In [1]:
```
from numpy import *
```

Vectors and matrices in Numpy

The NumPy and SciPy libraries have a rich array of functions available for creating and manipulating matrices, and for solving linear systems.

Below, we give some simple examples of creating arrays and vectors using the NumPy `array` function. We then illustrate how to extract individual entries, and some basic matrix operations.

We also give an example of solving a linear system using Gaussian Elimination with the SciPy routine `solve`.

Creating Numpy vectors and matrices

To create vectors and matrices in Numpy, we use the `array` function. Below, we use the following as examples of a 1×3 row vector $\mathbf{u}$, a 3×1 column vector $\mathbf{v}$ and a 3×3 matrix A.

$$\mathbf{u} = \begin{bmatrix} -1 & 7 & -3 \end{bmatrix} \qquad \mathbf{v} = \begin{bmatrix} 4 \\ -5 \\ 2 \end{bmatrix} \qquad A = \begin{bmatrix} 1 & 4 & -1 \\ 2 & 0 & 12 \\ 0 & -2 & 7 \end{bmatrix} \tag{1}$$

Note

There is also a NumPy `matrix` class, which behaves more like a true matrix in the linear algebra sense. However, the documentations discourages its use, and hints at its future removal.

In [2]:
```
def display_mat(msg,A):
    print(msg)
    display(A)
    print("Shape = ",A.shape)
    print("")
```

In [3]:
```
# Create row vector u
u = array([-1,7,-3])
display_mat("u = ", u)
type(u)
```

```
u =
array([-1,  7, -3])
Shape =  (3,)
```

Out[3]:
```
numpy.ndarray
```

In [4]:
```
# Create column vector v
v = array([[4],[-5],[2]])
display_mat("v = ",v)
```

```
v =
array([[ 4],
       [-5],
       [ 2]])
Shape =  (3, 1)
```

In [5]:
```
# Create matrix A : Notice the use of two sets of brackets

A = array([[1,4,-1],[2,0,12],[0,-2,7]])

display_mat("A = ",A)
```

```
A =
array([[ 1,  4, -1],
       [ 2,  0, 12],
       [ 0, -2,  7]])
Shape =  (3, 3)
```

Using Matlab-like syntax

In case you find all the extra brackets a bit tedious, you can also input arrays using a more Matlab-like syntax. For example, in Matlab, you can construct the above arrays with commands

```
u = [-1, 7, -3]
v = [4; -5; 2]
A = [1,4,-1; 2,0,12; 0,-2, 7]
```

Notice how the semi-colon ";" is used to indicate when a new row should start. In NumPy, a similar syntax can be used, along with the NumPy function `array` and `mat`.

In [6]:
```
# Create row vector u using Matlab-like syntax
u = array(mat('-1, 7, -3'))    # Notice that no brackets [] are required.
display_mat("u = ",u)
```

```
u =
array([[-1,  7, -3]])
Shape =  (1, 3)
```

In [7]:
```
# Create column vector u using Matlab-like syntax
v = array(mat('4; -5; 2'))
display_mat("v = ",v)
```

```
v =
```

```
array([[ 4],
       [-5],
       [ 2]])
Shape =  (3, 1)
```

In [8]:
```
# Create matrix A using Matlab-like syntax

A = array(mat('1,4,-1; 2,0,12; 0,-2, 7'))
display_mat("A = ",A)
```

```
A =
array([[ 1,  4, -1],
       [ 2,  0, 12],
       [ 0, -2,  7]])
Shape =  (3, 3)
```

Basic information about matrices and vectors

We can get basic information about the above matrices and vectors using attributes of NumPy arrays.

In [9]:
```
# Get the shape of an matrices and vectors

print("Shape of A \n", A.shape)   # A.shape returns a "tuple"
print("")

print("Shape of u\n", u.shape)
print("")

print("Shape of v\n", v.shape)
print("")
```

```
Shape of A
 (3, 3)

Shape of u
 (1, 3)

Shape of v
 (3, 1)
```

Matrix and vector indexing

Extracting individual entries from NumPy arrays can be done using familar `[i,j]` indexing. We can also extract subarrays using a colon ":" operator.

