Lecture7-Numpy

August 12, 2021

1 Numpy - multidimensional data arrays

Adapted from J.R. Johansson scientific python lectures

Numpy package adds support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

[1]:
```
 # what is this line all about?
%pylab inline
import matplotlib.pyplot as plt
```

```
Populating the interactive namespace from numpy and matplotlib
```

1.1 Introduction

The `numpy` package (module) is used in almost all numerical computation using Python. It is a package that provide high-performance vector, matrix and higher-dimensional data structures for Python. It is implemented in C and Fortran so when calculations are vectorized (formulated with vectors and matrices), performance is very good.

To use `numpy` you need to import the module, using for example:

[2]:
```
import numpy as np
```

[3]:
```
"N-Dimension array"
```

[3]:
```
'N-Dimension array'
```

[4]:
```
from numpy import *
```

In the `numpy` package the terminology used for vectors, matrices and higher-dimensional data sets is *array*.

1.2 Creating numpy arrays

There are a number of ways to initialize new numpy arrays, for example from

- a Python list or tuples

- using functions that are dedicated to generating numpy arrays, such as `arange`, `linspace`, etc.
- reading data from files

1.2.1 From lists

```
[5]: type([1,4,9])
```

```
[5]: list
```

```
[6]: a1=np.array([1,4,9])
     a1
```

```
[6]: array([1, 4, 9])
```

```
[7]: type(a1)
```

```
[7]: numpy.ndarray
```

For example, to create new vector and matrix arrays from Python lists we can use the `numpy.array` function.

```
[8]: # a vector: the argument to the array function is a Python list
     v = array([1,2,3,4])

     v
```

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

```
[9]: # a matrix: the argument to the array function is a nested Python list
     M = array([[1, 2], [3, 4]])

     M
```

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

The `v` and `M` objects are both of the type `ndarray` that the `numpy` module provides.

```
[10]: type(v), type(M)
```

```
[10]: (numpy.ndarray, numpy.ndarray)
```

The difference between the `v` and `M` arrays is only their shapes. We can get information about the shape of an array by using the `ndarray.shape` property.

```
[11]: v.shape
```

```
[11]: (4,)
```

```
[12]: M.shape
```

```
[12]: (2, 2)
```

```
[13]: M3 = array([[[1, 2],[1,1]],[ [3, 4],[1,2]]])
```

```
[14]: M3
```

```
[14]: array([[[1, 2],
              [1, 1]],

             [[3, 4],
              [1, 2]]])
```

```
[15]: M3.shape
```

```
[15]: (2, 2, 2)
```

The number of elements in the array is available through the `ndarray.size` property:

```
[16]: M.size
```

```
[16]: 4
```

```
[17]: M3.size
```

```
[17]: 8
```

Equivalently, we could use the function `numpy.shape` and `numpy.size`

```
[18]: numpy.shape(M)
```

```
[18]: (2, 2)
```

```
[19]: np.size(M3)
```

```
[19]: 8
```

```
[20]: np.byte(M3)
```

```
[20]: array([[[1, 2],
              [1, 1]],

             [[3, 4],
              [1, 2]]], dtype=int8)
```

So far the `numpy.ndarray` looks awefully 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, etc. Implementing such functions for Python lists would not be very efficient because of the dynamic typing.
- 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 is used).

Using the `dtype` (data type) property of an `ndarray`, we can see what type the data of an array has:

```
[21]: M.dtype
```

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

We get an error if we try to assign a value of the wrong type to an element in a numpy array:

```
[22]: M[0,0]=1
```

```
[23]: M[0,0] = "hello"
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-23-e1f336250f69> in <module>
----> 1 M[0,0] = "hello"

ValueError: invalid literal for int() with base 10: 'hello'
```

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

```
[24]: M = array([[1, 2], [3, 4]], dtype=complex)

      M
```

```
[24]: array([[1.+0.j, 2.+0.j],
             [3.+0.j, 4.+0.j]])
```

```
[25]: M = array([[1, 2], [3, 4]], dtype=complex128)

      M
```

```
[25]: array([[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, for example: `int64`, `int16`, `float128`, `complex128`.

1.2.2 Using array-generating functions

For larger arrays it is inpractical to initialize the data manually, using explicit python lists. Instead we can use one of the many functions in `numpy` that generate arrays of different forms. Some of the more common are:

```
[26]: list (range(6))
```

```
[26]: [0, 1, 2, 3, 4, 5]
```

```
[27]: p=np.arange(0,10,1)
      p
```

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

1.2.3 arange

Generating a uniform sequence of numbers within a range

```
[28]: # create a range

      x = arange(0, 10, 1) # arguments: start, stop, step

      x
```

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

```
[29]: linspace (1,4, 4, endpoint=True)
```

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

```
[30]: #logspace?
```

```
[31]: x = arange(-1, 1, 0.1)

      x
```

```
[31]: array([-1.00000000e+00, -9.00000000e-01, -8.00000000e-01, -7.00000000e-01,
             -6.00000000e-01, -5.00000000e-01, -4.00000000e-01, -3.00000000e-01,
       -2.00000000e-01, -1.00000000e-01, -2.22044605e-16,  1.00000000e-01,
        2.00000000e-01,  3.00000000e-01,  4.00000000e-01,  5.00000000e-01,
        6.00000000e-01,  7.00000000e-01,  8.00000000e-01,  9.00000000e-01])
```

1.2.4 linspace and logspace

[32]:
```
logspace(0, 10, 10, base=e)
```

[32]:
```
array([1.00000000e+00, 3.03773178e+00, 9.22781435e+00, 2.80316249e+01,
       8.51525577e+01, 2.58670631e+02, 7.85771994e+02, 2.38696456e+03,
       7.25095809e+03, 2.20264658e+04])
```

[33]:
```
# using linspace, both end points ARE included
linspace(0, 10, 25)
```

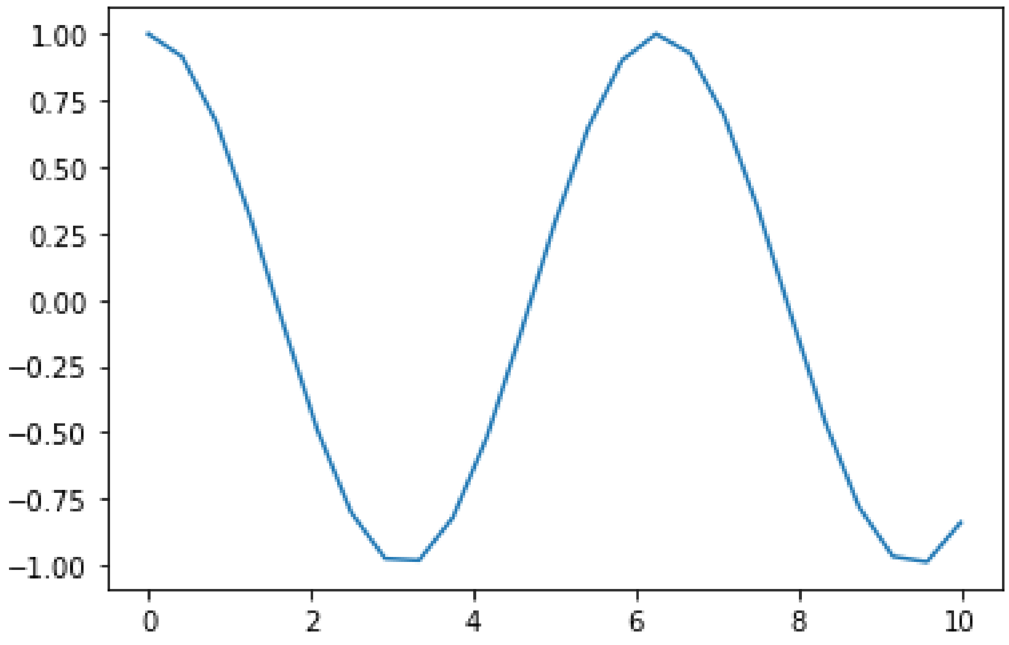
[33]:
```
array([ 0.        ,  0.41666667,  0.83333333,  1.25      ,  1.66666667,
        2.08333333,  2.5       ,  2.91666667,  3.33333333,  3.75      ,
        4.16666667,  4.58333333,  5.        ,  5.41666667,  5.83333333,
        6.25      ,  6.66666667,  7.08333333,  7.5       ,  7.91666667,
        8.33333333,  8.75      ,  9.16666667,  9.58333333, 10.        ])
```

[34]:
```
x=linspace(0, 10, 25)
y=np.cos(x)
y
```

[34]:
```
array([ 1.        ,  0.91444307,  0.67241224,  0.31532236, -0.09572355,
       -0.49038983, -0.80114362, -0.97481062, -0.981674  , -0.82055936,
       -0.51903563, -0.1286977 ,  0.28366219,  0.64748354,  0.90051148,
        0.99944942,  0.9273677 ,  0.69660051,  0.34663532, -0.06264399,
       -0.46120404, -0.78084568, -0.9668738 , -0.98745641, -0.83907153])
```

[35]:
```
plot( x,y)
```

[35]:
```
[<matplotlib.lines.Line2D at 0x7fdcd2808970>]
```



random data

```
[36]: from numpy import random
```

```
[37]: # uniform random numbers in [0,1]
      random.rand(5,5)
```

```
[37]: array([[0.30951268, 0.77928544, 0.16901818, 0.25917191, 0.7828285 ],
             [0.1032502 , 0.35682634, 0.76955186, 0.53914917, 0.80258703],
             [0.62996954, 0.88446193, 0.67245207, 0.41564092, 0.79658858],
             [0.42448812, 0.11847871, 0.05118312, 0.15308033, 0.53068815],
             [0.23883174, 0.43375573, 0.64872536, 0.6963636 , 0.22784041]])
```

```
[38]: random.rand(2,3,5)
```

```
[38]: array([[[0.70014052, 0.82520039, 0.57397227, 0.68254125, 0.7640451 ],
              [0.34956663, 0.38578772, 0.1520008 , 0.45564271, 0.22114899],
              [0.59179652, 0.29860677, 0.1706749 , 0.81816191, 0.97899971]],

             [[0.35356495, 0.07010458, 0.86279325, 0.94240227, 0.00953878],
              [0.02147495, 0.5439528 , 0.33091377, 0.57789781, 0.25821672],
              [0.95143958, 0.67415712, 0.70696788, 0.46943587, 0.17130585]]])
```

```
[36]: #help(random)
```

[37]:
```
# standard normal distributed random numbers
random.randn(5,5)
```

[37]:
```
array([[-1.36194228, -0.43669067,  0.33709594,  0.9099189 ,  0.29108419],
       [-0.57333526, -1.73120288,  0.55444915, -0.4189142 ,  1.19580453],
       [ 0.97959712,  0.6336788 ,  1.97504587,  0.99245584,  0.34754526],
       [ 0.89146995, -0.38390246, -1.25490661,  0.47010874,  0.53251642],
       [-0.0731591 , -2.04530934,  0.28894173,  0.8683622 ,  0.39647244]])
```

[38]:
```
random.randint(1,100,  10)
```

[38]:
```
array([27, 66, 40, 35, 81, 58, 46, 75, 68, 55])
```

[39]:
```
random.randint(1,100,  (5,5))
```

[39]:
```
array([[60, 99, 84, 11, 77],
       [22, 44, 12, 23, 68],
       [60, 46, 99, 26, 77],
       [67, 45,  5, 25,  5],
       [ 2, 31, 35, 21, 19]])
```

[40]:
```
rn=random.randint(1,100,  (5,5))
rn
```

[40]:
```
array([[93, 81, 10, 99, 83],
       [23, 18, 43, 14,  1],
       [78, 54, 53, 20, 18],
       [75, 71, 71, 15, 57],
       [65, 52, 57, 14, 11]])
```

[41]:
```
rn[1,:]
```

[41]:
```
array([23, 18, 43, 14,  1])
```

[42]:
```
rn[:,1]
```

[42]:
```
array([81, 18, 54, 71, 52])
```

[43]:
```
rn[0:2, ]
```

[43]:
```
array([[93, 81, 10, 99, 83],
       [23, 18, 43, 14,  1]])
```

[44]:
```
rn
```

[44]:
```
array([[93, 81, 10, 99, 83],
       [23, 18, 43, 14,  1],
         [78, 54, 53, 20, 18],
         [75, 71, 71, 15, 57],
         [65, 52, 57, 14, 11]])
```

```
[47]: rn[::2, ]
```

```
[47]: array([[93, 81, 10, 99, 83],
             [78, 54, 53, 20, 18],
             [65, 52, 57, 14, 11]])
```

```
[48]: rn[:,::2]
```

```
[48]: array([[93, 10, 83],
             [23, 43,  1],
             [78, 53, 18],
             [75, 71, 57],
             [65, 57, 11]])
```

1.2.5 Creating Diagonal Matrix Object with diag

```
[ ]:
```

```
[49]: # a diagonal matrix
      numpy.diag(arange(1,5))
```

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

```
[50]: numpy.diag(arange(1,5),k=-1)
```

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

```
[51]: # diagonal with offset from the main diagonal
      diag([1,2,3], k=1)
```

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

1.2.6 Creating arrays of zeros and ones

```
[52]: zeros((3,6))
```

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

```
[53]: zeros((3,3,3))
```

```
[53]: array([[[0., 0., 0.],
              [0., 0., 0.],
              [0., 0., 0.]],

             [[0., 0., 0.],
              [0., 0., 0.],
              [0., 0., 0.]],

             [[0., 0., 0.],
              [0., 0., 0.],
              [0., 0., 0.]]])
```

```
[54]: ones((3,3))
```

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

```
[55]: ones((3,3,3))
```

```
[55]: array([[[1., 1., 1.],
              [1., 1., 1.],
              [1., 1., 1.]],

             [[1., 1., 1.],
              [1., 1., 1.],
              [1., 1., 1.]],

             [[1., 1., 1.],
              [1., 1., 1.],
              [1., 1., 1.]]])
```

1.3 File I/O

1.3.1 Comma-separated values (CSV)

A very common file format for data files is comma-separated values (CSV), or related formats such as TSV (tab-separated values). To read data from such files into Numpy arrays we can use the `numpy.genfromtxt` function. For example,

[56]:
```
import numpy
import numpy as np
from numpy import *
```

[57]:
```
#for Unix like machines if it fails try next cell for windows
!tail -5 stockholm_td_adj.dat
```

```
2011 12 27     8.3     7.6     7.6 1
2011 12 28     2.6     1.9     1.9 1
2011 12 29     4.9     4.2     4.2 1
2011 12 30     0.6    -0.1    -0.1 1
2011 12 31    -2.6    -3.3    -3.3 1
```

[58]:
```
#for windows machines
# !more -5 stockholm_td_adj.dat
```

[62]:
```
data = genfromtxt('stockholm_td_adj.dat')
```

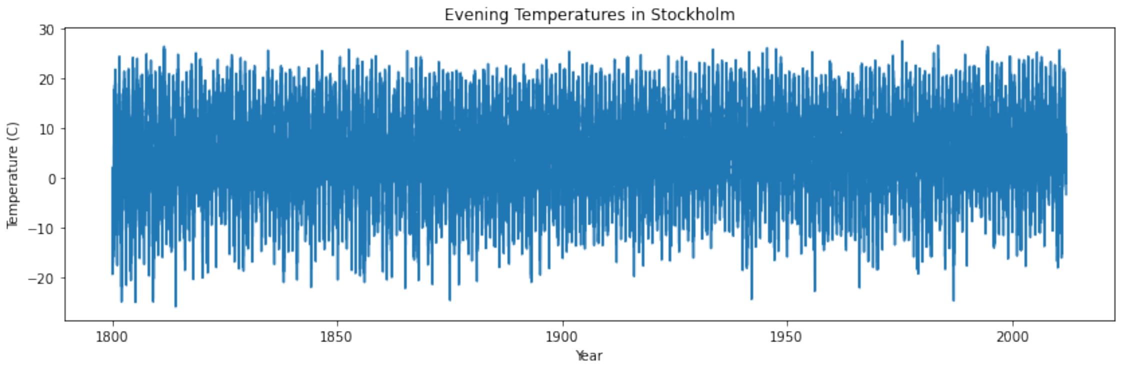
[63]:
```
# look up genfromtxt documentation
#np.genfromtxt?
```

[64]:
```
data.shape # get the number of rows and columns
```

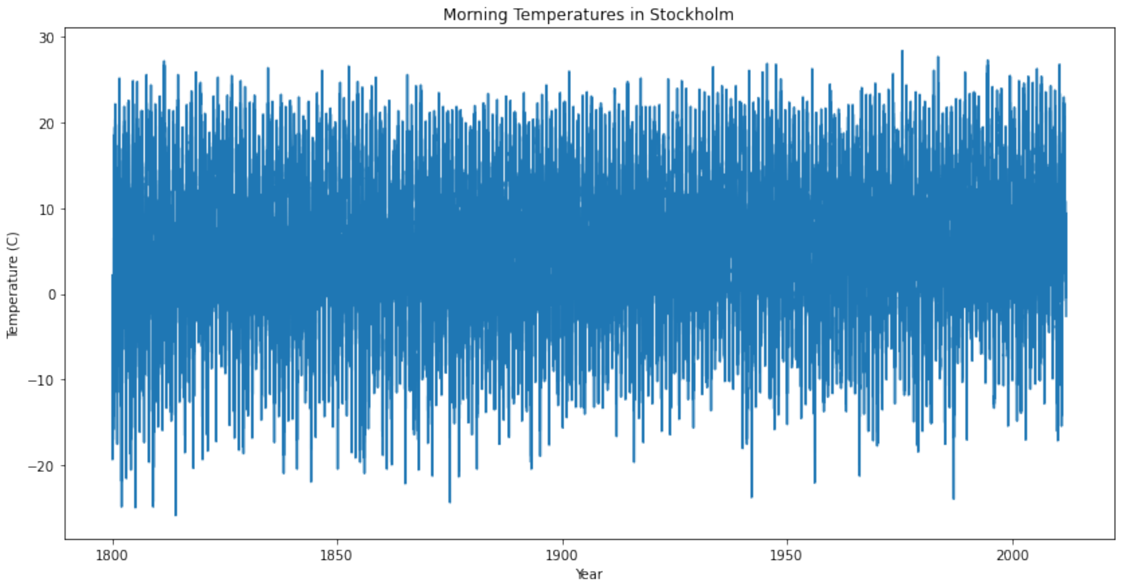
[64]:
```
(77431, 7)
```

[65]:
```
#do some simple x vs y plot  x-->year and y--> temperature
```

[67]:
```
#First we look @ Evening Temperatures



fig, ax = plt.subplots(figsize=(14,4))
ax.plot(data[:,0]+data[:,1]/12.0+data[:,2]/365, data[:,5])
ax.axis('tight')
ax.set_title('Evening Temperatures in Stockholm')
ax.set_xlabel('Year')
ax.set_ylabel('Temperature (C)');
```

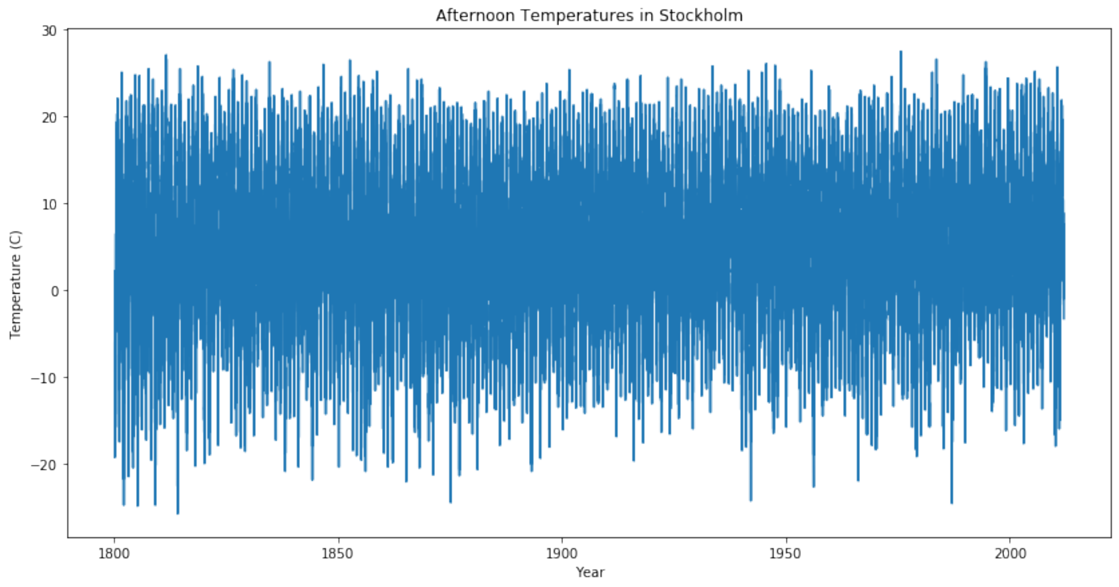


[68]:
```
#next  we look @ Morning Temperatures

fig, ax = plt.subplots(figsize=(14,7))
ax.plot(data[:,0]+data[:,1]/12.0+data[:,2]/365, data[:,3])
ax.axis('tight')
ax.set_title('Morning Temperatures in Stockholm')
ax.set_xlabel('Year')
ax.set_ylabel('Temperature (C)');
```



1.3.2 Exercise

```
[74]: ##Write code to Plot Afternoon Temps
```



1.3.3 Exploratory Data Analysis (EDA)

Perform some exploratory statistics analysis on the data

```
[ ]:
```

```
[75]: data[:,5].mean() # average for Evening Temperatures
```

```
[75]: 5.785461895106611
```

```
[76]: data[:,4].mean()
```

```
[76]: 5.832170577675608
```

```
[77]: data[:,3].mean() # average for Morning Temperatures
```

```
[77]: 6.197109684751585
```

```
[78]: data[:,5].std() # standard deviation for Evening Temperatures
```

```
[78]: 8.195919794033603
```

```
[79]: data[:,5].var() # variance for Evening Temperatures
```

```
[79]: 67.17310127023183
```

```
[80]: data[0:1001,3:6].mean()
```

```
[80]: 5.820279720279721
```

```
[81]: data[0:3,3:6]
```

```
[81]: array([[ -6.1,  -6.1,  -6.1],
              [-15.4, -15.4, -15.4],
              [-15. , -15. , -15. ]])
```

```
[82]: data[0:3][3:6]
```

```
[82]: array([], shape=(0, 7), dtype=float64)
```

Using `numpy.savetxt` we can store a Numpy array to a file in CSV format:

```
[70]: M = random.rand(10,5)

      M
```

```
[70]: array([[0.09990402, 0.5550446 , 0.35010804, 0.9781904 , 0.6224126 ],
              [0.61822692, 0.42856064, 0.1986308 , 0.74045149, 0.14352753],
              [0.41406803, 0.39317735, 0.85181569, 0.90579844, 0.44251318],
              [0.69433211, 0.80338309, 0.25563253, 0.90791681, 0.46041614],
              [0.47239972, 0.72520953, 0.58065307, 0.7766819 , 0.24750281],
              [0.37015486, 0.58511578, 0.14354664, 0.22880735, 0.49287729],
              [0.56103113, 0.25208411, 0.59228539, 0.72606034, 0.03079991],
              [0.56087782, 0.15584017, 0.44920521, 0.76865568, 0.69264691],
              [0.44028212, 0.18028016, 0.27869661, 0.85713004, 0.16796867],
              [0.83102986, 0.32758873, 0.29688657, 0.98195499, 0.91289389]])
```

```
[71]: savetxt("random-matrix.csv", M)
```

```
[72]: !cat random-matrix.csv
```

```
9.990402163319733742e-02 5.550446022005015578e-01 3.501080421727730263e-01
9.781904030838612929e-01 6.224126047826715746e-01
6.182269244937682595e-01 4.285606372560537558e-01 1.986307977541763581e-01
7.404514879187601428e-01 1.435275305763168419e-01
4.140680270244053718e-01 3.931773493925482255e-01 8.518156870961482996e-01
9.057984429108782987e-01 4.425131823114547558e-01
6.943321050906794412e-01 8.033830895655192927e-01 2.556325316828705896e-01
9.079168050675234802e-01 4.604161381356679517e-01
4.723997160165507303e-01 7.252095330671231022e-01 5.806530686665177798e-01
```

```
7.766818978985596811e-01 2.475028094573151760e-01
3.701548575842296129e-01 5.851157786261430571e-01 1.435466350833083382e-01
2.288073546738541575e-01 4.928772867486566067e-01
5.610311258310609617e-01 2.520841085373980439e-01 5.922853863515079587e-01
7.260603440226999217e-01 3.079990806053423125e-02
5.608778174784845305e-01 1.558401670282887341e-01 4.492052074615394686e-01
7.686556763555347782e-01 6.926469097575121969e-01
4.402821224483902673e-01 1.802801629735163447e-01 2.786966108984065826e-01
8.571300357765431199e-01 1.679686708134426887e-01
8.310298648080345352e-01 3.275887270464660217e-01 2.968865686019870331e-01
9.819549885380900456e-01 9.128938893766740392e-01
```

[73]:
```
!head -3 random-matrix.csv
```

```
9.990402163319733742e-02 5.550446022005015578e-01 3.501080421727730263e-01
9.781904030838612929e-01 6.224126047826715746e-01
6.182269244937682595e-01 4.285606372560537558e-01 1.986307977541763581e-01
7.404514879187601428e-01 1.435275305763168419e-01
4.140680270244053718e-01 3.931773493925482255e-01 8.518156870961482996e-01
9.057984429108782987e-01 4.425131823114547558e-01
```

[74]:
```
savetxt("random-matrix.csv", M, fmt='%.5f') # fmt specifies the format, here␣
↪keep 5 decimal places

!cat random-matrix.csv
```

```
0.09990 0.55504 0.35011 0.97819 0.62241
0.61823 0.42856 0.19863 0.74045 0.14353
0.41407 0.39318 0.85182 0.90580 0.44251
0.69433 0.80338 0.25563 0.90792 0.46042
0.47240 0.72521 0.58065 0.77668 0.24750
0.37015 0.58512 0.14355 0.22881 0.49288
0.56103 0.25208 0.59229 0.72606 0.03080
0.56088 0.15584 0.44921 0.76866 0.69265
0.44028 0.18028 0.27870 0.85713 0.16797
0.83103 0.32759 0.29689 0.98195 0.91289
```

1.3.4 Managing Data in Numpy's native file format

Useful when storing and reading back numpy array data. Use the functions `numpy.save` and `numpy.load`:

Writing Data in Numpy's native file format

[82]:
```
save("random-matrix.npy", M)
```

[84]:
```
#Writing data in compressed format
```

[85]:
```
savez_compressed("random-matrix", M)
!ls -tlhr random-matrix*
```

