Matrices

A matrix is a rectangular array, in our case of real numbers.

- In general, we use bold capital letters, e.g. $\mathbf{A}$, for matrices, e.g.
  \[
  \mathbf{A} = \begin{bmatrix}
  2 & 4 & 0 \\
  1 & 3 & 2 \\
  \end{bmatrix}
  \]

- **Dimension**: A matrix with $m$ rows and $n$ columns is an $m \times n$ matrix
  - What are $m$ and $n$ for $\mathbf{A}$?

- We refer to an element of a matrix either using subscripts or indexes
  - $A_{i,j}$ or $\mathbf{A}[i,j]$ is the element in the $i$th row and $j$th column
  - We will index from 1
    - However, we will sometimes use position 0 for 'technical' purposes
    - And we must be aware that Python numpy arrays and matrices are 0-indexed
  - So what are $\mathbf{A}_{2,1}$, $\mathbf{A}_{1,2}$, $\mathbf{A}_{0,0}$ and $\mathbf{A}_{3,2}$?
A vector is a matrix that has only one column, i.e. a $m \times 1$ matrix

- A vector with $m$ rows is called a $m$-dimensional vector
- In general, we use bold lowercase letters for vectors, e.g.

$$x = \begin{bmatrix} 2 \\ 4 \\ 3 \end{bmatrix}$$

- Sometimes this is called a column vector
- Then, by contrast, a row vector is a matrix that has only one row, i.e. a $1 \times n$ matrix, e.g.

$$[2, 4, 3]$$

- Unless stated otherwise, a vector should be assumed to be a column vector.
- We can refer to an element using a single subscript, again most of the time indexed from 1
  - So what is $x_1$?

## Vectors and Matrices in Python

Of the many ways of representing vectors and matrices in Python, we will use two:

- From the pandas library, for vectors we will use Series, which are a kind of one-dimensional array, and for matrices we will use DataFrames, which are tabular data structures of rows and columns. As well as supporting fairly traditional indexing, the columns in a DataFrame can have names, which can also be used to extract data from the DataFrame.
- From the numpy library, we will use numpy arrays, which can be one-dimensional, two-dimensional, or have more dimensions. The machine learning library that we will use, scikit-learn, expects its data to arrive as numpy arrays.

For the rest of this lecture, we'll exemplify numpy arrays. (Quite a few of the clever indexing methods that we look at below also apply to pandas data structures.)

```python
In [3]: # Vectors
# We will use a numpy 1d array, which we can create from a list
# But, done this way, there is no way for us to distinguish between column- and row-vectors
x = np.array([2, 4, 3])

# Matrices
# We will use a numpy 2d array, which we can create from a list of lists
A = np.array([[2, 4, 0], [1, 3, 2]])
```

## Indexing

Be warned that numpy arrays are indexed from zero, in contrast to what we used above.
In [4]: # x = np.array([2, 4, 3])
x[1]
Out[4]: 4

In [5]: # A = np.array([[2, 4, 0], [1, 3, 2]])
   A[1, 2] # Note you can write A[1, 2] as well as A[1][2]
Out[5]: 2

Shape and Type

Every numpy array has a shape attribute (dimension):

In [6]: x.shape
Out[6]: (3,)

In [7]: A.shape
Out[7]: (2, 3)

All the elements of a numpy array must be the same type. The np.array function infers the type from the data:

In [8]: A.dtype
Out[8]: dtype('int64')

We'll sometimes want to change the type, which we can do using the astype method, which creates a copy of the array but with the new type:

In [9]: A.astype(float)
Out[9]: array([[ 2.,  4.,  0.],
                [ 1.,  3.,  2.]])

