
A_row_argmax = A_row_argmax[:,None]

print(A_row_argmax)
```

```
[[2]
 [0]]
```

If we try `np.argmax(A)` without using the `axis` keyword argument, then Python first flattens the matrix into a one-dimensional array, after which it returns the position of the maximum in the flattened array. Note that this flattening happens according to the largest index changing fasted principle (so it it places all the rows after each other, and not all the columns under each other).

Also note that if the maximum is attained in multiple places, then Python only returns the position of the first element that attains the maximum.

```
# Gives position of maximum
N = np.array([
[2,3,4],
[4,4,3]
])

N_argmax = np.argmax(N) # Turns N into [2,3,4,4,4,3];
                        # returns first position with maximum

print(N_argmax)
```

```
2
```

There are also more advance functions that give some summative information about an array:

- Cumulative sum: `cumsum()`,
- Cumulative product: `cumprod()`.

The function `cumsum()` return the cumulative sum of a one-dimensional array. As an example, if $x = [1, 4, 2, 5]$, then the cumulative sums will be given by

$$x_{\text{cumsum}} = [1, 1+4, 1+4+2, 1+4+2+5] = [1, 5, 7, 12].$$

```
x = np.array([1,4,2,5])

# Cumulative sum of x
x_cumsum = np.cumsum(x)
print('x_cumsum =', x_cumsum)
```

```
x_cumsum = [ 1  5  7 12]
```

The function can also be vectorized using the `axis` keyword argument.

The function `cumprod()` returns the cumulative product. Again, if $x = [1, 4, 2, 5]$, then the cumulative products will be given by

$$x_{\text{cumprod}} = [1, 1 \cdot 4, 1 \cdot 4 \cdot 2, 1 \cdot 4 \cdot 2 \cdot 5] = [1, 4, 8, 40].$$

```
x = np.array([1,4,2,5])

# Cumulative product of x
x_cumprod = np.cumprod(x)
print('x_cumprod =', x_cumprod)
```

```
x_cumprod = [ 1  4  8 40]
```

This function can also be vectorized using the `axis` keyword argument.