# Matrices and NumPy

Matrices are used extensively in machine learning, so it is important that you know what they are and how to use them in Python.

## Basic definitions

A **matrix** $\mathbf{A}$ is an array of numbers, usually presented in the form:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

- The $m$ horizontal lists of numbers are called the **rows** of $\mathbf{A}$.
- The $n$ vertical lists of numbers are called the **columns** of $\mathbf{A}$.
- A matrix is usually represented by a capital letter in bold, e.g. $\mathbf{A}$ or $\mathbf{X}$.

To create the matrix

$$A = \begin{bmatrix} 2 & 1 & 3 \\ -4 & 5 & 7 \end{bmatrix}$$

in Python, we can use the code in the cell below.

In [1]:
```
import numpy as np
A = np.array([[2,1,3],[-4,5,7]])

print(A)
```

```
[[ 2  1  3]
 [-4  5  7]]
```

### Elements of a matrix

- $a_{ij}$ is used to represent the $ij$-th entry in a matrix $\mathbf{A}$, i.e. the number that appears in row $i$ and column $j$.
- In Python, we can access an individual element of a matrix as follows:

In [2]:
```
print("a_22 =", A[1,1])
```

```
a_22 = 5
```

Note that indexing starts at 0 in Python. Therefore A[0,0] is the number in the first row and first column of $\mathbf{A}$, i.e. $a_{11}$ . This can be annoying at first, but you get used to it!

If you want to be fancy and print subscripts to screen, you can use unicode characters. For example:

In [3]:
```
print("a\u2082\u2082 =", A[1,1])
```

```
a₂₂ = 5
```

or:

In [4]:
```
print("a₂₂ =", A[1][1])
```
```
a₂₂ = 5
```

The index `-1` is a shortcut for the last index, e.g.:

In [5]:
```
print("A[0,-1] =", A[0][-1]) #row 0 (i.e. first row), last column
print("A[-1,-1] =", A[-1][-1]) #last row, last column
```
```
A[0,-1] = 4
A[-1,-1] = 7
```

We can also print out a specific row:

In [6]:
```
# Print the second row of the matrix
print('Row two of A')
print(A[1,:])
```
```
Row two of A
[-4  5  7]
```

or specific elements of a row:

In [7]:
```
# Print the first two elements of the second row of the matrix
# Notice that we use 0:2 to refer to the first two columns, i.e. indices 0 and 1
print('First two elements of Row two of A')
print(A[1,0:2])
```
```
First two elements of Row two of A
[-4  5]
```

And of course, we can do this for columns as well.

For more info on 'slicing' NumPy arrays, a good reference (and in the context of machine learning) is:

https://machinelearningmastery.com/index-slice-reshape-numpy-arrays-machine-learning-python/ (https://machinelearningmastery.com/index-slice-reshape-numpy-arrays-machine-learning-python/)

#### Order of a matrix

- The **order** of a matrix with $m$ rows and $n$ columns is $m \times n$ (read '$m$ by $n$')
  - So, for example, the order of matrix $\mathbf{A}$ defined above is $2 \times 3$ (we do not multiply 2 by 3).
- In Python, we can use the `shape` function to determine the order of a matrix.

In [8]:
```
print('The order of matrix A is',A.shape)
```
```
The order of matrix A is (2, 3)
```

#### Special matrices

##### *Vectors*

A vector is a matrix with only one row, or only one column.

- A vector with only one row is called a **row vector**
- A vector with only one column is called a **column vector**

We may be tempted to create a vector $\mathbf{v}$ just as an array as follows (note, names of vectors are usually *lower-case* and bold):

In [9]:
```
v = np.array([1,2,3])
print('v =\n',v)
```

```
v =
 [1 2 3]
```

However, let's check the order of $\mathbf{v}$:

In [10]:
```
print(v.shape)
```

```
(3,)
```

Note that the shape of $\mathbf{v}$ is stated as (3,).