Reminder: Unlike Matlab, array indexing in Pytyon starts with 0.

$$A = \begin{bmatrix} 1 & 4 & -1 \\ 2 & 0 & 12 \\ 0 & -2 & 7 \end{bmatrix} \tag{2}$$

In [10]:
```
A = array(mat("1,4,-1; 2,0,12; 0,-2,7"))
display_mat("A = ",A)
```

```
A =
array([[ 1,  4, -1],
       [ 2,  0, 12],
       [ 0, -2,  7]])
Shape =  (3, 3)
```

In [11]:
```
# Extract individual entries

print(A[2,1])
```

```
-2
```

In [12]:
```
# Extract the first row of A.  Note that that extracting a row this way does nc
# result in the usual 1xN row  vector.
display_mat("A[1] = ",A[1])
```

```
A[1] =
array([ 2,  0, 12])
Shape =  (3,)
```

In [13]:
```
# A better way to extra the first row
display_mat("A[0:1,:] = ",A[0:1,:])
```

```
A[0:1,:] =
array([[ 1,  4, -1]])
Shape =  (1, 3)
```

In [14]:
```
# Extra the 2x2 subarray in the lower right corner

display_mat("A[1:,1:] = ",A[1:,1:])
```

```
A[1:,1:] =
array([[ 0, 12],
       [-2,  7]])
Shape =  (2, 2)
```

In [15]:
```
# Extract diagonal from matrix A

d = diag(A)
display_mat("diag(A) = ",d)
```

```
diag(A) =
array([1, 0, 7])
Shape =  (3,)
```

In [16]:
```
# Extract the lower triangular portion of A


```

```
L = tril(A)   # Includes the diagonal entries
print(L)
```

```
[[ 1  0  0]
 [ 2  0  0]
 [ 0 -2  7]]
```

Extracting columns of a matrix A

When using slices to extract a column of a matrix, the notation is slightly different from what you might expect.

To return a column vector, **use slices in both sets of indices**

In [17]:
```
# Wrong way to extract a column vector.  This returns a row vector.
# The values are correct, but the shape of the array is not convenient. 1º

c = A[:,1]   # "slice" (":") used only in the first dimension
display_mat("A[:,1] = ", c)
display_mat("A[:,1].T = ",c.T)
```

```
A[:,1] =
array([ 4,  0, -2])
Shape =  (3,)

A[:,1].T =
array([ 4,  0, -2])
Shape =  (3,)
```

In [18]:
```
# Right way to extract a column vector.

c = A[:,1:2]   # "slices" (":") used in both directions.
display_mat("c = ",c)
```

```
c =
array([[ 4],
       [ 0],
       [-2]])
Shape =  (3, 1)
```

In [19]:
```
# However, trying to print the first component can lead to an error.
display_mat("c[0] = ",c[0])
```

```
c[0] =
array([4])
Shape =  (1,)
```

Indexing column vectors

One tricky aspect of using column vectors is that we have to use two indices to get the entries. This is because a column vector is really a "vector of vectors".

In [20]:
```
# This returns the first entry in [[1],[2],[3]].  This entry is itself an array
display_mat("v[0] = \n",v[0])
```

```
v[0] =

array([4])
Shape =  (1,)
```

Notice the `[]` in the result. This indicates that the result is itself an array.

Problems an occur if you try to print out what you think is just the first entry of the vector v. For example, this will fail :

```
print("{:f}".format(v[0]))
```

```
------------------------------------------------------------
TypeError                                 Traceback (most
recent call last)
<ipython-input-35-270cd991ee18> in <module>
----> 1 print("{:f}".format(v[0]))

TypeError: unsupported format string passed to
numpy.ndarray.__format__
```

Instead, we must use two indices :

In [21]:
```
print("{:f}".format(v[0,0]))
```

```
4.000000
```

Matrix and vector multiplication

Recall that matrices and vectors can be multiplied if they are *compatible*. That is, we can multiple A times v on the right, since A and 3 columns and $\mathbf{v}$ has 3 rows. We can also multiply A by $\mathbf{u}$ on the left.