Other Ways to Create Arrays

The zeros, ones and empty functions create arrays of the given shape containing just zeros, ones and arbitrary values, respectively:
More Indexing, and also Slicing

Python's indexing and slicing operators work on numpy arrays and matrices, e.g.

```python
In [11]: # First element of x where x = np.array([2, 4, 3])
    x[0]
Out[11]: 2

In [12]: # All but the first element of x
    x[1:]
Out[12]: array([4, 3])

In [13]: # All but the last element of x
    x[:1]
Out[13]: array([2, 4])

In [14]: # First row of A where A = np.array([[2, 4, 0], [1, 3, 2]])
    A[0]
Out[14]: array([2, 4, 0])

In [15]: # All but the first row of A
    A[1:]
Out[15]: array([[1, 3, 2]])

In [16]: # All but the last row of A
    A[:1]
Out[16]: array([[2, 4, 0]])

In [17]: # First column of A
    A[:, 0]
Out[17]: array([2, 1])

In [18]: # All but the first column of A
    A[:, 1:]
Out[18]: array([[4, 0],
                [3, 2]])
```
In [19]: # All but the last column of A
   A[:, :-1]
Out[19]: array([[2, 4],
    [1, 3]])

Unlike Python lists, slices don't copy the data; they refer to the original array. Hence, updates affect the original array:

In [20]: x[1:] = 12
   
x
Out[20]: array([ 2, 12, 12])

Fancy Indexing and Boolean Indexing

In addition to indexing and slicing, for numpy arrays there is so-called fancy indexing (where you use a list or array of ints) and Boolean indexing (where a numpy array of Booleans selects which elements to return):

In [21]: # Change x back to what it was
   x = np.array([2, 4, 3])
   
   # Fancy indexing to get positions 2 and 0 of x - in that order
   x[[2, 0]]
Out[21]: array([3, 2])

In [22]: # Boolean indexing
   bools = np.array([False, True, True])
   x[bools]
Out[22]: array([4, 3])

In [23]: # You can create the Boolean array with an array comparison:
   bools = [x > 2]
   x[bools]
Out[23]: array([4, 3])

In [24]: # More concisely:
   x[x > 2]
Out[24]: array([4, 3])

Matrix Addition and Subtraction

Two matrices $A$ and $B$ can be added or subtracted if they have the same dimensions. Their sum is obtained by adding or subtracting corresponding elements, e.g.

$$ A = \begin{bmatrix} 2 & 4 & 0 \\ 1 & 3 & 2 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 0 & 5 \\ 2 & 3 & 2 \end{bmatrix} \quad A + B = \begin{bmatrix} 3 & 4 & 5 \\ 3 & 6 & 4 \end{bmatrix} $$
Scalar Multiplication

Matrices can be multiplied by real numbers (in this context, often called 'scalars'). The scalar product is obtained by multiplying each element by the scalar, e.g.

\[
A = \begin{bmatrix} 2 & 4 & 0 \\ 1 & 3 & 2 \end{bmatrix}, \quad 2A = \begin{bmatrix} 4 & 8 & 0 \\ 2 & 6 & 4 \end{bmatrix}, \quad A/2 = \begin{bmatrix} 1 & 2 & 0 \\ 0.5 & 1.5 & 1 \end{bmatrix}
\]

Matrix Addition, Subtraction and Scalar Multiplication in Python

Numpy arrays enable operations like these using the normal addition, subtraction and multiplication operators and without writing for loops:

In [25]:
   :A = np.array([[2, 4, 0], [1, 3, 2]])
   :B = np.array([[1, 0, 5], [2, 3, 2]])
   :
   :A + B

Out[25]:
   :array([[3, 4, 5],
          [3, 6, 4]]
   :

Similarly, operations with scalars are also applied elementwise:

In [26]:
   :2 * A

Out[26]:
   :array([[4, 8, 0],
          [2, 6, 4]])

Matrix Multiplication
We can compute \( AB \), the result of multiplying matrices \( A \) and \( B \), provided the number of columns of \( A \) equals the number of rows of \( B \)