If $\mathbf{v}$ is created as above, it can be treated as either a $3 \times 1$ or $1 \times 3$ vector. However, it is important that we know which it actually is (e.g. for matrix multiplication, covered below).

So instead we can create a row vector as follows:

In [3]:
```
# Create a 1 x 3 row vector
w = np.array([[1,2,3]])
print('w =\n',w)
print(w.shape)
```

```
w =
 [[1 2 3]]
(1, 3)
```

And we can create a column vector as follows (however, see the section on transpose for a more convenient way):

In [12]:
```
# Create a 3 x 1 column vector
x = np.array([[1],[2],[3]])
print('\nx =\n',x)
print(x.shape)
```

```

x =
 [[1]
 [2]
 [3]]
(3, 1)
```

***Square matrices***

A **square** matrix has the same number of rows as columns. For example:

$$F = \begin{bmatrix} 5 & 6 & 1 \\ -4 & 5 & -3 \\ 0 & 3 & 2 \end{bmatrix}$$

is a square matrix, because it has 3 rows and 3 columns.

In [13]:

```python
F = np.array([[5,6,1],[-4,5,-3],[0,3,2]])
print(F)
print(F.shape)
```

```
[[ 5  6  1]
 [-4  5 -3]
 [ 0  3  2]]
(3, 3)
```

In a square matrix, the diagonal of numbers from top left to bottom right is called the **leading diagonal** (or the **principal diagonal**).

***Identity matrix***

The **identity matrix** is a square matrix which has ones along the leading diagonal, and zeroes everywhere else. The $n \times n$ identity matrix is denoted by $\mathbf{I}_n$. For example:

$$\mathbf{I}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

We can create an identity matrix in Python using the following code:

In [14]:

```python
# Create a 3 x 3 identity matrix
I = np.eye(3)
print('\nI =\n',I)
```

```

I =
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

***Zero matrix***

The **zero matrix** (or **null matrix**) is a matrix with every element zero. It does not need to be square.

We can create a zero matrix in Python using the following code:

In [15]:

```python
# Create a 5 x 3 matrix of zeros
B = np.zeros((5,3))
print('B =\n',B)
```

```
B =
 [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

***Matrix of ones***

A **matrix of ones** (or **all-ones matrix**) is a matrix where every element is equal to one. It is also sometimes called a **unit matrix**.

We can create a matrix of ones in Python as follows:

In [16]:

```
# Create a 3 x 2 matrix of ones
C = np.ones((3,2))
print('\nC =\n',C)
```

```

C =
 [[1. 1.]
 [1. 1.]
 [1. 1.]]
```

***Other matrices***

We can create a matrix of random numbers using:

In [17]:

```
# Create a 3 x 3 random matrix
D = np.random.random((3,3))
print('\nD =\n',D)
```

```

D =
 [[0.17966605 0.20106432 0.04518572]
 [0.50228958 0.23299641 0.99896609]
 [0.17789557 0.78684618 0.77343693]]
```

We can create a constant matrix, where every element is equal, using:

In [18]:

```
E = np.full((2,4), 5)
print('\nE =\n',E)
```

```

E =
 [[5 5 5 5]
 [5 5 5 5]]
```

## Matrix operations

**Matrix transpose**

The transpose of a matrix $\mathbf{A}$, written $\mathbf{A}^T$, is the matrix obtained by writing the rows of $\mathbf{A}$ (in order from top to bottom) as columns (from left to right). So if $\mathbf{A}$ is a $m \times n$ matrix, then $\mathbf{A}^T$ is a $n \times m$ matrix.

For example, the transpose of $\mathbf{A} = \begin{bmatrix} 2 & 1 & 3 \\ -4 & 5 & 7 \end{bmatrix}$ is $\mathbf{A}^T = \begin{bmatrix} 2 & -4 \\ 1 & 5 \\ 3 & 7 \end{bmatrix}$

- The first row of $\mathbf{A}$ is the first column of $\mathbf{A}^T$
- The second row of $\mathbf{A}$ is the second column of $\mathbf{A}^T$.