$$A\mathbf{v} = \begin{bmatrix} 1 & 4 & -1 \\ 2 & 0 & 12 \\ 0 & -2 & 7 \end{bmatrix} \begin{bmatrix} 4 \\ -5 \\ 2 \end{bmatrix} = \ldots \qquad \mathbf{u}A = \begin{bmatrix} -1 & 7 & -3 \end{bmatrix} \begin{bmatrix} 1 & 4 & -1 \\ 2 & 0 & 12 \\ 0 & -2 & 7 \end{bmatrix} = \ldots$$

In [22]:
```
# Multiply Av
v = array(mat('4;-5;2'))

display_mat("A = ",A)
display_mat("Av = ", A@v)
```

```
A =
array([[ 1,  4, -1],
       [ 2,  0, 12],
       [ 0, -2,  7]])
```

```
Shape =  (3, 3)

Av =
array([[-18],
       [ 32],
       [ 24]])
Shape =  (3, 1)
```

In [23]:
```
# multiply uA

display_mat("uA = ", u@A)
```

```
uA =
array([[13,  2, 64]])
Shape =  (1, 3)
```

Inner and outer products

We can compute the following vector "inner products" (resulting in a scalar) and vector "outer products" (resulting in a matrix)

$$\mathbf{uv} = \begin{bmatrix} -1 & 7 & -3 \end{bmatrix} \begin{bmatrix} 4 \\ -5 \\ 2 \end{bmatrix} = \ldots \qquad \mathbf{vu} = \begin{bmatrix} 4 \\ -5 \\ 2 \end{bmatrix} \begin{bmatrix} -1 & 7 & -3 \end{bmatrix} = \ldots \quad (4)$$

In [24]:
```
# print uv

display_mat("uv = ",u@v)    # Note that the result is an array
```

```
uv =
array([[-45]])
Shape =  (1, 1)
```

In [25]:
```
# print vu
display_mat("vu = ",v@u)
```

```
vu =
array([[ -4,  28, -12],
       [  5, -35,  15],
       [ -2,  14,  -6]])
Shape =  (3, 3)
```

Matrix transpose

We can also use a vector transpose to construct products $v^T A$, Au^T and so on.

$$\mathbf{v}^T A = \begin{bmatrix} 4 & -5 & 2 \end{bmatrix} \begin{bmatrix} 1 & 4 & -1 \\ 2 & 0 & 12 \\ 0 & -2 & 7 \end{bmatrix} = \ldots \qquad A\mathbf{u}^T = \begin{bmatrix} 1 & 4 & -1 \\ 2 & 0 & 12 \\ 0 & -2 & 7 \end{bmatrix} \begin{bmatrix} -1 \\ 7 \\ -3 \end{bmatrix} = .$$

In [26]:
```
# Product v^T A
```

```
display_mat("V^TA = ",v.T@A)
```

```
V^TA =
array([[ -6,  12, -50]])
Shape =  (1, 3)
```

In [27]:
```
# Product A u^T

display_mat("Au^T = ",A@u.T)
```

```
Au^T =
array([[ 30],
       [-38],
       [-35]])
Shape =  (3, 1)
```

Special vectors matrices

We can construct specialized vectors and matrices, as below

In [28]:
```
# Row vector of all zeros

display_mat("zeros(5) = ",zeros(5))
```

```
zeros(5) =
array([0., 0., 0., 0., 0.])
Shape =  (5,)
```

In [29]:
```
# Column vector of all ones.  Note use of a 'tuple' (5,1) to specify the shape

display_mat("ones((5,1)) = ",ones((5,1)))
```

```
ones((5,1)) =
array([[1.],
       [1.],
       [1.],
       [1.],
       [1.]])
Shape =  (5, 1)
```

In [30]:
```
# Adding two matrices together

B = ones((5,5))
C = zeros((5,5))
display_mat("B = ", B)

display_mat("C = ",C)

display_mat("2C + 5D = ",2*B + C)
```

```
B =
```

```
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.]])
Shape =  (5, 5)

C =
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.]])
Shape =  (5, 5)

2C + 5D =
array([[2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2.]])
Shape =  (5, 5)
```