- If \( A \) is a \( m \times p \) matrix and \( B \) is a \( p \times n \) matrix, then we can compute \( C = AB \)
- \( C \) will be a \( m \times n \) matrix
- \( C_{i,j} \) is obtained by multiplying elements of the \( i \)th row of \( A \) by corresponding elements of the \( j \)th column of \( B \) and summing:
  \[
  C_{i,j} = \sum_{k=1}^{p} A_{i,k}B_{k,j}
  \]
- E.g.
  \[
  A = \begin{bmatrix}
  2 & 4 & 0 \\
  1 & 3 & 2 
  \end{bmatrix}
  B = \begin{bmatrix}
  3 & 1 & 2 \\
  2 & 3 & 1 \\
  1 & 3 & 3 
  \end{bmatrix}
  AB = \begin{bmatrix}
  14 & 14 & 8 \\
  11 & 16 & 11 
  \end{bmatrix}
  \]
- Since vectors are just one-column vectors, matrix multiplication can apply — provided the dimensions are OK, e.g.

\[
A = \begin{bmatrix}
  2 & 4 & 0 \\
  1 & 3 & 2 
  \end{bmatrix}
  x = \begin{bmatrix}
  2 \\
  3 \\
  1 
  \end{bmatrix}
  y = \begin{bmatrix}
  2 \\
  3 \\
  1 
  \end{bmatrix}
  Ax = \begin{bmatrix}
  16 \\
  13 
  \end{bmatrix}
  Ay \text{ is undefined}
  \]

**Matrix Multiplication in Python**

`numpy` offers `dot` as a function or method for matrix multiplication:

```python
In [27]:
A = np.array([[2, 4, 0], [1, 3, 2]])
B = np.array([[3, 1, 2], [2, 3, 1], [1, 3, 3]])

# Multiplication as a function
np.dot(A, B)

# Multiplication as a method
A.dot(B)

Out[27]:
arrray([[14, 14, 8],
        [11, 16, 11]])
```

Be warned that \( A \ast B \) on `numpy` arrays is **not** the same as matrix multiplication as defined above. Instead, it is performed elementwise on corresponding elements of two equal-sized arrays.

**Transpose**
The transpose of $m \times n$ matrix $A$, written $A^T$, is the $n \times m$ matrix in which the first row of $A$ becomes the first column of $A^T$, the second row of $A$ becomes the second column of $B$, and so on.

- $A_{i,j}^T = A_{j,i}$
- E.g.

$$A = \begin{bmatrix} 2 & 4 & 0 \\ 1 & 3 & 2 \end{bmatrix} A^T = \begin{bmatrix} 2 & 1 \\ 4 & 3 \\ 0 & 2 \end{bmatrix}$$

- As a special case, if $x$ is a $m$-dimensional column vector ($m \times 1$), then $x^T$ is a $m$-dimensional row vector ($1 \times m$), e.g.

$$x = \begin{bmatrix} 2 \\ 4 \\ 3 \end{bmatrix} x^T = [2, 4, 3]$$

NumPy arrays offer easy ways to compute their transpose: either the transpose method or the T attribute:

```python
In [28]: A = np.array([[2, 4, 0], [1, 3, 2]])

    # Transpose as a method
    # A.transpose()

    # Transpose as an attribute
    A.T

Out[28]: array([[2, 1],
               [4, 3],
               [0, 2]])
```

**Inverses and Identity Matrices**
The $n \times n$ identity matrix, $I_n$, contains zeros except for entries on the main diagonal (from top left to bottom right).

- $I_n[i, i] = 1$ for $i = 1, \ldots, n$ and $I_n[i, j] = 0$ for $i \neq j$, e.g.

$$
I_3 = \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
$$

- If $A$ is an $m \times n$ matrix then, $A I_n = I_m A = A$

If $A$ is a $n \times n$ matrix, then its inverse, $A^{-1}$ (if it has one) is also a $n \times n$ matrix such that $AA^{-1} = I_n$.