The cell below shows how to take the transpose of a matrix in Python:

In [19]:

```python
# Obtain the transpose of a matrix
print('Transpose of A')
print(A.T)
```

```
Transpose of A
[[ 2 -4]
 [ 1  5]
 [ 4  7]]
```

Transpose gives us a convenient way to create column vectors:

In [20]:

```python
# Alternative way to create a 3 x 1 column vector using transpose:
y = np.array([[1,2,3]]).T
print('\ny =\n',y)
print(y.shape)
```

```

y =
 [[1]
 [2]
 [3]]
(3, 1)
```

**Addition and subtraction of matrices**

Two matrices $\mathbf{A}$ and $\mathbf{B}$ can only be added / subtracted if they have the same order.

The sum of $\mathbf{A}$ and $\mathbf{B}$, written as $\mathbf{A}+\mathbf{B}$, is the matrix obtained by adding corresponding elements from $\mathbf{A}$ and $\mathbf{B}$.

- So if $\mathbf{C}=\mathbf{A}+\mathbf{B}$, then $c_{ij}=a_{ij}+b_{ij}$

Similarly, $\mathbf{A}-\mathbf{B}$ is the matrix obtained by subtracting the elements of $\mathbf{B}$ from the corresponding elements of $\mathbf{A}$.

- So if $\mathbf{D}=\mathbf{A}-\mathbf{B}$, then $d_{ij}=a_{ij}-b_{ij}$

$\mathbf{C}$ has the same order as $\mathbf{A}^T$, so they can be added together as follows:

In [21]:

```python
# Add C to the transpose of A
print('\nC + A.T')
print(C + A.T)
```

```

C + A.T
[[ 3. -3.]
 [ 2.  6.]
 [ 5.  8.]]
```

In Python, we can restrict operations such as addition to specific elements of a matrix. For example:

In [22]:

```python
# Add 1 to each element in the first two columns and then print the updated matrix
A[:,0:2] = A[:,0:2] + 1
print('Updated matrix')
print(A)
```

```
Updated matrix
[[ 3  2  4]
 [-3  6  7]]
```

**Scalar multiplication of a matrix**

To multiply a matrix $\mathbf{A}$ by a scalar $k$, we simply multiply each element of $\mathbf{A}$ by $k$.

In the cell below we calculate $5\mathbf{C}$:

In [23]:
```
# Scalar multiplication
print('\n5C')
print(5*C)
```

```

5C
[[5. 5.]
 [5. 5.]
 [5. 5.]]
```