In [31]:
```
# 5x5 identity matrix

I = eye(5)
display_mat("eye(5) = ",eye(5))
```

```
eye(5) =
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
Shape =  (5, 5)
```

In [32]:
```
# Diagonal matrix constructed from a vector of values

d = array([1,2,3,4,5])

display_mat("diag(d) = ",diag(d))
```

```
diag(d) =
array([[1, 0, 0, 0, 0],
       [0, 2, 0, 0, 0],
       [0, 0, 3, 0, 0],
       [0, 0, 0, 4, 0],
       [0, 0, 0, 0, 5]])
Shape =  (5, 5)
```

Gaussian Elimination

A common way to solve linear systems by hand is to use Gaussian Elimination. In this approach, variables are eliminated from the system, one by one, until only a scalar equation

remains. This scalar equation is solved, and the values are then substituted back into the remaining equations until the linear system is solved. Solving 2×2 and 3×3 systems by hand can be a bit tedious, but mostly straightforward. This procedure can be carried out with very few lines of code on a computer, and is implemented in most scientific libraries.

In SciPy, the routine that carries out Gaussian Elimination is the `scipy.linalg.solve` routine. This routine is essentially equivalent to Matlab's `A\b`. To see how it works, we can solve the following system

$$\begin{bmatrix} 1 & 4 & -1 \\ 2 & 0 & 12 \\ 0 & -2 & 7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ -5 \\ 2 \end{bmatrix} \tag{6}$$

for the vector of values $\mathbf{x} = (x_1, x_2, x_3)^T$.

In [33]:
```
from numpy.linalg import solve

# Use the Matlab-like syntax to construct matrix A and vector b
A = array(mat('1, 4, -1; 2, 0, 12; 0, -2, 7'))
b = array(mat('4; -5; 2'))

# Solve the linear system
x = solve(A,b)

display_mat("Solution to Ax = b : ", x)
```

```
Solution to Ax = b :
array([[-11.5 ],
       [  4.25],
       [  1.5 ]])
Shape =  (3, 1)
```

We can check the this solution by computing a residual $\mathbf{r} = \mathbf{b} - A\mathbf{x}$.

In [34]:
```
r = b - A@x
display_mat("Residual r = b - Ax : ", r)
```

```
Residual r = b - Ax :
array([[0.],
       [0.],
       [0.]])
Shape =  (3, 1)
```

For smaller matrices, or "dense" (not many zeros) matrices, Gaussian Elimination is often a good choice. For "sparse" matrices, there are often better choices.

Vector and matrix norms

We often need a way to compare the size of one vector or matrix to another. When working with scalars, we use the absolute value $|\cdot|$. When working with vectors and matrices, we

use a "norm".

Vector norms

One way to do this is by comparing their lengths. For two and three dimensional vectors, we are familiar with the notion of the geometric length of a vector. For example, we have

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \tag{7}$$

then the length easily calculated as $\sqrt{v_1^2 + v_2^2} = \sqrt{1+4} = \sqrt{5}$. This idea extends naturally to length N arrays (although it becomes more difficult to imagine the length as having geometric meaning. Given

$$\mathbf{v} = (v_1, v_2, v_3, \ldots, v_N)^T \tag{8}$$

we can compute the length as

$$\sqrt{\sum_{k=1}^{N} v_k^2} \tag{9}$$

For scalar values, we use $|\cdot|$ to denote the "length" of a scalar value (i.e. its distance from the origin). For vectors, we use analogous notation, $\|\cdot\|$. The above "length" is called the "2-norm" and is denoted

$$\|\mathbf{v}\|_2 = \sqrt{\sum_{k=1}^{N} v_k^2} \tag{10}$$

Other norms are useful as well. We can simply add each component in a vector. This norm is called the "1-norm".

$$\|\mathbf{v}\|_1 = \sum_{k=1}^{N} |v_k| \tag{11}$$

The absolute value is necessary so that the 1-norm is only zero if each component is zero.