- $A = \begin{bmatrix}
1 & 0 & 2 \\
2 & -1 & 3 \\
4 & 1 & 8
\end{bmatrix}$ $A^{-1} = \begin{bmatrix}
-11 & 2 & 2 \\
-4 & 0 & 1 \\
6 & -1 & -1
\end{bmatrix}$

- Some $n \times n$ matrices do not have inverses, e.g.

$$
\begin{bmatrix}
1 & 1 & 1 \\
1 & 1 & 1 \\
1 & 1 & 1
\end{bmatrix}
$$

In these cases, provided the matrix is square, you can compute a pseudo-inverse, which you can use for some of the same purposes instead.

`numpy.linalg` offers function `inv` for computing inverses, but also function `pinv` for computing the Moore-Penrose pseudo-inverse:

```
In [29]: import numpy.linalg as npla
   A = np.array([[1, 0, 2], [2, -1, 3], [4, 1, 8]])
   npla.inv(A)
```

```
Out[29]: array([[-11.,  2.,  2.],
               [-4.,  0.,  1.],
               [ 6., -1., -1.]]
```

```
In [30]: npla.pinv(A)
```

```
Out[30]: array([[-1.10000000e+01,  2.00000000e+00,  2.00000000e+00],
               [-4.00000000e+00,  1.42108547e-14,  1.00000000e+00],
               [ 6.00000000e+00, -1.00000000e+00, -1.00000000e+00]])
```
Vectorization

Algorithms that might otherwise need for-loops can often be written much more succinctly by expressing them in terms of matrix operations. More than this, if your programming language has efficient implementations of these operations, the resulting programs can run much faster too.

Using fast matrix operations in this way is known as vectorization.

numPy’s vectorized array operations, for example, are typically one or more orders of magnitude faster than their pure Python equivalents.

Here are a few more examples.

Evaluating a Linear Equation
Suppose you have a linear equation, i.e. of this form:

\[ y = \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n \]

- ... lots of little multiplications, all added together

You know that this can be written in this form:

\[ y = \sum_{i=1}^{n} \beta_i x_i \]

If you had to write code to implement this, you might be thinking of a for-loop.

But if we assume that \(x\) is an \(n\)-dimensional row vector and \(\beta\) is an \(n\)-dimensional column vector, then we can instead write the equation in this form:

\[ y = x\beta \]

and you can implement this with the numpy library's matrix multiplication method.

Let's make this more concrete. Suppose we have this linear equation:

\[ y = 3x_1 + 2x_2 + 4x_4 \]

and we want to evaluate it for \(x = [5, 1, 3]\)

We take the coefficients to give us \(\beta = \begin{bmatrix} 3 \\ 2 \\ 4 \end{bmatrix}\) and now we compute

\[ y = [5, 1, 3] \begin{bmatrix} 3 \\ 2 \\ 4 \end{bmatrix} = 29 \]

Let's see it in Python:

```python
In [33]: beta = np.array([3, 2, 4])
x = np.array([5, 1, 3])

# Evaluate the linear equation - without a for-loop
y = x.dot(beta)

# Display y
y
```

```
Out[33]: 29
```

**Evaluating a Linear Equation Multiple Times**
Suppose we want to evaluate the linear equation $y = \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$ on a number of different row vectors, first for $[5, 1, 3]$, then for $[2, 6, 2]$, then for $[3, 3, 3]$, and finally for $[3, 3, 5]$. Maybe you're thinking of a for-loop again?

Instead, we put these different row vectors into a single matrix $X$

$$X = \begin{bmatrix} 5 & 1 & 3 \\ 2 & 6 & 2 \\ 3 & 3 & 3 \\ 3 & 3 & 5 \end{bmatrix}$$

And we simply evaluate

$$y = X \beta$$

Here it is in Python:

```python
In [34]: beta = np.array([3, 2, 4])
X = np.array([[5, 1, 3], [2, 6, 2], [3, 3, 3], [3, 3, 5]])

# Evaluate the linear equation multiple times - without a for-loop!
y = X.dot(beta)

# Display y
y
```

```
Out[34]: array([29, 26, 27, 35])
```