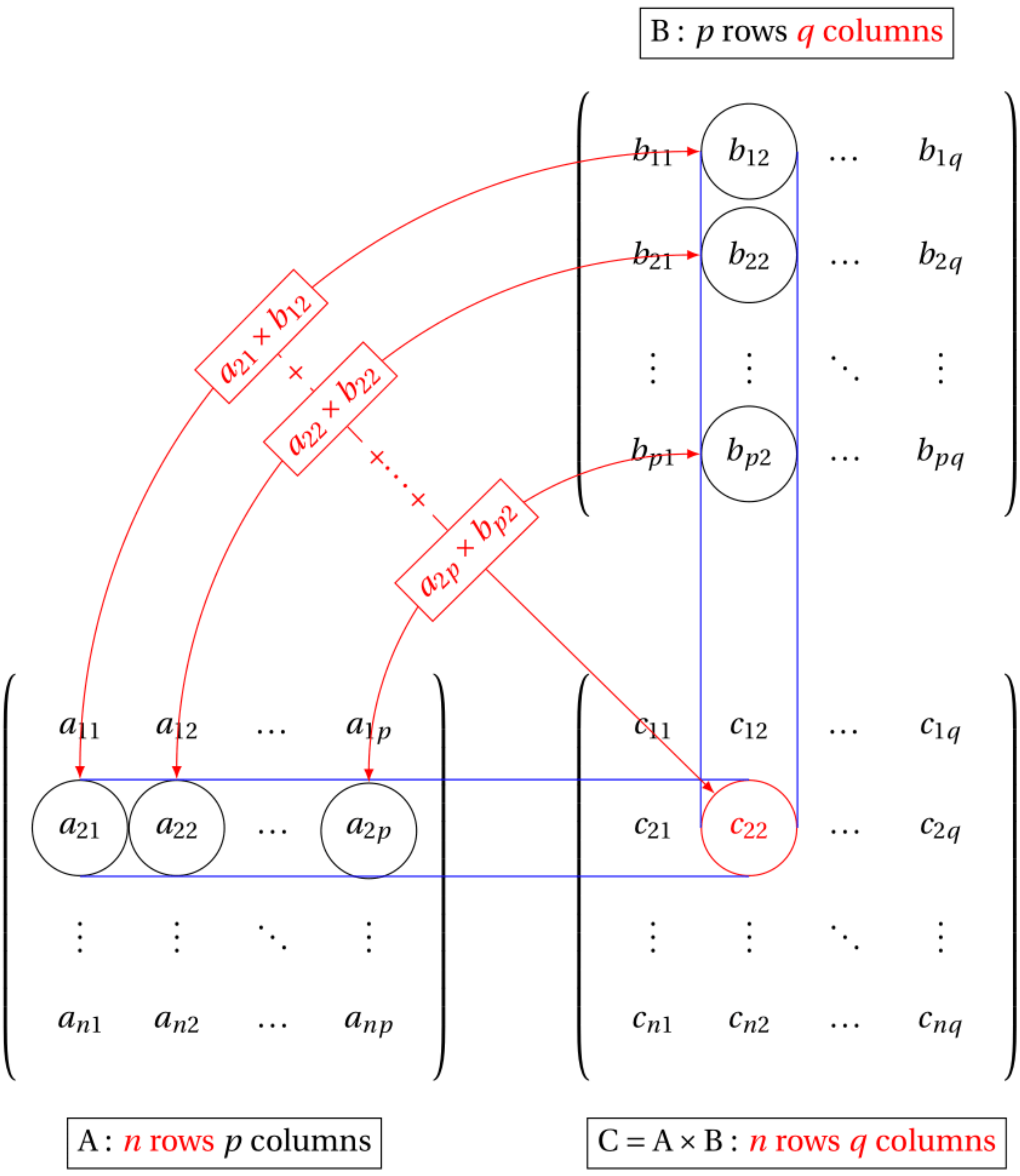
**Matrix Multiplication and Powers**

Consider the $n \times p$ matrix $\mathbf{A}$, and the $r \times q$ matrix $\mathbf{B}$:

- The multiplication $\mathbf{AB}$ can only be carried out if $p = r$. That is, if the number of columns of $\mathbf{A}$ equals the number of rows of $\mathbf{B}$.
- The order of the resulting matrix is $n \times q$.

If the order of $\mathbf{A}$ is $n \times p$ and the order of $\mathbf{B}$ is $p \times q$, then the product $\mathbf{AB}$ gives a $n \times q$ matrix $\mathbf{C}$ whose elements are given by $c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}$

In words, to get the element in row $i$ and column $j$ of matrix $\mathbf{C} = \mathbf{AB}$, work along row $i$ of matrix $\mathbf{A}$, and down column $j$ of matrix $\mathbf{B}$, adding together the products of corresponding elements. This is illustrated below.



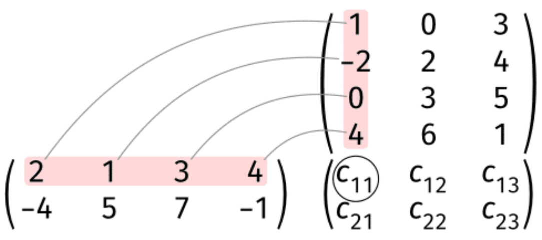
Note, matrix multiplication is **non-commutative**: this means in general $\mathbf{AB} \neq \mathbf{BA}$. In fact, one multiplication may be possible where the other may not be.