The "inf-norm" is the maximum value of the vector,

$$\|\mathbf{v}\|_\infty = \max_{1 \leq k \leq N} |v_k| \tag{12}$$

In Numpy, we have to explicitly import the norm functions for the Numpy submodule `linalg`.

```
from numpy.linalg import norm
```

In [35]:
```
from numpy.linalg import norm
```

In [36]:
```
v = array(mat('1,2,3,4,5'))
display_mat("norm(v,1) = ", norm(v,1))      # largest column sum
display_mat("norm(v,2) = ", norm(v,2))      # sqrt of sum of squares of entries
display_mat("norm(v,inf) = ", norm(v,inf))  # largest row sum
```

```
norm(v,1) =
5.0
Shape =  ()

norm(v,2) =
7.416198487095664
Shape =  ()

norm(v,inf) =
15.0
Shape =  ()
```

Matrix norms

We can also derive a meaningful notion of a "norm" for matrices by considering how a matrix "stretches" or "contracts" a vector. Suppose we have a vector norm $\|\mathbf{x}\|_p$, for $p = 1, 2, \infty$. Then for a given matrix A, we can compute

$$\|A\mathbf{x}\|_p \tag{13}$$

This doesn't yet give us anything meaningful because the value of $\|A\mathbf{x}\|_p$ will depend on the value of $\mathbf{x}$. But this idea can be used to define a matrix norm in terms of a vector norm as

$$\|A\|_p = \max_{\mathbf{x}} \frac{\|A\mathbf{x}\|_p}{\|\mathbf{x}\|_p} \tag{14}$$

or

$$\|A\|_p = \max_{\|\mathbf{x}\|_p=1} \|A\mathbf{x}\|_p \tag{15}$$

This seems like it is impossible to compute, but in fact, we can compute these norms for $p = 1, \infty$ quite easily as follows

In [ ]:
```
```

1-norm : Maximum column sum

$$\|A\|_1 = \max_{0 \le j \le N} \sum_{i=1}^{N} |a_{ij}| \tag{16}$$

That is, sum up the absolute values of entries in each column. Then take the maximum "column sum".

∞-norm : Maximum row sum

$$\|A\|_\infty = \max_{0 \le i \le N} \sum_{j=1}^{N} |a_{ij}| \tag{17}$$

That is, sum up the absolute values of entries in each row. Then take the maximum "row sum".

2-norm and the Frobenius norm

The 2-norm is more difficult to compute, and so often, we replace a 2-norm with the Frobenius norm, defined as

$$\|A\|_F = \sqrt{\sum_{i=1}^{N} \sum_{i=j}^{N} |a_{ij}|^2} \tag{18}$$

What we can show is that

$$\|A\|_2 \ \le \ \|A\|_F \tag{19}$$

The 2-norm is connected to the singular values of a matrix, and can be defined as

$$\|A\|_2 = \sqrt{\sum_{i=1}^{r} \sigma_i^2} \tag{20}$$

where r is the rank of A.

Some pitfalls when working with NumPy arrays

Below are some common mistakes when working with NumPy arrays.

Indexing columns and rows

We have to be careful when extracting rows and columns from a matrix. In most cases, we want the shape of our resulting row to be $1 \times N$ and the column to be $N \times 1$. This is

especially important if we plan to involve the resulting row or column by another matrix operation.

In [37]:
```
A = array(mat("1,2,3; 3,4,5; 5,6,7"))
display_mat("A = ", A)
```

```
A =
array([[1, 2, 3],
       [3, 4, 5],
       [5, 6, 7]])
Shape =  (3, 3)
```

You might notice that the matrix `A` is stored as an "array of arrays". You might think that extract the first row could be extracted with a single index, as in

In [38]:
```
display_mat("First row of A : A[0] = ",A[0])
```

```
First row of A : A[0] =
array([1, 2, 3])
Shape =  (3,)
```

The shape of this array, however, is not 1×3, but rather is a one-dimensional NumPy array with "length" 3.

To get a true row vector, with the proper shape, we need to use NumPy "slices" and double indexing to extract the row.