```
-rw-rw-r-- 1 jebalunode jebalunode 400 Aug 11 23:26 random-matrix.csv
-rw-rw-r-- 1 jebalunode jebalunode 528 Aug 11 23:32 random-matrix.npy
-rw-rw-r-- 1 jebalunode jebalunode 631 Aug 11 23:35 random-matrix.npz
```

Reading Data in Numpy's native file format

[81]:
```
load("random-matrix.npy")
```

[81]:
```
array([[0.09990402, 0.5550446 , 0.35010804, 0.9781904 , 0.6224126 ],
       [0.61822692, 0.42856064, 0.1986308 , 0.74045149, 0.14352753],
       [0.41406803, 0.39317735, 0.85181569, 0.90579844, 0.44251318],
       [0.69433211, 0.80338309, 0.25563253, 0.90791681, 0.46041614],
       [0.47239972, 0.72520953, 0.58065307, 0.7766819 , 0.24750281],
       [0.37015486, 0.58511578, 0.14354664, 0.22880735, 0.49287729],
       [0.56103113, 0.25208411, 0.59228539, 0.72606034, 0.03079991],
       [0.56087782, 0.15584017, 0.44920521, 0.76865568, 0.69264691],
       [0.44028212, 0.18028016, 0.27869661, 0.85713004, 0.16796867],
       [0.83102986, 0.32758873, 0.29688657, 0.98195499, 0.91289389]])
```

[83]:
```
newM=load("random-matrix.npy")
newM
```

[83]:
```
array([[0.09990402, 0.5550446 , 0.35010804, 0.9781904 , 0.6224126 ],
       [0.61822692, 0.42856064, 0.1986308 , 0.74045149, 0.14352753],
       [0.41406803, 0.39317735, 0.85181569, 0.90579844, 0.44251318],
       [0.69433211, 0.80338309, 0.25563253, 0.90791681, 0.46041614],
       [0.47239972, 0.72520953, 0.58065307, 0.7766819 , 0.24750281],
       [0.37015486, 0.58511578, 0.14354664, 0.22880735, 0.49287729],
       [0.56103113, 0.25208411, 0.59228539, 0.72606034, 0.03079991],
       [0.56087782, 0.15584017, 0.44920521, 0.76865568, 0.69264691],
       [0.44028212, 0.18028016, 0.27869661, 0.85713004, 0.16796867],
       [0.83102986, 0.32758873, 0.29688657, 0.98195499, 0.91289389]])
```

A little more effort is required to read compressed data

[87]:
```
load("random-matrix.npz")
```

[87]:
```
<numpy.lib.npyio.NpzFile at 0x7fdccf3849a0>
```

[88]:
```
load("random-matrix.npz").keys()
```

[88]:
```
KeysView(<numpy.lib.npyio.NpzFile object at 0x7fdccf3848e0>)
```

[89]:
```
list(load("random-matrix.npz").keys())
```

```
[89]: ['arr_0']
```

```
[91]: load("random-matrix.npz")['arr_0'] # use the key to access the data
```

```
[91]: array([[0.09990402, 0.5550446 , 0.35010804, 0.9781904 , 0.6224126 ],
              [0.61822692, 0.42856064, 0.1986308 , 0.74045149, 0.14352753],
              [0.41406803, 0.39317735, 0.85181569, 0.90579844, 0.44251318],
              [0.69433211, 0.80338309, 0.25563253, 0.90791681, 0.46041614],
              [0.47239972, 0.72520953, 0.58065307, 0.7766819 , 0.24750281],
              [0.37015486, 0.58511578, 0.14354664, 0.22880735, 0.49287729],
              [0.56103113, 0.25208411, 0.59228539, 0.72606034, 0.03079991],
              [0.56087782, 0.15584017, 0.44920521, 0.76865568, 0.69264691],
              [0.44028212, 0.18028016, 0.27869661, 0.85713004, 0.16796867],
              [0.83102986, 0.32758873, 0.29688657, 0.98195499, 0.91289389]])
```

```
[93]: a=load("random-matrix.npz")['arr_0']  # use the key to access the data

      print(a)
```

```
[[0.09990402 0.5550446  0.35010804 0.9781904  0.6224126 ]
 [0.61822692 0.42856064 0.1986308  0.74045149 0.14352753]
 [0.41406803 0.39317735 0.85181569 0.90579844 0.44251318]
 [0.69433211 0.80338309 0.25563253 0.90791681 0.46041614]
 [0.47239972 0.72520953 0.58065307 0.7766819  0.24750281]
 [0.37015486 0.58511578 0.14354664 0.22880735 0.49287729]
 [0.56103113 0.25208411 0.59228539 0.72606034 0.03079991]
 [0.56087782 0.15584017 0.44920521 0.76865568 0.69264691]
 [0.44028212 0.18028016 0.27869661 0.85713004 0.16796867]
 [0.83102986 0.32758873 0.29688657 0.98195499 0.91289389]]
```

```
[ ]:
```

```
[94]: help(np.save)
```

```
Help on function save in module numpy:

save(file, arr, allow_pickle=True, fix_imports=True)
    Save an array to a binary file in NumPy ``.npy`` format.

    Parameters
    ----------
    file : file, str, or pathlib.Path
        File or filename to which the data is saved.  If file is a file-object,
        then the filename is unchanged.  If file is a string or Path, a ``.npy``
        extension will be appended to the filename if it does not already
        have one.
    arr : array_like
        Array data to be saved.
    allow_pickle : bool, optional
        Allow saving object arrays using Python pickles. Reasons for disallowing
        pickles include security (loading pickled data can execute arbitrary
        code) and portability (pickled objects may not be loadable on different
        Python installations, for example if the stored objects require
libraries
        that are not available, and not all pickled data is compatible between
        Python 2 and Python 3).
        Default: True
    fix_imports : bool, optional
        Only useful in forcing objects in object arrays on Python 3 to be
        pickled in a Python 2 compatible way. If `fix_imports` is True, pickle
        will try to map the new Python 3 names to the old module names used in
        Python 2, so that the pickle data stream is readable with Python 2.

    See Also
    --------
    savez : Save several arrays into a ``.npz`` archive
    savetxt, load

    Notes
    -----
    For a description of the ``.npy`` format, see :py:mod:`numpy.lib.format`.

    Any data saved to the file is appended to the end of the file.

    Examples
    --------
    >>> from tempfile import TemporaryFile
    >>> outfile = TemporaryFile()

    >>> x = np.arange(10)
    >>> np.save(outfile, x)

    >>> _ = outfile.seek(0) # Only needed here to simulate closing & reopening
file
    >>> np.load(outfile)
    array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])


    >>> with open('test.npy', 'wb') as f:
    …     np.save(f, np.array([1, 2]))
    …     np.save(f, np.array([1, 3]))
    >>> with open('test.npy', 'rb') as f:
    …     a = np.load(f)
    …     b = np.load(f)
    >>> print(a, b)
    # [1 2] [1 3]
```

[98]:
```python
help(savez_compressed)
```

```
Help on function savez_compressed in module numpy.lib.npyio:

savez_compressed(file, *args, **kwds)
    Save several arrays into a single file in compressed ``.npz`` format.

    If keyword arguments are given, then filenames are taken from the keywords.
    If arguments are passed in with no keywords, then stored file names are
    arr_0, arr_1, etc.

    Parameters
    ----------
    file : str or file
        Either the file name (string) or an open file (file-like object)
        where the data will be saved. If file is a string or a Path, the
        ``.npz`` extension will be appended to the file name if it is not
        already there.
    args : Arguments, optional
        Arrays to save to the file. Since it is not possible for Python to
        know the names of the arrays outside `savez`, the arrays will be saved
        with names "arr_0", "arr_1", and so on. These arguments can be any
        expression.
    kwds : Keyword arguments, optional
        Arrays to save to the file. Arrays will be saved in the file with the
        keyword names.

    Returns
    -------
    None

    See Also
    --------
    numpy.save : Save a single array to a binary file in NumPy format.
    numpy.savetxt : Save an array to a file as plain text.
    numpy.savez : Save several arrays into an uncompressed ``.npz`` file format
    numpy.load : Load the files created by savez_compressed.

    Notes
    -----
    The ``.npz`` file format is a zipped archive of files named after the
    variables they contain.  The archive is compressed with
    ``zipfile.ZIP_DEFLATED`` and each file in the archive contains one variable
    in ``.npy`` format. For a description of the ``.npy`` format, see
    :py:mod:`numpy.lib.format`.
    When opening the saved ``.npz`` file with `load` a `NpzFile` object is
    returned. This is a dictionary-like object which can be queried for
    its list of arrays (with the ``.files`` attribute), and for the arrays
    themselves.

    Examples
    --------
    >>> test_array = np.random.rand(3, 2)
    >>> test_vector = np.random.rand(4)
    >>> np.savez_compressed('/tmp/123', a=test_array, b=test_vector)
    >>> loaded = np.load('/tmp/123.npz')
    >>> print(np.array_equal(test_array, loaded['a']))
    True
    >>> print(np.array_equal(test_vector, loaded['b']))
    True
```

[99]:
```
savez_compressed("various-data.npz", M, random.rand(4,4), linspace(0,10,5))
```

[100]:
```
list (load('various-data.npz').keys())
```

[100]:
```
['arr_0', 'arr_1', 'arr_2']
```

[101]:
```
various_data=load("various-data.npz")
for key,value in various_data.items():
    print ("key = ",key)
    print("value = ", value)
```

```
key =  arr_0
value =  [[0.09990402 0.5550446  0.35010804 0.9781904  0.6224126 ]
 [0.61822692 0.42856064 0.1986308  0.74045149 0.14352753]
 [0.41406803 0.39317735 0.85181569 0.90579844 0.44251318]
 [0.69433211 0.80338309 0.25563253 0.90791681 0.46041614]
 [0.47239972 0.72520953 0.58065307 0.7766819  0.24750281]
 [0.37015486 0.58511578 0.14354664 0.22880735 0.49287729]
 [0.56103113 0.25208411 0.59228539 0.72606034 0.03079991]
 [0.56087782 0.15584017 0.44920521 0.76865568 0.69264691]
 [0.44028212 0.18028016 0.27869661 0.85713004 0.16796867]
 [0.83102986 0.32758873 0.29688657 0.98195499 0.91289389]]
key =  arr_1
value =  [[0.51063914 0.52930462 0.99993401 0.84881395]
 [0.64164389 0.96451481 0.53614269 0.10199684]
 [0.74504328 0.34142105 0.45264598 0.06213483]
 [0.7288016  0.98488401 0.44337661 0.74033652]]
key =  arr_2
value =  [ 0.   2.5  5.   7.5 10. ]
```

1.4 More properties of the numpy arrays

```
[102]: M.itemsize # bytes per element
```

```
[102]: 8
```

```
[103]: M.nbytes # number of bytes
```

```
[103]: 400
```

```
[104]: M.ndim # number of dimensions
```

```
[104]: 2
```

1.5 Manipulating arrays

1.5.1 Indexing

We can index elements in an array using square brackets and indices:

```
[105]: v=random.rand(10)
```

```
[106]: # v is a vector, and has only one dimension, taking one index
       v[0]
```

```
[106]: 0.8462065114169961
```

```
[107]: # M is a matrix, or a 2 dimensional array, taking two indices
       M[1,1]
```

```
[107]: 0.42856063725605376
```

If we omit an index of a multidimensional array it returns the whole row (or, in general, a N-1 dimensional array)

```
[108]: M
```

```
[108]: array([[0.09990402, 0.5550446 , 0.35010804, 0.9781904 , 0.6224126 ],
              [0.61822692, 0.42856064, 0.1986308 , 0.74045149, 0.14352753],
              [0.41406803, 0.39317735, 0.85181569, 0.90579844, 0.44251318],
              [0.69433211, 0.80338309, 0.25563253, 0.90791681, 0.46041614],
              [0.47239972, 0.72520953, 0.58065307, 0.7766819 , 0.24750281],
              [0.37015486, 0.58511578, 0.14354664, 0.22880735, 0.49287729],
              [0.56103113, 0.25208411, 0.59228539, 0.72606034, 0.03079991],
              [0.56087782, 0.15584017, 0.44920521, 0.76865568, 0.69264691],
              [0.44028212, 0.18028016, 0.27869661, 0.85713004, 0.16796867],
        [0.83102986, 0.32758873, 0.29688657, 0.98195499, 0.91289389]])
```

```
[109]: M[1]
```

```
[109]: array([0.61822692, 0.42856064, 0.1986308 , 0.74045149, 0.14352753])
```

The same thing can be achieved with using `:` instead of an index:

```
[110]: M[1,:] # row 1
```

```
[110]: array([0.61822692, 0.42856064, 0.1986308 , 0.74045149, 0.14352753])
```

```
[111]: M[:,1] # column 1
```

```
[111]: array([0.5550446 , 0.42856064, 0.39317735, 0.80338309, 0.72520953,
              0.58511578, 0.25208411, 0.15584017, 0.18028016, 0.32758873])
```

We can assign new values to elements in an array using indexing:

```
[112]: M[0,0] = 1
```

```
[113]: M
```

```
[113]: array([[1.        , 0.5550446 , 0.35010804, 0.9781904 , 0.6224126 ],
               [0.61822692, 0.42856064, 0.1986308 , 0.74045149, 0.14352753],
               [0.41406803, 0.39317735, 0.85181569, 0.90579844, 0.44251318],
               [0.69433211, 0.80338309, 0.25563253, 0.90791681, 0.46041614],
               [0.47239972, 0.72520953, 0.58065307, 0.7766819 , 0.24750281],
               [0.37015486, 0.58511578, 0.14354664, 0.22880735, 0.49287729],
               [0.56103113, 0.25208411, 0.59228539, 0.72606034, 0.03079991],
               [0.56087782, 0.15584017, 0.44920521, 0.76865568, 0.69264691],
               [0.44028212, 0.18028016, 0.27869661, 0.85713004, 0.16796867],
               [0.83102986, 0.32758873, 0.29688657, 0.98195499, 0.91289389]])
```

```
[116]: # also works for rows and columns
       M[1,:] = 0 # Assign 0 to all entries in the second row
       M[:,2] = -1 # Assign 1 to all entries in the third column
```

```
[117]: M
```

```
[117]: array([[ 1.        ,  0.5550446 , -1.        ,  0.9781904 ,  0.6224126 ],
               [ 0.        ,  0.        , -1.        ,  0.        ,  0.        ],
               [ 0.41406803,  0.39317735, -1.        ,  0.90579844,  0.44251318],
               [ 0.69433211,  0.80338309, -1.        ,  0.90791681,  0.46041614],
               [ 0.47239972,  0.72520953, -1.        ,  0.7766819 ,  0.24750281],
               [ 0.37015486,  0.58511578, -1.        ,  0.22880735,  0.49287729],
               [ 0.56103113,  0.25208411, -1.        ,  0.72606034,  0.03079991],
               [ 0.56087782,  0.15584017, -1.        ,  0.76865568,  0.69264691],
               [ 0.44028212,  0.18028016, -1.        ,  0.85713004,  0.16796867],
               [ 0.83102986,  0.32758873, -1.        ,  0.98195499,  0.91289389]])
```

1.5.2 Index slicing

Index slicing is the technical name for the syntax `M[lower:upper:step]` to extract part of an array:

```
[118]: A = array([1,2,3,4,5])
       A
```

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

```
[119]: A[1:3]
```

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

Array slices are *mutable*: if they are assigned a new value the original array from which the slice was extracted is modified:

```
[120]: A[1:3] = [-2,-3]

       A
```

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

We can omit any of the three parameters in `M[lower:upper:step]`:

```
[121]: A[::] # lower, upper, step all take the default values
```

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

```
[122]: A[::2] # step is 2, lower and upper defaults to the beginning and end of the␣
        ↪array
```

```
[122]: array([ 1, -3,  5])
```

```
[123]: A[:3] # first three elements
```

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

```
[124]: A[3:] # elements from index 3
```

```
[124]: array([4, 5])
```

```
[125]: A[::-1]
```

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

Negative indices counts from the end of the array (positive index from the begining):

```
[126]: A = array([1,2,3,4,5])
```

```
[127]: A[-1] # the last element in the array
```

```
[127]: 5
```

```
[128]: A[-3:] # the last three elements
```

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

Index slicing works exactly the same way for multidimensional arrays:

```
[129]: A = array([[n+m*10 for n in range(5)] for m in range(5)])

       A
```

```
[129]: array([[ 0,  1,  2,  3,  4],
              [10, 11, 12, 13, 14],
              [20, 21, 22, 23, 24],
              [30, 31, 32, 33, 34],
              [40, 41, 42, 43, 44]])
```

```
[130]: # a block from the original array
       A[1:4, 1:4]
```

```
[130]: array([[11, 12, 13],
              [21, 22, 23],
              [31, 32, 33]])
```

```
[131]: # strides
       A[::2, ::2]
```

```
[131]: array([[ 0,  2,  4],
              [20, 22, 24],
              [40, 42, 44]])
```

1.5.3 Fancy indexing

Fancy indexing is the name for when an array or list is used in-place of an index:

```
[132]: A
```

```
[132]: array([[ 0,  1,  2,  3,  4],
              [10, 11, 12, 13, 14],
              [20, 21, 22, 23, 24],
              [30, 31, 32, 33, 34],
              [40, 41, 42, 43, 44]])
```

[133]:
```
row_indices = [1, 2, 3]
A[row_indices]
```

[133]:
```
array([[10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34]])
```

[134]:
```
col_indices = [1, 2, -1] # remember, index -1 means the last element
A[row_indices, col_indices]
```

[134]:
```
array([11, 22, 34])
```

We can also use index masks: If the index mask is an Numpy array of data type `bool`, then an element is selected (True) or not (False) depending on the value of the index mask at the position of each element:

[135]:
```
B = arange(5)
B
```

[135]:
```
array([0, 1, 2, 3, 4])
```

[136]:
```
row_mask = array([True, False, True, False, False])
B[row_mask]
```

[136]:
```
array([0, 2])
```

[137]:
```
# same thing
row_mask = array([1,0,1,0,0], dtype=bool)
print (row_mask)
B[row_mask]
```

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

[137]:
```
array([0, 2])
```

This feature is very useful to conditionally select elements from an array, using for example comparison operators:

[138]:
```
x = arange(0, 10, 0.5)
x
```

[138]:
```
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. , 5.5, 6. ,
       6.5, 7. , 7.5, 8. , 8.5, 9. , 9.5])
```

```
[139]: mask = (5 < x) * (x < 7.5)

       mask
```

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

```
[140]: mask = (5 < x) & (x < 7.5)

       mask
```

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

```
[141]: x[mask]
```

```
[141]: array([5.5, 6. , 6.5, 7. ])
```

Recall descripitve statistics for data can be computed. What about comparing means for different periods ?

```
[147]: print(f"All time mean temperature is  {data[:,5].mean():.5f} C")
```

```
All time mean temperature is  5.78546 C
```

```
[148]: mask1800 = data[:,0]==1800 #create a mask to capture rows marked as 1800


       print("All time mean for the 1800s")
       data[mask1800][:,5].mean()
```

```
All time mean for the 1800s
```

```
[148]: 4.917808219178082
```

This feature is very useful to conditionally select elements from an array, using for example comparison operators:

2 Exercises

2.0.1 Produce a line plot for date vs temperatures for only the 1800 century

[143]:
```
import matplotlib.pyplot as plt
%pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

[ ]:
```

```

2.0.2 Plot temperatures for only the 1900 century

[ ]:
```

```

2.0.3 Whats the average evening temperatures for the 1800 century

[ ]:
```

```

2.0.4 Whats the average evening temperatures for the 1900 century

[ ]:
```

```

2.0.5 Is this region getting warmer?

[ ]:
```

```

2.1 Functions for extracting data from arrays and creating arrays

2.1.1 where

The index mask can be converted to position index using the `where` function

[155]:
```
x = arange(0, 10, 0.5)
print(f'original x\n {x}')
mask = (5 < x) & (x < 7.5)
print(f'\n\nmask on  x \n{mask}')


indices = where(mask)
```

```
print(f'\n\nactual positions on x that meet mask criteria \n {indices}')
```

```
original x
 [0.  0.5 1.  1.5 2.  2.5 3.  3.5 4.  4.5 5.  5.5 6.  6.5 7.  7.5 8.  8.5
 9.  9.5]


mask on  x
[False False False False False False False False False False False  True
  True  True  True False False False False False]


actual positions on x that meet mask criteria
 (array([11, 12, 13, 14]),)
```

[156]:
```
x[indices] # this indexing is equivalent to the fancy indexing x[mask]
```

[156]:
```
array([5.5, 6. , 6.5, 7. ])
```

3 Matrix and Algebra

3.0.1 diag

With the diag function we can also extract the diagonal and subdiagonals of an array:

[157]:
```
random.seed(15)
A=random.randint(1, 50, (5,5))
A
```

[157]:
```
array([[ 9, 13,  6,  1, 29],
       [28,  8, 12, 22, 48],
       [30, 18, 46, 32, 24],
       [33, 11, 16,  5, 42],
       [40, 38, 20, 45, 14]])
```

[158]:
```
diag(A)
```

[158]:
```
array([ 9,  8, 46,  5, 14])
```

[159]:
```
diag(A, 1)
```

[159]:
```
array([13, 12, 32, 42])
```

[160]:
```
diag(A, -1)
```

```
[160]: array([28, 18, 16, 45])
```

3.0.2 take

The `take` function is similar to fancy indexing described above:

```
[161]: v2 = arange(-3,3)
       v2
```

```
[161]: array([-3, -2, -1,  0,  1,  2])
```

```
[162]: row_indices = [1, 3, 5]
       v2[row_indices] # fancy indexing
```

```
[162]: array([-2,  0,  2])
```

```
[163]: v2.take(row_indices)
```

```
[163]: array([-2,  0,  2])
```

But `take` also works on lists and other objects:

```
[164]: take([-3, -2, -1,  0,  1,  2], row_indices)
```

```
[164]: array([-2,  0,  2])
```

3.0.3 choose

Constructs an array by picking elements from several arrays:

```
[165]: which = [1, 0, 1, 0]
       choices = [[-2,-2,-2,-2], [5,5,5,5]]

       choose(which, choices)
```

```
[165]: array([ 5, -2,  5, -2])
```

```
[166]: which = [1, 0, 1, 0]
       choices = [[-2,-2,-2,3], [5,6,5,5]]

       choose(which, choices)
```

```
[166]: array([ 5, -2,  5,  3])
```

3.1 Linear algebra

Vectorizing code is the key to writing efficient numerical calculation with Python/Numpy. That means that as much as possible of a program should be formulated in terms of matrix and vector operations, like matrix-matrix multiplication.

3.1.1 Scalar-array operations

We can use the usual arithmetic operators to multiply, add, subtract, and divide arrays with scalar numbers.

```
[167]: v1 = arange(0, 5)
```

```
[168]: v1 * 2
```

```
[168]: array([0, 2, 4, 6, 8])
```

```
[169]: v1 + 2
```

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

```
[170]: A * 2, A + 2
```

```
[170]: (array([[18, 26, 12,  2, 58],
                [56, 16, 24, 44, 96],
                [60, 36, 92, 64, 48],
                [66, 22, 32, 10, 84],
                [80, 76, 40, 90, 28]]),
         array([[11, 15,  8,  3, 31],
                [30, 10, 14, 24, 50],
                [32, 20, 48, 34, 26],
                [35, 13, 18,  7, 44],
                [42, 40, 22, 47, 16]]))
```

3.1.2 Element-wise array-array operations

When we add, subtract, multiply and divide arrays with each other, the default behaviour is **element-wise** operations:

```
[178]: A
```

```
[178]: array([[ 9, 13,  6,  1, 29],
              [28,  8, 12, 22, 48],
              [30, 18, 46, 32, 24],
              [33, 11, 16,  5, 42],
              [40, 38, 20, 45, 14]])
```

[179]:
```
A * A # element-wise multiplication
```

[179]:
```
array([[  81,  169,   36,    1,  841],
       [ 784,   64,  144,  484, 2304],
       [ 900,  324, 2116, 1024,  576],
       [1089,  121,  256,   25, 1764],
       [1600, 1444,  400, 2025,  196]])
```

[180]:
```
v1 * v1
```

[180]:
```
array([ 0,  1,  4,  9, 16])
```

If we multiply arrays with compatible shapes, we get an element-wise multiplication of each row:

[181]:
```
A.shape, v1.shape
```

[181]:
```
((5, 5), (5,))
```

[182]:
```
A * v1
```

[182]:
```
array([[  0,  13,  12,   3, 116],
       [  0,   8,  24,  66, 192],
       [  0,  18,  92,  96,  96],
       [  0,  11,  32,  15, 168],
       [  0,  38,  40, 135,  56]])
```

3.1.3 Matrix algebra

What about matrix mutiplication? There are two ways. We can either use the `dot` function, which applies a matrix-matrix, matrix-vector, or inner vector multiplication to its two arguments:

[183]:
```
dot(A, A)
```

[183]:
```
array([[1818, 1442, 1082, 1797, 1477],
       [3482, 2710, 2128, 2858, 3080],
       [4170, 2626, 3504, 3138, 4518],
       [2930, 2456, 1986, 2702, 2667],
       [4069, 2211, 2616, 2371, 5550]])
```

[184]:
```
dot(A, v1)
```

[184]:
```
array([144, 290, 302, 226, 269])
```

[185]:
```
dot(v1, v1)
```

[185]:
```
30
```

Alternatively, we can cast the array objects to the type `matrix`. This changes the behavior of the standard arithmetic operators `+, -, *` to use matrix algebra.

```
[186]: M = matrix(A)
       v = matrix(v1).T # make it a column vector
```

```
[187]: v
```

```
[187]: matrix([[0],
                [1],
                [2],
                [3],
                [4]])
```

```
[188]: M * M
```

```
[188]: matrix([[1818, 1442, 1082, 1797, 1477],
                [3482, 2710, 2128, 2858, 3080],
                [4170, 2626, 3504, 3138, 4518],
                [2930, 2456, 1986, 2702, 2667],
                [4069, 2211, 2616, 2371, 5550]])
```

```
[189]: M * v
```

```
[189]: matrix([[144],
                [290],
                [302],
                [226],
                [269]])
```

```
[190]: # inner product
       v.T * v
```

```
[190]: matrix([[30]])
```

```
[191]: # with matrix objects, standard matrix algebra applies
       v + M*v
```

```
[191]: matrix([[144],
                [291],
                [304],
                [229],
                [273]])
```

If we try to add, subtract or multiply objects with incomplatible shapes we get an error:

```
[192]: v = matrix([1,2,3,4,5,6]).T
```

```
[193]: shape(M), shape(v)
```

```
[193]: ((5, 5), (6, 1))
```

```
[194]: M * M
```

```
[194]: matrix([[1818, 1442, 1082, 1797, 1477],
                [3482, 2710, 2128, 2858, 3080],
                [4170, 2626, 3504, 3138, 4518],
                [2930, 2456, 1986, 2702, 2667],
                [4069, 2211, 2616, 2371, 5550]])
```

See also the related functions: `inner`, `outer`, `cross`, `kron`, `tensordot`. Try for example `help(kron)`.

3.1.4 Array/Matrix transformations

Above we have used the `.T` to transpose the matrix object `v`. We could also have used the `transpose` function to accomplish the same thing.

Other mathematical functions that transform matrix objects are:

```
[195]: C = matrix([[1j, 2j], [3j, 4j]])
       C
```

```
[195]: matrix([[0.+1.j, 0.+2.j],
                [0.+3.j, 0.+4.j]])
```

```
[196]: conjugate(C)
```

```
[196]: matrix([[0.-1.j, 0.-2.j],
                [0.-3.j, 0.-4.j]])
```

Hermitian conjugate: transpose + conjugate

```
[197]: C.H
```

```
[197]: matrix([[0.-1.j, 0.-3.j],
                [0.-2.j, 0.-4.j]])
```

We can extract the real and imaginary parts of complex-valued arrays using `real` and `imag`:

```
[198]: real(C) # same as: C.real
```

```
[198]: matrix([[0., 0.],
                [0., 0.]])
```

```
[199]: imag(C) # same as: C.imag
```

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

Or the complex argument and absolute value

```
[182]: angle(C+1) # heads up MATLAB Users, angle is used instead of arg
```

```
[182]: array([[0.78539816, 1.10714872],
              [1.24904577, 1.32581766]])
```

```
[183]: abs(C)
```

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

3.1.5 Matrix computations

Inverse

```
[200]: linalg.inv(C) # equivalent to C.I
```

```
[200]: matrix([[0.+2.j , 0.-1.j ],
                [0.-1.5j, 0.+0.5j]])
```

```
[201]: C.I * C
```

```
[201]: matrix([[1.00000000e+00+0.j, 0.00000000e+00+0.j],
                [2.22044605e-16+0.j, 1.00000000e+00+0.j]])
```

Determinant

```
[186]: linalg.det(C)
```

```
[186]: (2.0000000000000004+0j)
```

```
[187]: linalg.det(C.I)
```

```
[187]: (0.49999999999999967+0j)
```

3.1.6 Data processing

Often it is useful to store datasets in Numpy arrays. Numpy provides a number of functions to calculate statistics of datasets in arrays.

For example, let's calculate some properties from the Stockholm temperature dataset used above.

```
[188]: # reminder, the tempeature dataset is stored in the data variable:
       shape(data)
```

```
[188]: (77431, 7)
```

mean

```
[189]: # the temperature data is in column 3
       mean(data[:,3])
```

```
[189]: 6.197109684751585
```

The daily mean temperature in Stockholm over the last 200 years has been about 6.2 C.

standard deviations and variance

```
[190]: std(data[:,3]), var(data[:,3])
```

```
[190]: (8.282271621340573, 68.59602320966341)
```

min and max

```
[191]: # lowest daily average temperature
       data[:,3].min()
```

```
[191]: -25.8
```

```
[192]: # highest daily average temperature
       data[:,3].max()
```

```
[192]: 28.3
```

sum, prod, and trace

```
[193]: d = arange(0, 10)
       d
```

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

```
[194]: # sum up all elements
       sum(d)
```

```
[194]: 45
```

```
[195]: # product of all elements
       prod(d+1)
```

[195]: 3628800

[196]:
```
# cummulative sum
cumsum(d)
```

[196]:
```
array([ 0,  1,  3,  6, 10, 15, 21, 28, 36, 45])
```

[197]:
```
# cummulative product
cumprod(d+1)
```

[197]:
```
array([      1,       2,       6,      24,     120,     720,    5040,
         40320,  362880, 3628800])
```

[198]:
```
# same as: diag(A).sum()
trace(A)
```

[198]: 82

3.1.7 Computations on subsets of arrays

We can compute with subsets of the data in an array using indexing, fancy indexing, and the other methods of extracting data from an array (described above).

For example, let's go back to the temperature dataset:

[199]:
```
!head -n 3 stockholm_td_adj.dat
```

```
1800  1  1    -6.1    -6.1    -6.1 1
1800  1  2   -15.4   -15.4   -15.4 1
1800  1  3   -15.0   -15.0   -15.0 1
```

The dataformat is: year, month, day, daily average temperature, low, high, location.

If we are interested in the average temperature only in a particular month, say February, then we can create a index mask and use it to select only the data for that month using:

[200]:
```
unique(data[:,1]) # the month column takes values from 1 to 12
```

[200]:
```
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.])
```

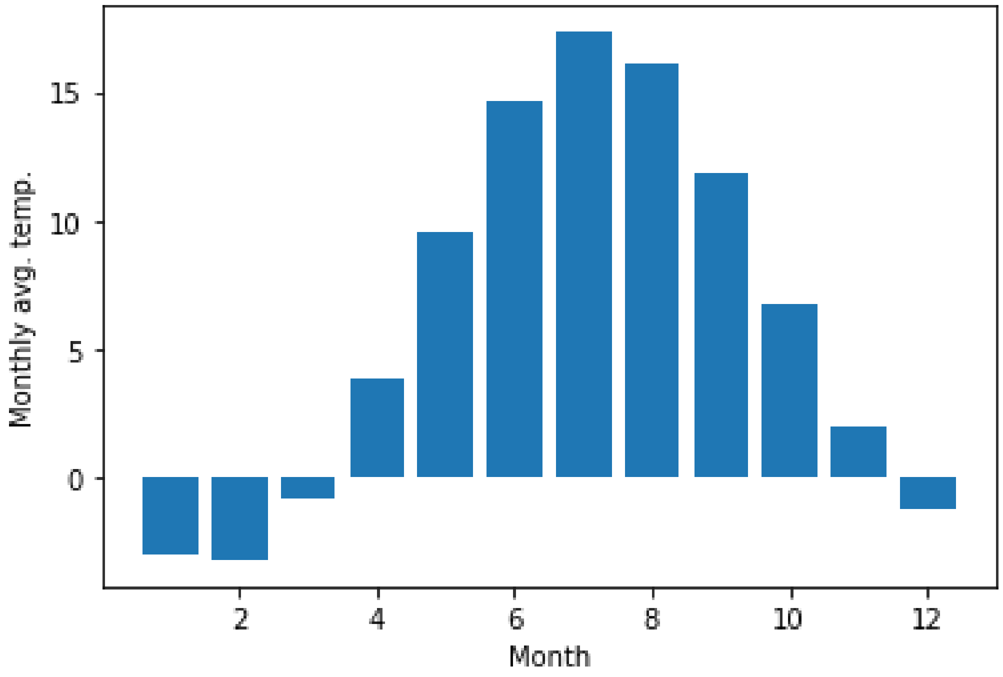
[201]:
```
mask_feb = data[:,1] == 2
```

[202]:
```
# the temperature data is in column 3
mean(data[mask_feb,3])
```

[202]: -3.212109570736596

With these tools we have very powerful data processing capabilities at our disposal. For example, to extract the average monthly average temperatures for each month of the year only takes a few lines of code:

[202]:
```
months = arange(1,13)
monthly_mean = [mean(data[data[:,1] == month, 3]) for month in months]

fig, ax = plt.subplots()
ax.bar(months, monthly_mean)
ax.set_xlabel("Month")
ax.set_ylabel("Monthly avg. temp.");
```



3.1.8 Calculations with higher-dimensional data

When functions such as `min`, `max`, etc. are applied to a multidimensional arrays, it is sometimes useful to apply the calculation to the entire array, and sometimes only on a row or column basis. Using the `axis` argument we can specify how these functions should behave:

[203]:
```
m = random.rand(3,3)
m
```

[203]:
```
array([[0.42239068, 0.83354409, 0.70591664],
       [0.77464577, 0.11659908, 0.36366418],
       [0.82343003, 0.2591544 , 0.08740126]])
```

[204]:
```
# global max
m.max()
```

[204]:
```
0.833544091636263
```

[205]:
```
# max in each column
m.max(axis=0)
```

[205]:
```
array([0.82343003, 0.83354409, 0.70591664])
```

[206]:
```
# max in each row
m.max(axis=1)
```

[206]:
```
array([0.83354409, 0.77464577, 0.82343003])
```

Many other functions and methods in the `array` and `matrix` classes accept the same (optional) `axis` keyword argument.

3.2 Reshaping, resizing and stacking arrays

The shape of an Numpy array can be modified without copying the underlaying data, which makes it a fast operation even for large arrays.

[207]:
```
A
```

[207]:
```
array([[ 9, 13,  6,  1, 29],
       [28,  8, 12, 22, 48],
       [30, 18, 46, 32, 24],
       [33, 11, 16,  5, 42],
       [40, 38, 20, 45, 14]])
```

[208]:
```
n, m = A.shape
```

[209]:
```
B = A.reshape((1,n*m))
B
```

[209]:
```
array([[ 9, 13,  6,  1, 29, 28,  8, 12, 22, 48, 30, 18, 46, 32, 24, 33,
        11, 16,  5, 42, 40, 38, 20, 45, 14]])
```

[210]:
```
B[0,0:5] = 5 # modify the array

B
```

[210]:
```
array([[ 5,  5,  5,  5,  5, 28,  8, 12, 22, 48, 30, 18, 46, 32, 24, 33,
        11, 16,  5, 42, 40, 38, 20, 45, 14]])
```

```
[211]: A # and the original variable is also changed. B is only a different view of␣
        ↪the same data
```

```
[211]: array([[ 5,  5,  5,  5,  5],
              [28,  8, 12, 22, 48],
              [30, 18, 46, 32, 24],
              [33, 11, 16,  5, 42],
              [40, 38, 20, 45, 14]])
```

We can also use the function `flatten` to make a higher-dimensional array into a vector. But this function create a copy of the data.

```
[212]: B = A.flatten()

       B
```

```
[212]: array([ 5,  5,  5,  5,  5, 28,  8, 12, 22, 48, 30, 18, 46, 32, 24, 33, 11,
              16,  5, 42, 40, 38, 20, 45, 14])
```

```
[213]: B[0:5] = 10

       B
```

```
[213]: array([10, 10, 10, 10, 10, 28,  8, 12, 22, 48, 30, 18, 46, 32, 24, 33, 11,
              16,  5, 42, 40, 38, 20, 45, 14])
```

```
[214]: A # now A has not changed, because B's data is a copy of A's, not refering to␣
        ↪the same data
```

```
[214]: array([[ 5,  5,  5,  5,  5],
              [28,  8, 12, 22, 48],
              [30, 18, 46, 32, 24],
              [33, 11, 16,  5, 42],
              [40, 38, 20, 45, 14]])
```

3.3 Adding a new dimension: newaxis

With `newaxis`, we can insert new dimensions in an array, for example converting a vector to a column or row matrix:

```
[215]: v = array([1,2,3])
```

```
[216]: shape(v)
```

```
[216]: (3,)
```

```
[217]: # make a column matrix of the vector v
       v[:, newaxis]
```

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

```
[218]: # column matrix
       v[:,newaxis].shape
```

```
[218]: (3, 1)
```

```
[219]: v
```

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

```
[220]: # row matrix
       v[newaxis,:].shape
```

```
[220]: (1, 3)
```

3.4 Stacking and repeating arrays

Using function `repeat`, `tile`, `vstack`, `hstack`, and `concatenate` we can create larger vectors and matrices from smaller ones:

3.4.1 tile and repeat

```
[226]: a = array([[1, 2], [3, 4]])
```

```
[227]: # repeat each element 3 times
       repeat(a, 3)
```

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

```
[228]: # tile the matrix 3 times
       tile(a, 3)
```

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

3.4.2 concatenate

[229]:
```
b = array([[5, 6]])
```

[230]:
```
a
```

[230]:
```
array([[1, 2],
       [3, 4]])
```

[231]:
```
concatenate((a, b), axis=0)
```

[231]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

[232]:
```
concatenate((a, b.T), axis=1)
```

[232]:
```
array([[1, 2, 5],
       [3, 4, 6]])
```

3.4.3 hstack and vstack

[230]:
```
hstack((a,b.T))
```

[230]:
```
array([[1, 2, 5],
       [3, 4, 6]])
```

[233]:
```
vstack((a,b))
```

[233]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

3.5 Copy and "deep copy"

To achieve high performance, assignments in Python usually do not copy the underlaying objects. This is important for example when objects are passed between functions, to avoid an excessive amount of memory copying when it is not necessary (technical term: pass by reference).

[234]:
```
A = array([[1, 2], [3, 4]])

A
```

[234]:
```
array([[1, 2],
       [3, 4]])
```

[235]:
```
# now B is referring to the same array data as A
B = A
```

[236]:
```
# changing B affects A
B[0,0] = 10

B
```

[236]:
```
array([[10,  2],
       [ 3,  4]])
```

[237]:
```
A
```

[237]:
```
array([[10,  2],
       [ 3,  4]])
```

If we want to avoid this behavior, so that when we get a new completely independent object `B` copied from `A`, then we need to do a so-called “deep copy” using the function `copy`:

[238]:
```
B = np.copy(A)
```

[239]:
```
# now, if we modify B, A is not affected
B[0,0] = -5

B
```

[239]:
```
array([[-5,  2],
       [ 3,  4]])
```

[240]:
```
A
```

[240]:
```
array([[10,  2],
       [ 3,  4]])
```

3.6 Iterating over array elements

Generally, we want to avoid iterating over the elements of arrays whenever we can (at all costs). The reason is that in a interpreted language like Python (or MATLAB), iterations are really slow compared to vectorized operations.

However, sometimes iterations are unavoidable. For such cases, the Python `for` loop is the most convenient way to iterate over an array:

[241]:
```
v = array([1,2,3,4])

for element in v:
    print(element)
```

```
1
2
3
4
```

[242]:
```
M = array([[1,2], [3,4]])

for row in M:
    print("row", row)

    for element in row:
        print(element)
```

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

When we need to iterate over each element of an array and modify its elements, it is convenient to use the `enumerate` function to obtain both the element and its index in the `for` loop:

[240]:
```
for row_idx, row in enumerate(M):
    print("row_idx", row_idx, "row", row)

    for col_idx, element in enumerate(row):
        print("col_idx", col_idx, "element", element)

        # update the matrix M: square each element
        M[row_idx, col_idx] = element ** 2
```

```
row_idx 0 row [1 2]
col_idx 0 element 1
col_idx 1 element 2
row_idx 1 row [3 4]
col_idx 0 element 3
col_idx 1 element 4
```

[243]:
```
# each element in M is now squared
M
```

[243]:
```
array([[1, 2],
       [3, 4]])
```

3.7 Vectorizing functions

As mentioned several times by now, to get good performance we should try to avoid looping over elements in our vectors and matrices, and instead use vectorized algorithms. The first step in converting a scalar algorithm to a vectorized algorithm is to make sure that the functions we write work with vector inputs.

[244]:
```
def Theta(x):
    """
    Scalar implemenation of the Heaviside step function.
    """
    if x >= 0:
        return 1
    else:
        return 0
```

[245]:
```
Theta(array([-3,-2,-1,0,1,2,3]))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-245-2cb2062a7e18> in <module>
----> 1 Theta(array([-3,-2,-1,0,1,2,3]))

<ipython-input-244-f72d7f42be84> in Theta(x)
      3     Scalar implemenation of the Heaviside step function.
      4     """
----> 5     if x >= 0:
      6         return 1
      7     else:

ValueError: The truth value of an array with more than one element is ambiguous ␣
 ↪Use a.any() or a.all()
```

OK, that didn't work because we didn't write the `Theta` function so that it can handle a vector input...

To get a vectorized version of Theta we can use the Numpy function `vectorize`. In many cases it can automatically vectorize a function:

[247]:
```
Theta_vec = vectorize(Theta)
```

[248]:
```
Theta_vec(array([-3,-2,-1,0,1,2,3]))
```

[248]:
```
array([0, 0, 0, 1, 1, 1, 1])
```

We can also implement the function to accept a vector input from the beginning (requires more effort but might give better performance):

[249]:
```python
def Theta(x):
    """
    Vector-aware implemenation of the Heaviside step function.
    """
    return 1 * (x >= 0)
```

[250]:
```python
Theta(array([-3,-2,-1,0,1,2,3]))
```

[250]:
```
array([0, 0, 0, 1, 1, 1, 1])
```

[251]:
```python
# still works for scalars as well
Theta(-1.2), Theta(2.6)
```

[251]:
```
(0, 1)
```

3.8 Using arrays in conditions

When using arrays in conditions,for example `if` statements and other boolean expressions, one needs to use `any` or `all`, which requires that any or all elements in the array evalutes to `True`:

[252]:
```python
M
```

[252]:
```
array([[1, 2],
       [3, 4]])
```

[253]:
```python
if (M > 5).any():
    print("at least one element in M is larger than 5")
else:
    print("no element in M is larger than 5")
```

```
no element in M is larger than 5
```

[254]:
```python
if (M > 5).all():
    print("all elements in M are larger than 5")
else:
    print("all elements in M are not larger than 5")
```

```
all elements in M are not larger than 5
```

3.9 Type casting

Since Numpy arrays are *statically typed*, the type of an array does not change once created. But we can explicitly cast an array of some type to another using the `astype` functions (see also the similar `asarray` function). This always create a new array of new type:

[255]:
```python
M.dtype
```

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

```
[256]: M2 = M.astype(float)

       M2
```

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

```
[257]: M2.dtype
```

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

```
[255]: M3 = M.astype(bool)

       M3
```

```
[255]: array([[ True,  True],
              [ True,  True]])
```

3.10 Further reading

- http://numpy.scipy.org
- http://scipy.org/Tentative_NumPy_Tutorial
- http://scipy.org/NumPy_for_Matlab_Users - A Numpy guide for MATLAB users.

```
[ ]:
```