**Worked example of matrix multiplication**

Calculate $\mathbf{C} = \mathbf{AB}$, where $\mathbf{A} = \begin{bmatrix} 2 & 1 & 3 & 4 \\ -4 & 5 & 7 & -1 \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} 1 & 0 & 3 \\ -2 & 2 & 4 \\ 0 & 3 & 5 \\ 4 & 6 & 1 \end{bmatrix}$

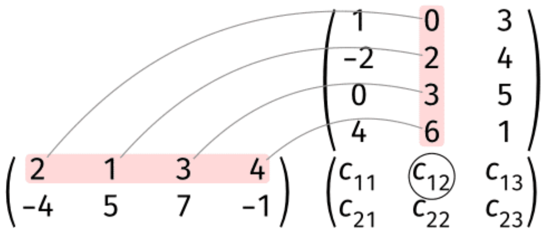
The entry $c_{11}$ in the *first* row and *first* column of $\mathbf{C} = \mathbf{AB}$ is given by summing the products of the entries in the *first* row of $\mathbf{A}$ with the entries of the *first* column of $\mathbf{B}$:

$c_{11} = (2 \times 1) + (1 \times -2) + (3 \times 0) + (4 \times 4) = 16$



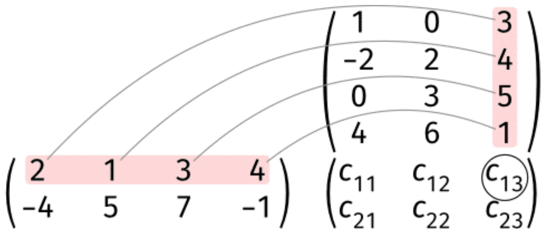
The entry $c_{12}$ in the *first* row and *second* column of $\mathbf{C} = \mathbf{AB}$ is given by summing the products of the entries in the *first* row of $\mathbf{A}$ with the entries of the *second* column of $\mathbf{B}$:

$c_{12} = (2 \times 0) + (1 \times 2) + (3 \times 3) + (4 \times 6) = 35$



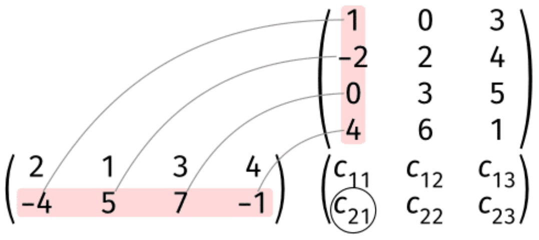
The entry $c_{13}$ in the *first* row and *third* column of $\mathbf{C} = \mathbf{AB}$ is given by summing the products of the entries in the *first* row of $\mathbf{A}$ with the entries of the *third* column of $\mathbf{B}$:

$c_{13} = (2 \times 3) + (1 \times 4) + (3 \times 5) + (4 \times 1) = 29$



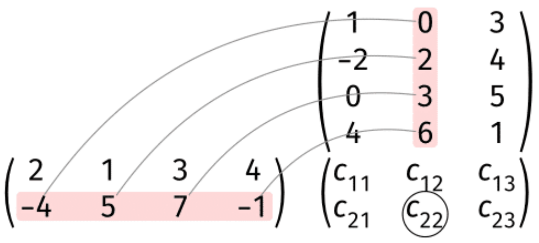
The entry $c_{21}$ in the *second* row and *first* column of $\mathbf{C} = \mathbf{AB}$ is given by summing the products of the entries in the *second* row of $\mathbf{A}$ with the entries of the *first* column of $\mathbf{B}$:

$c_{21} = (-4 \times 1) + (5 \times -2) + (7 \times 0) + (-1 \times 4) = -18$



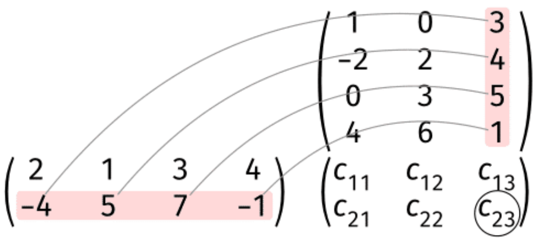
The entry $c_{22}$ in the *second* row and *second* column of $\mathbf{C} = \mathbf{AB}$ is given by summing the products of the entries in the *second* row of $\mathbf{A}$ with the entries of the *second* column of $\mathbf{B}$:

$c_{22} = (-4 \times 0) + (5 \times 2) + (7 \times 3) + (-1 \times 6) = 25$



The entry $c_{23}$ in the *second* row and *third* column of $\mathbf{C} = \mathbf{AB}$ is given by summing the products of the entries in the *second* row of $\mathbf{A}$ with the entries of the *third* column of $\mathbf{B}$:

$c_{23} = (-4 \times 3) + (5 \times 4) + (7 \times 5) + (-1 \times 1) = 42$



So overall, $\mathbf{C} = \begin{bmatrix} 16 & 35 & 29 \\ -18 & 25 & 42 \end{bmatrix}$

In [24]:
```
A = np.array([[2,1,3,4],[-4,5,7,-1]])
B = np.array([[1,0,3],[-2,2,4],[0,3,5],[4,6,1]])

print('AB')
print(A @ B)
```

```
AB
[[ 16  35  29]
 [-18  25  42]]
```

Alternatively, we can use:

In [25]:
```
print(np.dot(A,B))
```

```
[[ 16  35  29]
 [-18  25  42]]
```

There is a `np.multiply` method, which performs element-wise multiplication, i.e. $c_{ij} = a_{ij} b_{ij}$.

In this case $\mathbf{A}$ and $\mathbf{B}$ need to have the same order.

In [26]:
```
print(np.multiply(A,A))
```

```
[[ 4  1  9 16]
 [16 25 49  1]]
```

For a $m \times n$ matrix $\mathbf{A}$, we have the result: $\mathbf{I}_m\mathbf{A} = \mathbf{A}\mathbf{I}_n = \mathbf{A}$.

**Integer powers of matrices**

A square matrix $\mathbf{A}$ may be raised to integer powers:

- $\mathbf{A}^2 = \mathbf{A} \times \mathbf{A}$
- $\mathbf{A}^3 = \mathbf{A} \times \mathbf{A} \times \mathbf{A}$
- and so on $\dots$

In Python, we can take powers of matrices using `np.linalg.matrix_power` as follows:

In [27]:
```
# Cube matrix F
print('\nF^3')
print(np.linalg.matrix_power(F,3))
```

```

F^3
[[-247  288 -210]
 [-168 -355  -66]
 [-144   18  -85]]
```

# Inverse of a Matrix

The **inverse** of a square matrix $\mathbf{A}$ is the matrix $\mathbf{A}^{-1}$ such that $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix.

A square matrix may or may not have an inverse.

- If a matrix *has* an inverse, it is called **invertible** or **non-singular**.
- If a matrix *does not* have an inverse, it is called **non-invertible** or **singular**.

To determine whether a matrix $\mathbf{A}$ is singular or not, we can calculate a number called the **determinant** of $\mathbf{A}$, denoted det $\mathbf{A}$ or $|\mathbf{A}|$. $\mathbf{A}$ is invertible iff $|\mathbf{A}| \neq 0$. That is, the inverse of $\mathbf{A}$ exists iff $|\mathbf{A}| \neq 0$.

**Inverse of a $2 \times 2$ matrix**

Calculating the determinant is a simple process for $2 \times 2$ matrices. Consider the $2 \times 2$ matrix :

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

The **determinant** of $\mathbf{A}$, denoted det $\mathbf{A}$ or $|\mathbf{A}|$, is the number $ad - bc$.

The inverse of $\mathbf{A}$ exists iff $|\mathbf{A}| \neq 0$.

If $|\mathbf{A}| \neq 0$, then $\mathbf{A}^{-1} = \dfrac{1}{|\mathbf{A}|}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$.

***Example 1***

Determine the inverse of $\mathbf{A} = \begin{bmatrix} 5 & 3 \\ 4 & 8 \end{bmatrix}$

$|\mathbf{A}| = (5 \times 8) - (3 \times 4) = 40 - 12 = 28.$

So $\mathbf{A}^{-1} = \dfrac{1}{28}\begin{bmatrix} 8 & -3 \\ -4 & 5 \end{bmatrix}$

***Example 2***

Determine the inverse of $\mathbf{B} = \begin{bmatrix} 3 & 6 \\ 2 & 4 \end{bmatrix}$

$|\mathbf{B}| = (3 \times 4) - (2 \times 6) = 12 - 12 = 0$. So $\mathbf{B}^{-1}$ does not exist.

**Calculating determinants and inverses in Python**

We will use the function `det` from `linalg` to compute the determinant and another function `inv` to compute the inverse.

Note, however, that `inv` may well generate an answer even for a singular matrix so we must be careful. We should check the determinant using `det` and if it is very close to zero the matrix may be singular. (Because computer representations of numbers have a finite precision, there can be numerical errors so the determinant of a singular matrix might not come out as exactly zero.)

When we compute an inverse we can always check the solution. For example, if we calculate the inverse of the $3 \times 3$ matrix $\mathbf{F}$ we can check that when multiplied by the original matrix $\mathbf{F}$ it gives the $3 \times 3$ identity matrix.

In [28]:
```
# Determinant of matrix F
print('Determinant of F')
print(np.linalg.det(F))

# Inverse of F
F_inv = np.linalg.inv(F)
print('\nInverse of F')
print(F_inv)

# Check F * F_inv
print('\nCheck F x F_inv = I')
print(F @ F_inv)
```

```
Determinant of F
131.00000000000006

Inverse of F
[[ 0.14503817 -0.06870229 -0.17557252]
 [ 0.0610687   0.07633588  0.08396947]
 [-0.09160305 -0.11450382  0.3740458 ]]

Check F x F_inv = I
[[ 1.00000000e+00 -1.38777878e-17 -5.55111512e-17]
 [ 0.00000000e+00  1.00000000e+00  1.66533454e-16]
 [ 2.77555756e-17  2.77555756e-17  1.00000000e+00]]
```

## Solving Simultaneous Equations

Consider the simultaneous equations:

$$ax_1 + bx_2 = y_1$$
$$cx_1 + dx_2 = y_2$$

where $a, b, c, d, y_1, y_2$ are constants (the values of which we know) and $x_1, x_2$ are variables whose value we need to determine.

We can rewrite these equations using matrices as

$$\mathbf{A}\mathbf{x} = \mathbf{y}$$

where $\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$, $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ and $\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$

To solve for $\mathbf{x}$ we rearrange $\mathbf{A}\mathbf{x} = \mathbf{y}$ to be in the form $\mathbf{x} = \ldots$.

We do this by (left) multiplying both sides of $\mathbf{A}\mathbf{x} = \mathbf{y}$ by $\mathbf{A}^{-1}$:

$$\begin{aligned} \mathbf{A}^{-1}\mathbf{A}\mathbf{x} &= \mathbf{A}^{-1}\mathbf{y} \\ \mathbf{I}\mathbf{x} &= \mathbf{A}^{-1}\mathbf{y} \qquad \text{because } \mathbf{A}^{-1}\mathbf{A} = \mathbf{I} \\ \mathbf{x} &= \mathbf{A}^{-1}\mathbf{y} \qquad \text{because } \mathbf{I}\mathbf{x} = \mathbf{x} \end{aligned}$$

In the case of two simulteanous equations, we can do these matrix calculations by hand, because we know how to calculate the inverse of a $2 \times 2$ matrix.

***Example***

Using matrices, solve the simultaneous equations:

$$2x_1 + x_2 \ = 3$$
$$x_1 + 3x_2 = -1$$

Let $\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}$, $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ and $\mathbf{y} = \begin{bmatrix} 3 \\ -1 \end{bmatrix}$

Then $\mathbf{Ax} = \mathbf{y}$, so $\mathbf{x} = \mathbf{A}^{-1}\mathbf{y}$.

Now check that $\mathbf{A}^{-1}$ exists: $|\mathbf{A}| = (2 \times 3) - (1 \times 1) = 6 - 1 = 5 \neq 0$

So $\mathbf{A}^{-1}$ exists and is given by:

$$\mathbf{A}^{-1} = \frac{1}{|A|}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix} = \frac{1}{5}\begin{bmatrix} 3 & -1 \\ -1 & 2 \end{bmatrix}$$

Using $\mathbf{x} = \mathbf{A}^{-1}\mathbf{y}$ we have:

$$\mathbf{x} = \frac{1}{5}\begin{bmatrix} 3 & -1 \\ -1 & 2 \end{bmatrix}\begin{bmatrix} 3 \\ -1 \end{bmatrix} = \frac{1}{5}\begin{bmatrix} (3 \times 3) + (-1 \times -1) \\ (-1 \times 3) + (2 \times -1) \end{bmatrix} = \frac{1}{5}\begin{bmatrix} 10 \\ -5 \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$$

i.e. $x_1 = 2, x_2 = -1$

**Solving simultaneous equations using Python**

With a computer we can solve much larger problems than two simultaneous equations with two unknowns. Here we illustrate a case with just three equations and three unknowns (which could be done by hand without too much extra work).

Determine the solution to the following equations, i.e. determine the values of $x_1, x_2, x_3$.

$$\begin{aligned} 2x_1 + 3x_2 + x_3 &= 1 \\ x_1 + 5x_2 + 3x_3 &= 4 \\ 7x_1 + x_2 + 2x_3 &= 2 \end{aligned}$$

As before, we can represent this using matrices as $\mathbf{Ax} = \mathbf{y}$, where $\mathbf{A} = \begin{bmatrix} 2 & 3 & 1 \\ 1 & 5 & 3 \\ 7 & 1 & 2 \end{bmatrix}$, $\mathbf{y} = \begin{bmatrix} 1 \\ 4 \\ 2 \end{bmatrix}$, and $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$.

To calculate $\mathbf{x} = \mathbf{A}^{-1}\mathbf{y}$, we can use the function `solve` from `linalg`, as shown in the cell below. Note that as with the computation of inverses:

- we should check that $|\mathbf{A}| \neq 0$: if $|\mathbf{A}| = 0$ (or is extremely close to zero) then $\mathbf{A}$ is singular and there is no solution to the simultaneous equations
- once we have a solution $\mathbf{x}$ we can always use matrix multiplication to check that $\mathbf{A}$ times $\mathbf{x}$ does indeed give $\mathbf{y}$.

In [29]:

```python
A = np.array([[2,3,1],[1,5,3],[7,1,2]])
y = np.array([[1,4,2]]).T
print('A\n',A)
print('y\n',y)

x = np.linalg.solve(A,y)
print('x')
print(x)

print('\nSo the solution is:')
print('x_1 = ',x[0,0])
print('x_2 = ',x[1,0])
print('x_3 = ',x[2,0])
```

```
A
 [[2 3 1]
 [1 5 3]
 [7 1 2]]
y
 [[1]
 [4]
 [2]]
x
[[-0.13513514]
 [-0.08108108]
 [ 1.51351351]]

So the solution is:
x_1 =  -0.13513513513513514
x_2 =  -0.08108108108108114
x_3 =  1.5135135135135136
```