In [39]:
```
display_mat("First row of A : A[0:1,:] = ",A[0:1,:])
```

```
First row of A : A[0:1,:] =
array([[1, 2, 3]])
Shape =  (1, 3)
```

We can extract columns in an analogous fashion.

In [40]:
```
display_mat("First column of A : A[:,0:1] = ",A[:,0:1])
```

```
First column of A : A[:,0:1] =
array([[1],
       [3],
       [5]])
Shape =  (3, 1)
```

Pitfall : Difference between `@` and `*`

When involving matrices in numerical calculations, we have the option of using either `@` or `*`. In most linear algebra contexts, we are interested in using the `@` operator to get multiplication in the proper linear algebra sense. However, the `*` operator will also often work but can give us unexpected results (which are usually not what we want).

In general terms,

- use `@` when you want to do matrix-matrix, matrix-vector, or vector-vector mulplication. In this case, multiplying a $m \times p$ matrix on the right by a $p \times n$ matrix will result in a $m \times n$ matrix, acording to the rules of matrix multiplication in linear algebra.

- use `*` if you want elementwise multiplication. This is roughly equivalent to the Matlab "dot" operator `.*`. In this case, the product of two $N \times 1$ vectors is another $N \times 1$ vector whose entries are the result of multiplying corresponding entries of each vector.

In the following example, we multiply a matrix `A` by a vector `v` using both `@` and `*`. Can you see the difference between the two operations?

In [41]:
```
A = ones((3,3))
display_mat("A = ",A)
v = array(mat('1;2;3'))
display_mat("v = ",v)


display_mat("A@v = ",A@v)
display_mat("A*v = ",A*v)
```

```
A =
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
Shape =  (3, 3)

v =
array([[1],
       [2],
       [3]])
Shape =  (3, 1)

A@v =
array([[6.],
       [6.],
       [6.]])
Shape =  (3, 1)

A*v =
array([[1., 1., 1.],
       [2., 2., 2.],
       [3., 3., 3.]])
Shape =  (3, 3)
```

The `@` operator really gives us what we want in the linear algebra sense.

The `*` operator multiplies, in an element-wise way, each column of `A` by the vector `v`. The `*` operator is close to the Matlab dot operator `*.`.

Here is another example of "elementwise" operation using `@` and `*` with vectors.

In [42]:
```
v = reshape(arange(5),(5,1))
```

```
display_mat('v = ', v)

w = reshape(arange(2,7),(5,1))
display_mat('w = ', w)

display_mat('v*w (elementwise multiplication) = ',v*w)
display_mat('v.T@w (inner product) = ',v.T@w)
```

```
v =
array([[0],
       [1],
       [2],
       [3],
       [4]])
Shape =  (5, 1)

w =
array([[2],
       [3],
       [4],
       [5],
       [6]])
Shape =  (5, 1)

v*w (elementwise multiplication) =
array([[ 0],
       [ 3],
       [ 8],
       [15],
       [24]])
Shape =  (5, 1)

v.T@w (inner product) =
array([[50]])
Shape =  (1, 1)
```

To use `@` with `v` and `w`, we have to take the transpose of one of the vectors. In this case, we computed an inner product as $\mathbf{v}^T\mathbf{w}$.

Had we tried `v@w`, we would have gotten the following error :

```
ValueError: matmul: Input operand 1 has a mismatch in its core
dimension 0, with gufunc signature (n?,k),(k,m?)->(n?,m?) (size
1 is different from 5)
```

The message is not very helpful, but tells us that we are not allowed to multiply a 1×5 vector by another 1×5 vector in the linear algebra sense, because the inner dimensions do not agree.

Pitfall : Make sure arrays are of type `float`

Python decides whether to create a variable of type `int` verses of type `float` depending on how the constants are set. In the following example, Python assumes that matrix `A` is of type `int`, because none of the constants used to assign values have a decimal in the constant. This can create problems later when trying to assign floats to integers, because Python will round values to the nearest integer.

In most cases in numerical linear algebra, we are interested in matrices with floating point values.

In [43]:
```
# Create an array using constants that don't involve any decimal, e.g. "1" vs.
# The result will be a matrix of type `int`
A = array(mat("1,2; 3,4; 5,6"))
display_mat("A (before) = ",A)
A.dtype   # Type integer
```

```
A (before) =
array([[1, 2],
       [3, 4],
       [5, 6]])
Shape =  (3, 2)
```

Out[43]:
```
dtype('int64')
```

Now, when we try to assign float values to entries in the matrix `A`, these float values will be rounded to the nearest integer.

In [44]:
```
# Replace second column with float values
A[:,1:2] = array(mat("0.005;-1.45;5.1"))
display_mat("A (after) = ",A)
```

```
A (after) =
array([[ 1,  0],
       [ 3, -1],
       [ 5,  5]])
Shape =  (3, 2)
```

The values in the second column have all been rounded to the nearest integer values. To fix this, we need to be sure that are array is created with the proper type. We can explicitly set the type using the `dtype` argument.

In [45]:
```
# "float" is a double in Matlab or C
A = array(mat("1,2; 3,4; 5,6"),dtype='float')

display_mat("A (float type) = ",A)
A.dtype   # Type float64
```

```
A (float type) =
array([[1., 2.],
       [3., 4.],
       [5., 6.]])
Shape =  (3, 2)
```

Out[45]:
```
dtype('float64')
```

Now, we can reliably replace values in our original matrix to floating point values.

In [46]:
```
A[:,1:2] = array(mat("0.005;-1.45;5.1"))
display_mat("A (after) = ", A)
```

```
A (after) =
array([[ 1.00e+00,  5.00e-03],
       [ 3.00e+00, -1.45e+00],
       [ 5.00e+00,  5.10e+00]])
Shape =  (3, 2)
```

Alternatively, we can rely on Python's implicit typing assumptions and be sure to include a decimal in at least one of the constants used to define the entries in `A`. This is enough to tell Python to create an array of floats rather than of integers.

In [47]:
```
# Include a decimal in at least one entry, e.g. "2." instead of "2".
A = array(mat("1,2.; 3,4; 5,6"))

display_mat("A (float type) = ",A)
A.dtype   # Type float64
```

```
A (float type) =
array([[1., 2.],
       [3., 4.],
       [5., 6.]])
Shape =  (3, 2)
```

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

Pitfall : Copies vs. Views

When copying matrices, Numpy will, in some cases, only create a "view" into the original matrix. So any changes made to the view will also affect the original matrix.

In [48]:
```
A = array(mat('1,2,3; 4,5,6; 7,8,9'),dtype='int64')
display_mat("A (before) = ", A)

B = A
display_mat("B (view of A) = ", B)
```

```
A (before) =
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
Shape =  (3, 3)

B (view of A) =
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
Shape =  (3, 3)
```

Now you might think that if we change a value in `B`, that `A` will remain unchanged. But since `B` is only a view into `A`, changing `B` will also change `A`.

In [49]:
```
B[1,1] = -9999.00

display_mat("B (after) = ", B)

display_mat("A (after) = ", A)
```

```
B (after) =
array([[    1,     2,     3],
       [    4, -9999,     6],
       [    7,     8,     9]])
Shape =  (3, 3)

A (after) =
array([[    1,     2,     3],
       [    4, -9999,     6],
       [    7,     8,     9]])
Shape =  (3, 3)
```

To prevent this from happening, we should make a copy of `A`, not just a view. That way, when we change `B`, `A` will remain unchanged.

In [50]:
```
A = array(mat('1,2,3; 4,5,6; 7,8,9'),dtype='int64')
display_mat("A (before) = ", A)

B = A.copy()
display_mat("B (view of A) = ", B)

B[1,1] = -9999.00

display_mat("B (after) = ", B)

display_mat("A (after) = ", A)
```

```
A (before) =
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
Shape =  (3, 3)

B (view of A) =
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
Shape =  (3, 3)

B (after) =
array([[    1,     2,     3],
       [    4, -9999,     6],
       [    7,     8,     9]])
Shape =  (3, 3)

A (after) =
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
Shape =  (3, 3)
```

In [ ]: