Algebra & Calculus

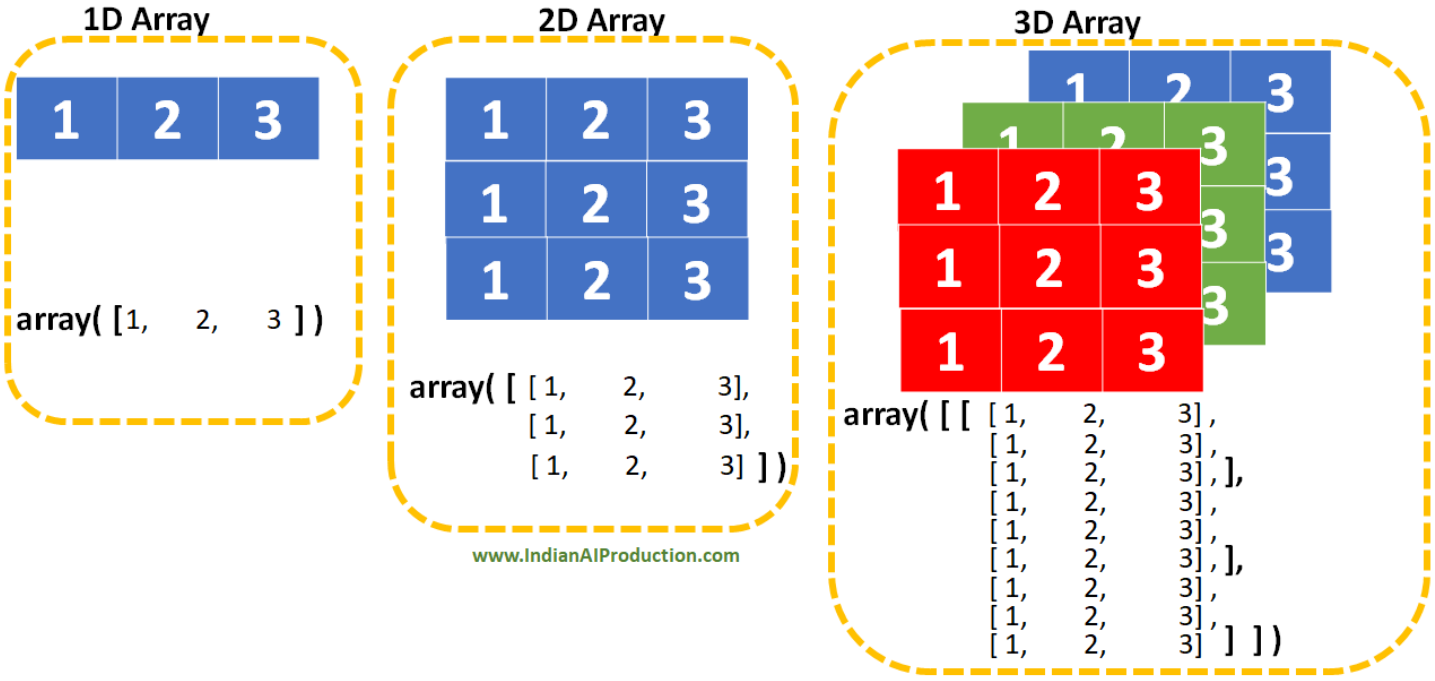
- **Recap (`NumPy`) - 5'**
- **Algebra - 1hr 10'**
  - Functions
  - Equations
  - Vectors
  - Matrices
  - Matrix multiplication for `data science`
- **Calculus - 15'**
  - Derivatives
  - Partial derivatives
  - Integrals

 lecture starter for teachers (https://github.com/lewagon/data-lecture-starters/blob/main/starters/03-Math_01-Algebra-Calculus_starter.ipynb)

Recap (`NumPy`)

`NumPy` is all about handling n-dimensional arrays

👉 objects of class `numpy.ndarray`



`NumPy` know-hows

📝 Let's open a notebook

👉 **creating** an array: `np.array`

👉 important *attributes* of an array: `shape` , `size` , `ndim` and `dtype`

👉 **extracting** values from an array: *indexing*, *slicing* and *boolean indexing*

In [ ]:
```
import numpy as np
A = np.array([[1, 2],
              [3, 4],
              [5, 6],])
```

In [ ]:
```
type(A)
```

Out[ ]: `numpy.ndarray`

In [ ]:
```
A.shape  # very important for today's lecture!
```

Out[ ]:
```
(3, 2)
```

In [ ]:
```
print(f"{A.size = }\n{A.ndim = }\n{A.dtype = }")
```

```
A.size = 6
A.ndim = 2
A.dtype = dtype('int64')
```

In [ ]:
```
A[1, 1] # indexing
```

Out[ ]:
```
4
```

In [ ]:
```
A[:2, :] # slicing
```

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

In [ ]:
```
A > 2 # this is a boolean mask
```

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

In [ ]:
```
A[A > 2] # boolean indexing = use a boolean mask as an index
```

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

`NumPy` is good for your health 💪

- faster computing (https://stackoverflow.com/questions/8385602/why-are-numpy-arrays-so-fast)
- very compact way to code
- same logic applies to `Pandas`
- 👉 before using a `for` loop, ask yourself 🤔 ***can i do it the `NumPy` way?***

1) Algebra

Algebra (https://en.wikipedia.org/wiki/Algebra) is the branch of mathematics that:

- **represents** mathematical `objects` as `symbols`
- **manipulates** them in formulas

☝️ `Data scientists` ❤️ **Linear Algebra** (https://en.wikipedia.org/wiki/Linear_algebra)

Mathematical vocabulary

Let's have a look at this `formula` :

$$x + 1 = c + {y_0}^2 + \frac{1}{4} y_1$$

- $(x + 1)$ and $(c + {y_0}^2 + \frac{1}{4} y_1)$ are `expressions`
- x is a `symbol` standing for a `variable` and c is a `constant`
- in $(x + 1)$ 👉 $+$ is an `operator` , x and 1 are `operands`
- in y_0^2 👉 0 is an `index` (subscript) and 2 is an `exponent` (superscript)

🔥 `LaTeX` syntax in Markdown cell of a `Jupyter` notebook 🔥

```
$$ x + 1 = c + y_0^2 + \frac{1}{4} y_1  $$
```

1.1) Functions

Function definition

👉 A `function` is a *mapping* of each element from one set to a unique element of a second set.

$$f: \mathbb{R} \to \mathbb{R}, f(x) = x^2 + 1$$

📝 Let's implement this mathematical function in `Python`

In [ ]:
```
# classical way...
def square_plus_one(x):
    return x**2 + 1
type(square_plus_one)
```

Out[ ]:
```
function
```

In [ ]:
```
# ... or using the "lambda" keyword
square_plus_one = lambda x: x**2 + 1
type(square_plus_one)
```

Out[ ]:
```
function
```

📝 Let's use the `square_plus_one` function

In [ ]:
```
for x in range(5):
    y = square_plus_one(x)
    print(f"{x=} -> {y=}")
```

```
x=0 -> y=1
x=1 -> y=2
x=2 -> y=5
x=3 -> y=10
x=4 -> y=17
```

Wait. 🤔 did we just use a `for` loop?

🤬 ***Let's do it the `NumPy` way*** 💪

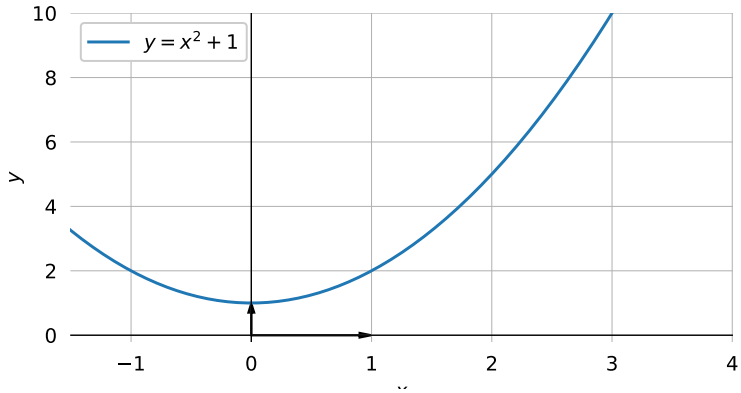
In [ ]:
```
x = np.arange(5)          # x contains 5 values
y = square_plus_one(x)    # y contains 5 values
print(f"{x = }")
print(f"{y = }")
```

```
x = array([0, 1, 2, 3, 4])
y = array([ 1,  2,  5, 10, 17])
```

Function representation

📝 Draw the $y = f(x)$ curve in an ***orthonormal coordinate system***.

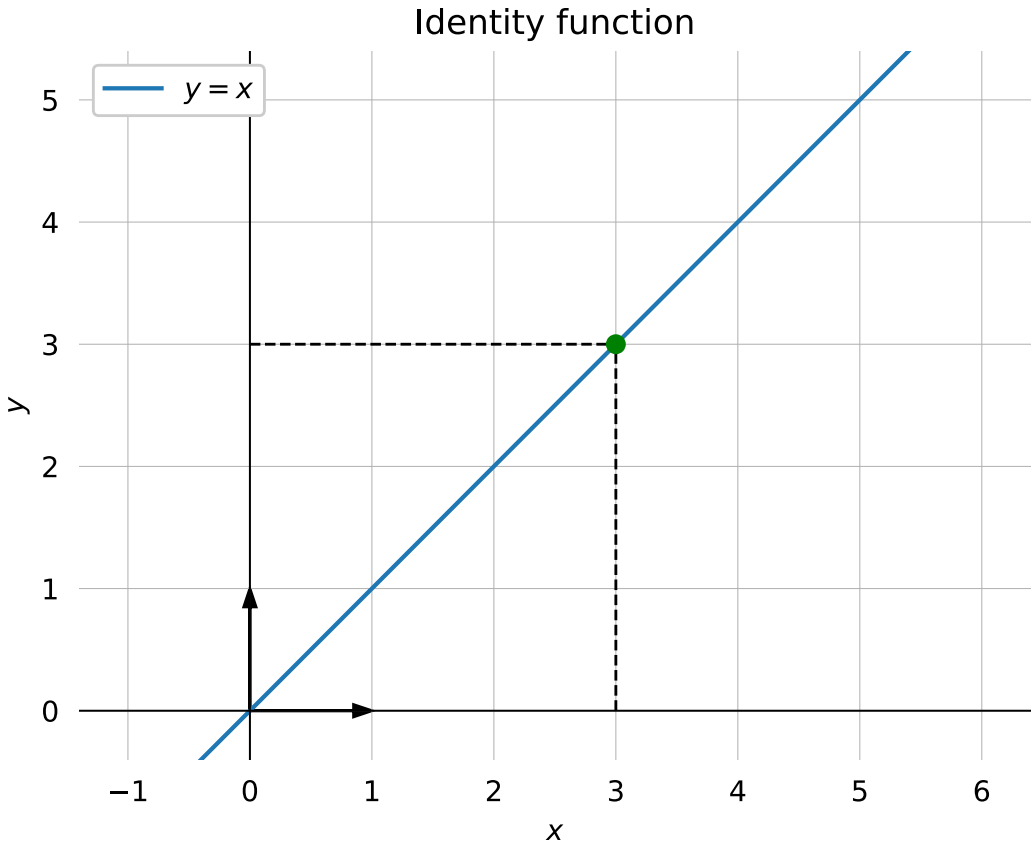
In [ ]:
```
x = np.linspace(-1, 2, 100)     # x contains 100 values
y = square_plus_one(x)          # y contains 100 values
plt.plot(x, y);                 # plot a line with 100 points
```



👉 No **vertical line** can intersect the curve twice 🧐 This is **always** true: one `input` x is mapped to only one `output` y

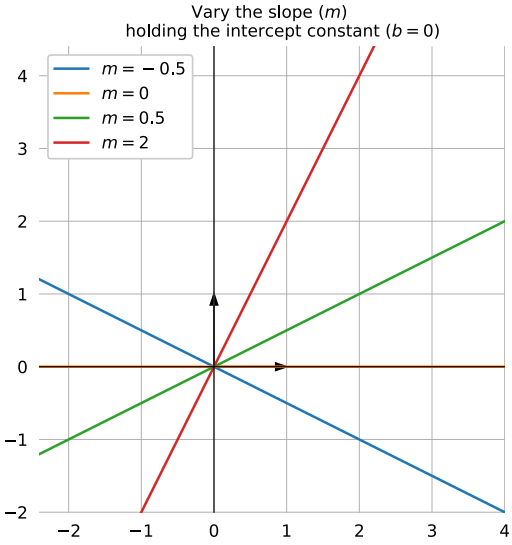
💡 Notice that a **horizontal line** may cross the curve several times 🧐 Several `input` x's may have the same `output` y

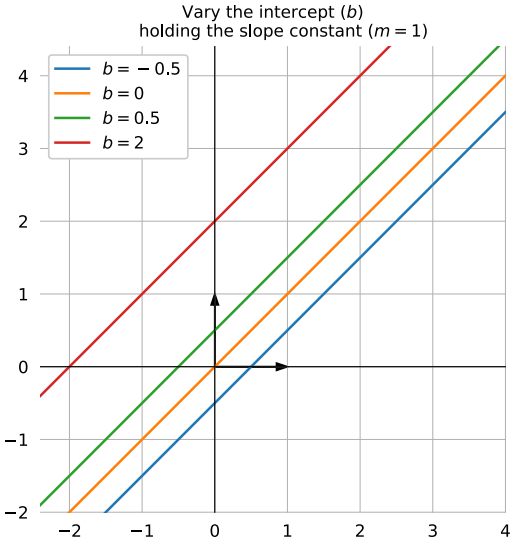
The identity function



Linear functions

A straight line is defined by the equation $f(x) = mx + b$
👉 m is the **slope** and 👉 b is the **intercept**



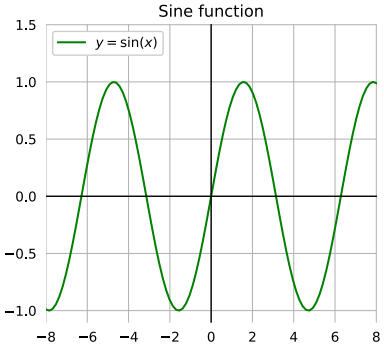


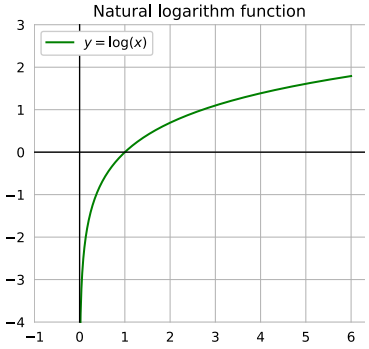
❓ can you calculate the **intercept** and the **slope** of any straight line?

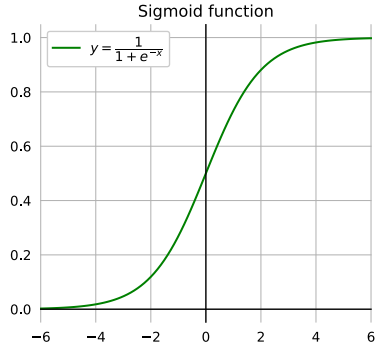
👉 $b = f(0)$ and 👉 $m = \frac{f(x_B) - f(x_A)}{x_B - x_A}$

Non-linear functions

👉 `square_plus_one` is not linear. Here are some more examples 🔽

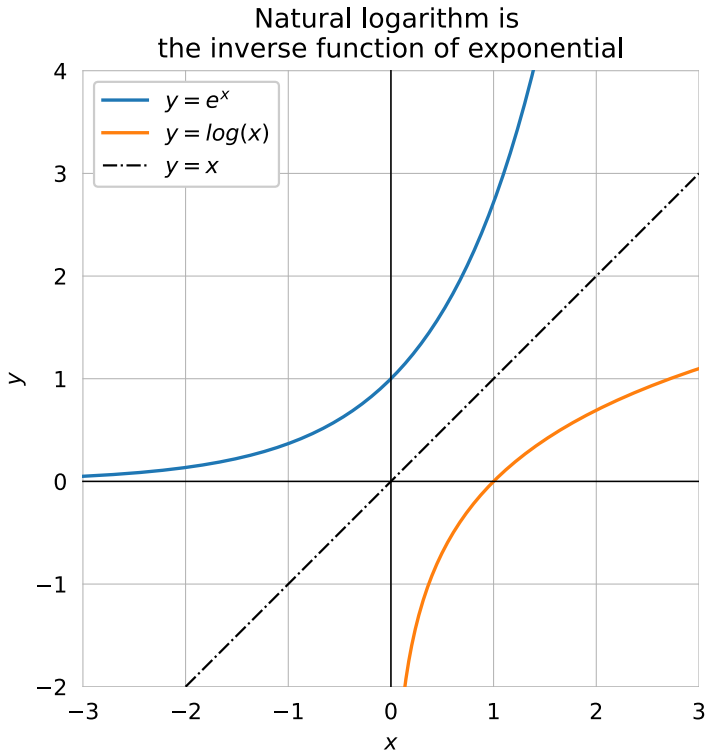






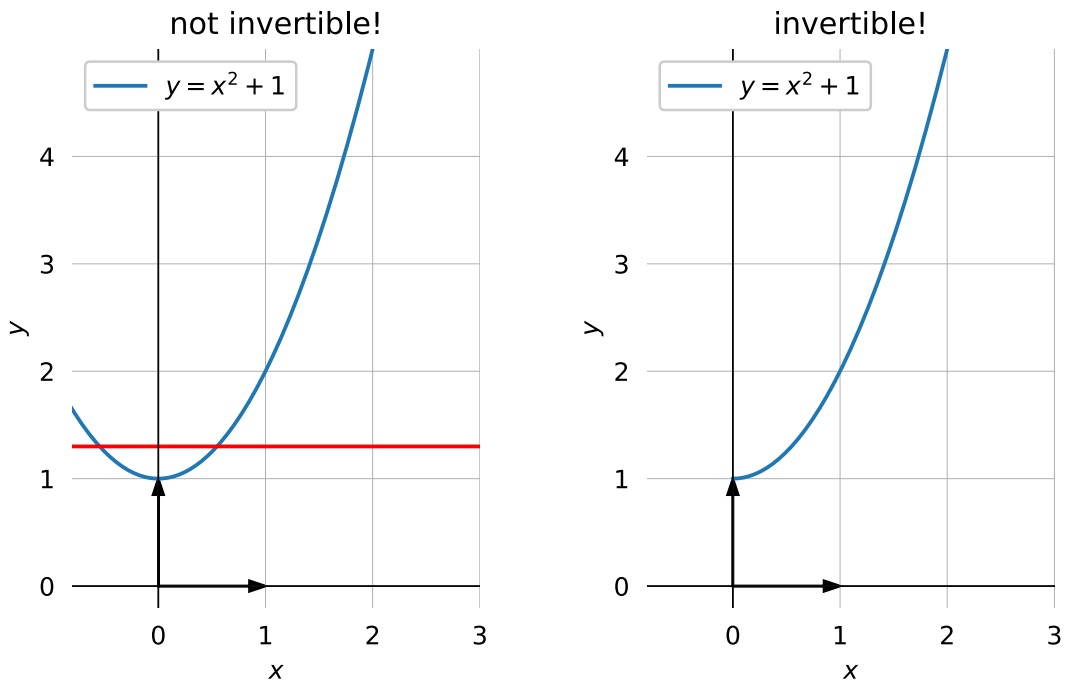
Inverse function (f^{-1}) of a function f

👉 An inverse function f^{-1} is a function that "reverses" f
$f^{-1}(f(x)) = x$ and $f(f^{-1}(x)) = x$



💡 The representation of f^{-1} is the mirror image of f through $y = x$

⚠️ Not all functions have an inverse. f has an inverse iff (if and only if) f is a bijection (https://en.wikipedia.org/wiki/Bijection).
💡 Graphically, it means **no** *horizontal line* can intersect the curve more than once.

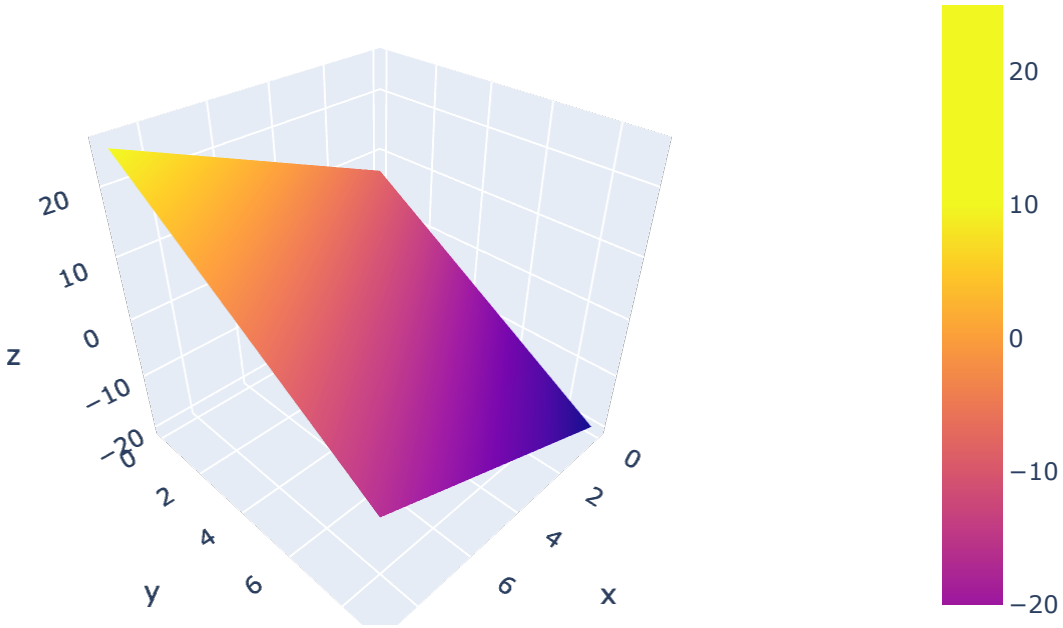


Multi-variate functions

👉 A multi-variate function takes 2 input variables or more.

$$f: \mathbb{R}^2 \to \mathbb{R}, f(x_1, x_2) = 2x_1 - 3x_2 + 7$$

In [ ]:
```
import plotly.graph_objects as go
fun = lambda x, y: 2*x - 3*y + 7           # bivariate function
x, y = np.meshgrid(range(10), range(10))   # grid of x's and y's
z = fun(x, y)                              # z's
fig = go.Figure(go.Surface(x=x, y=y, z=z)); fig.show()
```



1.2) Equations

👉 An `equation` is a formula that contains one or more `unknown` variables

$$2x = 4$$

👉 A `system of equations` is a collection of two or more equations involving the same set of `unknowns`.

$$\begin{cases} 3x + 2y - z = 1 \\ 2x - 2y + 4z = -2 \\ -x + \frac{1}{2}y - z = 0 \end{cases}$$

👉 A `solution` is a set of values of the unknown(s) that satisfy all these constraints.

⚠️ Depending on the equation, there may be:

- $2x = 4$ 👉 **one** solution
- $x^2 = 4$ 👉 **two** solutions
- ...
- $x = x$ 👉 an ***infinity*** of solutions
- $x = x + 2$ 👉 **zero** solutions

🤔 Problem:

- A `father` is 22 years older than his `daughter`.
- In 10 years, the `father` will be twice as old as his `daughter`.

How old are they?

👉 This translates into a system of two `equations` with two `unknowns`. Let y be the father's age and x his daughter's age.

$$\begin{cases} y = x + 22 & \textbf{(1)} \\ y + 10 = 2\,(x + 10) & \textbf{(2)} \end{cases}$$

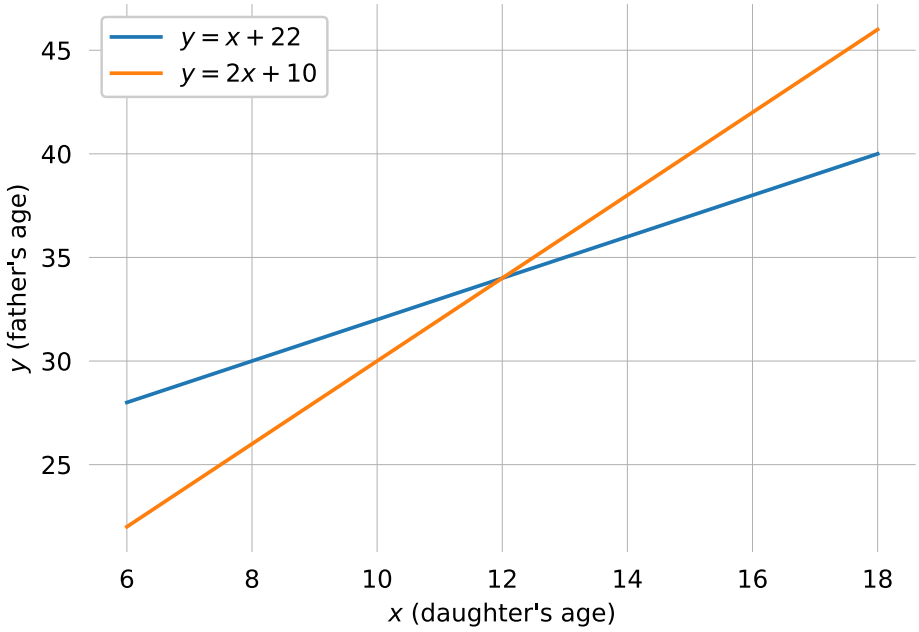
...which simplifies to:

$$\begin{cases} y = x + 22 & \textbf{(1)} \\ y = 2x + 10 & \textbf{(2)} \end{cases}$$

👉 We can solve this `graphically` 📈

💡 The blue line is the set of (x, y) that satisfy equation $\mathbf{(1)}$

💡 The orange line is the set of (x, y) that satisfy equation $\mathbf{(2)}$



👀 It seems there is one and only one point (x, y) that satisfies both equations $\mathbf{(1)}$ and $\mathbf{(2)}$.

The *high school* method 👩‍🏫 can get us to the exact answer:

👉 combine $\mathbf{(1)}$ and $\mathbf{(2)}$: $x + 22 = 2x + 10$, from which $x = 12$

👉 finally $\mathbf{(1)}$: $y = 12 + 22 = 34$ 😘

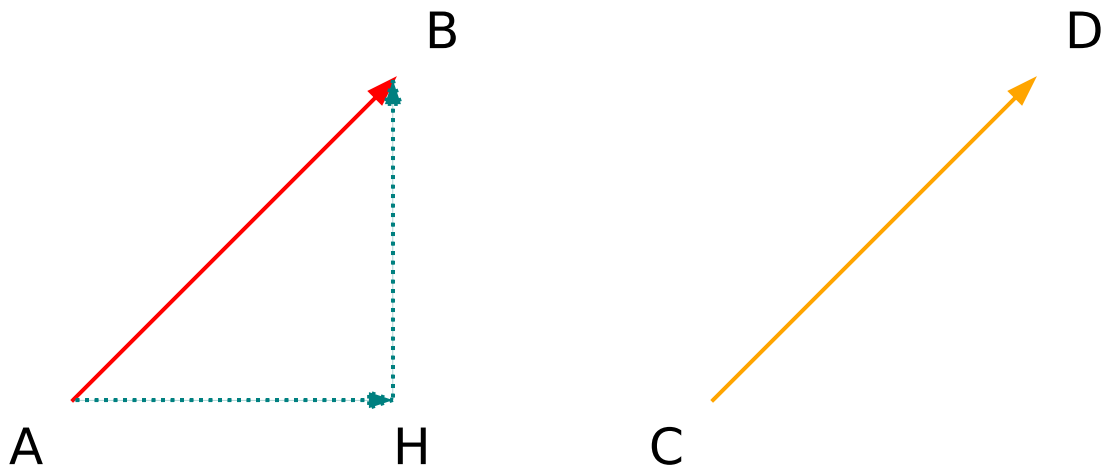
💡 today we will also solve this problem using `linear algebra` 🎉

1.3) Vectors

1.3.1) What's a vector?

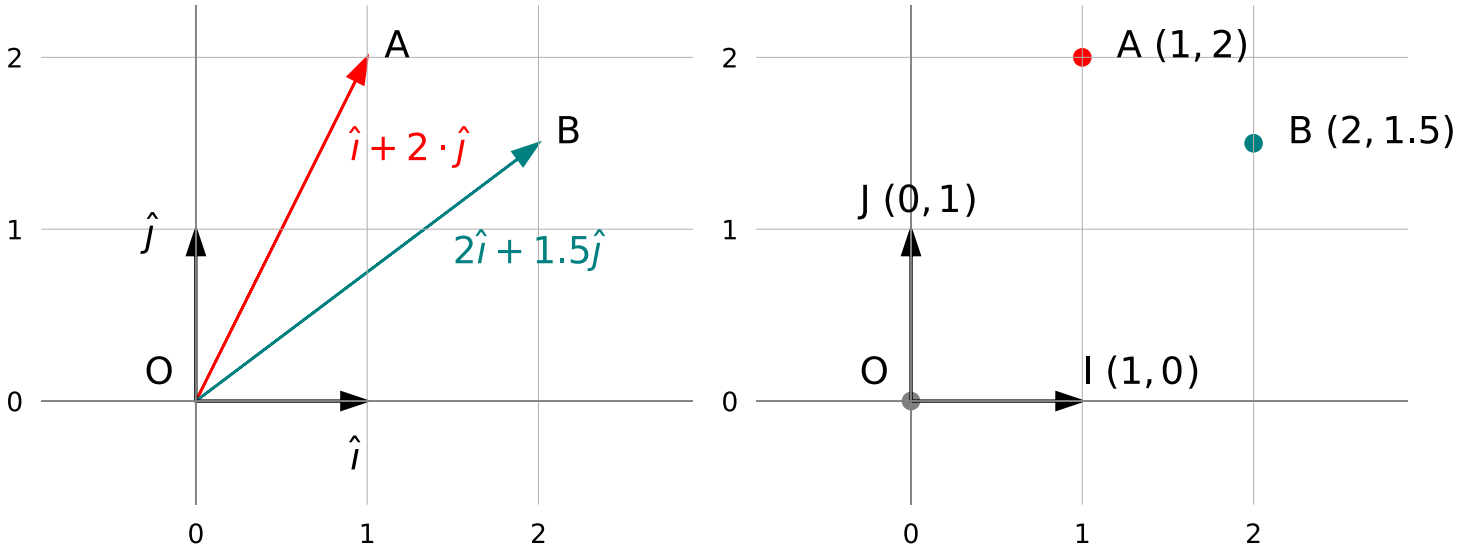
👉 In a **plane**, a geometrical `vector` is defined by 2 points or by a `magnitude`, a `direction` and a `sense`.

💡 Note that $\vec{AB} = \vec{AH} + \vec{HB}$ and $\vec{AB} = \vec{CD}$



Coordinates

👉 If we have an origin O and two unit vectors $\vec{i}$ and $\vec{j}$, any **vector** is uniquely defined by two coordinates.



💡 Conversely, any **couple** of numbers can be understood as a **point** in a plane, and as a **vector**.

Notation

👉 A `vector` :

- with an arrow on top 👉 $\vec{a}$
- or as a **bold**, *italic, lowercase* symbol 👉 $\boldsymbol{a}$

$$\boldsymbol{a} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} 1 & 2 \end{bmatrix}$$

The `unit vectors` are $\boldsymbol{i} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\boldsymbol{j} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

👉 By contrast, a `scalar` (= a simple quantity, a number): **not bold**, *italic, lowercase* symbol 👉 a

Vectors in higher dimensions

👉 Vector $\boldsymbol{a} = \begin{bmatrix} 1 & 2 \end{bmatrix}$ *lives* in a **2D space** (= a plane)

👉 Vector $\boldsymbol{u} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$ *lives* in a **3D space**

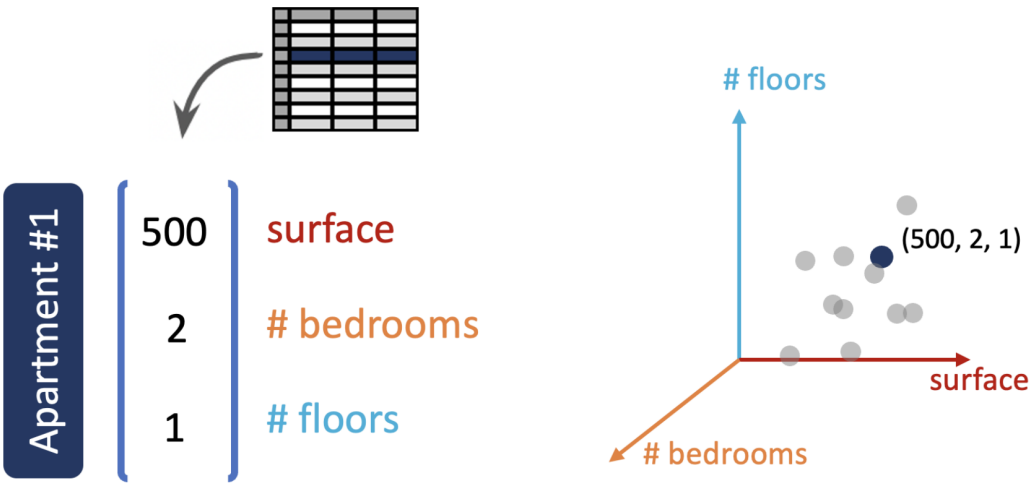
👉 Vector $\boldsymbol{x} = \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix}$ *lives* in a... **n-dimensional space**! 🤩

💡 At the end of the day, you can think of a `vector` **both** as:

- a **sequence of n numbers**
- a **point** in a **n-dimensional space** 🎉

1.3.2) Vectors are everywhere in data science!

👉 Imagine a `dataframe` of data on apartments. For each `apartment` (row), 3️⃣ numerical `features` (columns) are available.



👉 Each `apartment` is represented by a `vector` of size 3, or a `point` in a 3D space. 😲 So the whole dataset can be viewed as a `point cloud` in that 3D space.

🤔 Now what if there are *more* than 3 features?

1.3.3) Vectors and basic operations with `NumPy`

In [ ]:
```
u = np.array([1, 2, 3])              # shape (3,)
u_row = np.array([[1, 2, 4]])        # shape (1, 3) (row vector)
u_column = np.array([[1], [2], [3]]) # shape (3, 1) (column vector)
```

👉 Add two `vectors`

In [ ]:
```
v = np.array([4, 5, 6])
u + v
```

Out[ ]:
```
array([5, 7, 9])
```

👉 Multiply a `vector` by a `scalar` (⚠️ ≠ scalar product ⚠️)

```
In [ ]: 2 * u
```

```
Out[ ]: array([2, 4, 6])
```

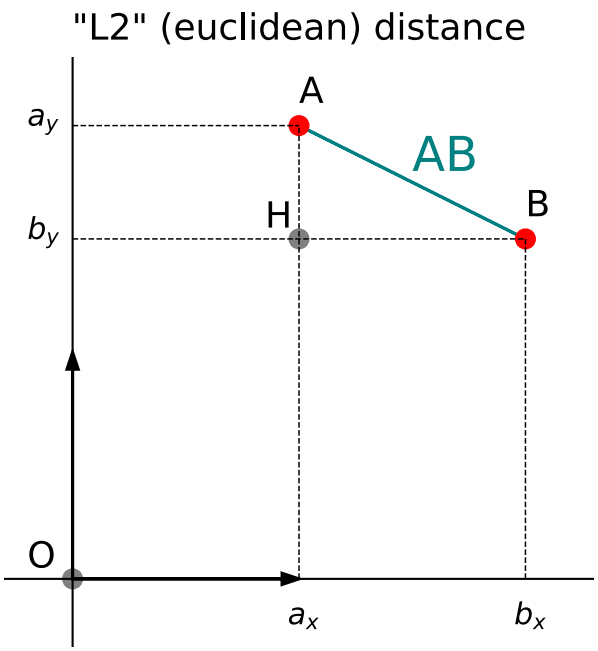
1.3.4) Distance between two vectors - Norm of a vector

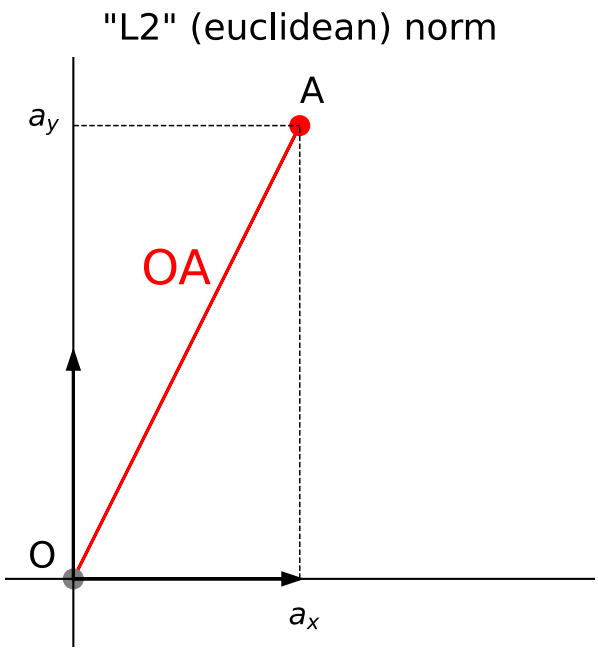
👉 The `L2` **distance** between `vectors` $\boldsymbol{a}$ and $\boldsymbol{b}$ is the `euclidean` distance 📏, AB from point A (a_x, a_y) to point B (b_x, b_y).

👉 The `L2` **norm** of `vector` $\boldsymbol{a}$, denoted $||\boldsymbol{a}||_2$, is the distance from O $(0, 0)$ to A (a_x, a_y).

💡 Since $\vec{AB} = \vec{OB} - \vec{OA}$, the `L2` **distance** between $\boldsymbol{a}$ and $\boldsymbol{b}$ is actually the `L2` **norm** of `vector` $\boldsymbol{b} - \boldsymbol{a}$:

$$AB = ||\boldsymbol{b} - \boldsymbol{a}||_2$$





According to Pythagoras' theorem:

$$AB = \sqrt{(b_x - a_x)^2 + (b_y - a_y)^2}$$

$$OA = ||\boldsymbol{a}||_2 = \sqrt{{a_x}^2 + {a_y}^2}$$

🤔 Can we measure 📏 distances in a **n-dimensional** space?

$$OA = ||\boldsymbol{a}||_2 = \sqrt{{a_1}^2 + {a_2}^2 + \ldots + {a_n}^2} = \sqrt{\sum_{i=1}^{n} a_i^2}$$

$$AB = ||\boldsymbol{b} - \boldsymbol{a}||_2 = \sqrt{(b_1 - a_1)^2 + (b_2 - a_2)^2 + \ldots + (b_n - a_n)^2} = \sqrt{\sum_{i=1}^{n} (b_i - a_i)^2}$$

💡 You'll also encounter the `L1` (aka `Manhattan` or `cityblock`) **distance**:

$$||\boldsymbol{b} - \boldsymbol{a}||_1 = |b_1 - a_1| + |b_2 - a_2| + \ldots + |b_n - a_n| = \sum_{i=1}^{n} |b_i - a_i|$$

📝 Let's compute the `L2` distance between $\boldsymbol{a} = [1 \quad 2 \quad 3 \quad 4]$ and $\boldsymbol{b} = [5 \quad 5 \quad 5 \quad 5]$

In [ ]:
```
a = np.array([1, 2, 3, 4])
b = np.array([5, 5, 5, 5]) # We both live in a 4D space!
```

In [ ]:
```
(b-a)
```

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

In [ ]:
```
(b-a)**2
```

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

In [ ]:
```
np.sqrt(np.sum((b-a)**2))
```

Out[ ]:
```
5.477225575051661
```

1.3.5) 🔥 Dot product (aka scalar product aka inner product) 🔥

👉 the **dot product** takes 2 input `vectors` (of same size) and outputs a `scalar` (a number)

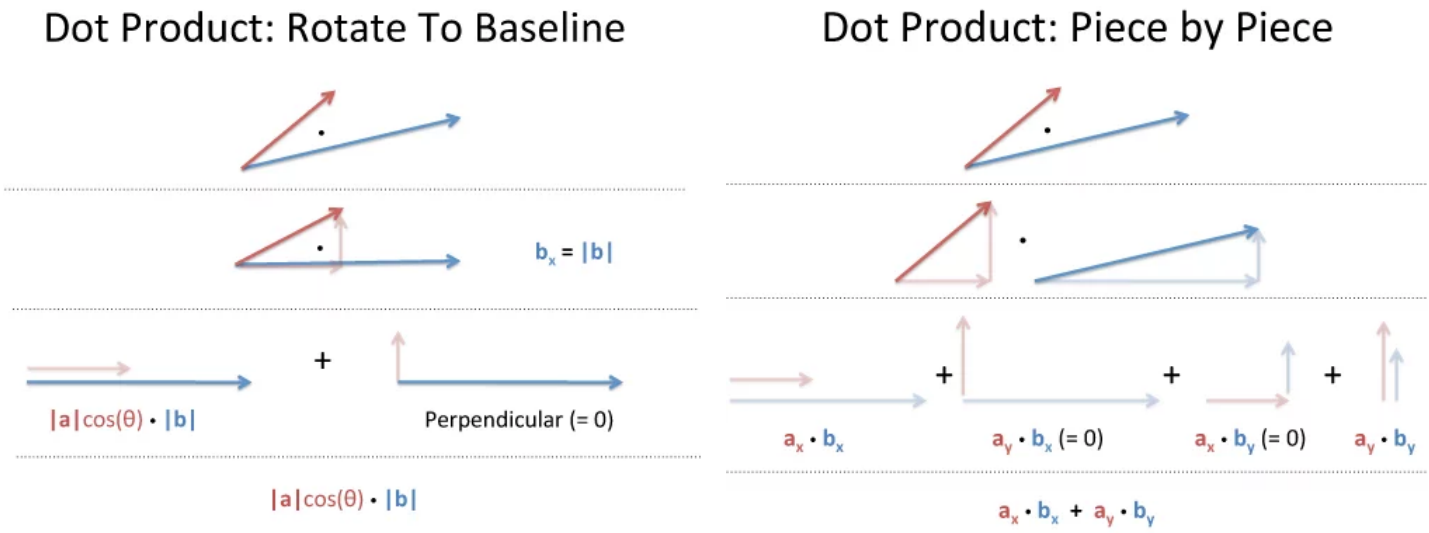
$$\boldsymbol{a} \cdot \boldsymbol{b}$$

👉 the **dot product** is maximal if the 2 vectors point in the **same direction**:

$$\boldsymbol{a} \cdot \boldsymbol{b} = ||\boldsymbol{a}||_2 \times ||\boldsymbol{b}||_2$$

👉 the **dot product** is null if the 2 vectors are **orthogonal**:

$$\boldsymbol{a} \cdot \boldsymbol{b} = 0$$



[better explained - dot product (https://betterexplained.com/articles/vector-calculus-understanding-the-dot-product/)](https://betterexplained.com/articles/vector-calculus-understanding-the-dot-product/)

Let $\boldsymbol{a}$ and $\boldsymbol{b}$ two vectors ⚠️ ***of identical size*** n.

👉 The **dot product** of $\boldsymbol{a}$ and $\boldsymbol{b}$ is:

$$\boldsymbol{a} \cdot \boldsymbol{b} = a_1 b_1 + a_2 b_2 + \ldots + a_n b_n = \sum_{i=1}^{n} a_i b_i$$

👉 The **dot product** is related to the **angle** θ formed by $\boldsymbol{a}$ and $\boldsymbol{b}$:

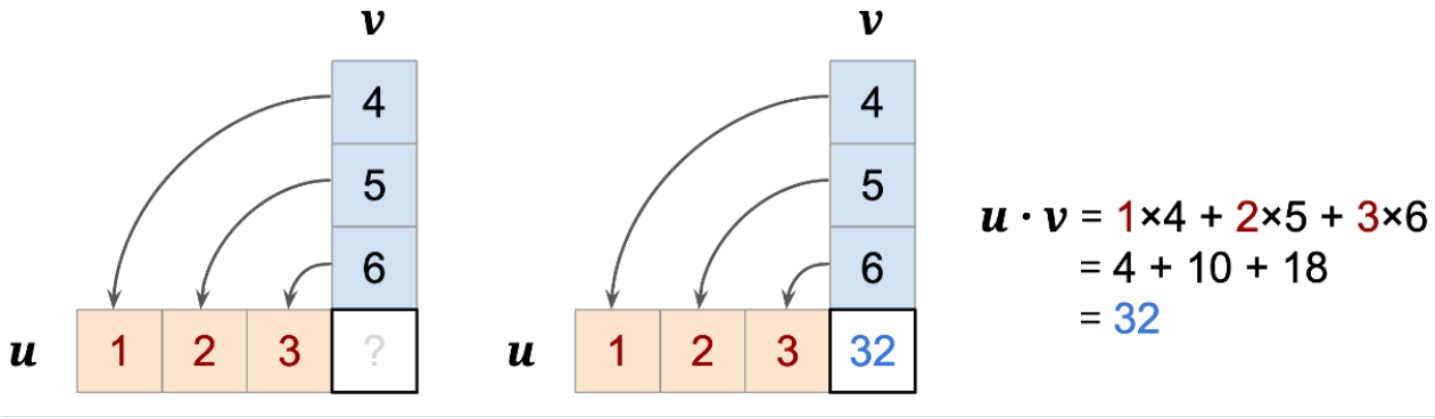
$$\boldsymbol{a} \cdot \boldsymbol{b} = ||a|| \times ||b|| \times \cos\theta$$

💡 $\cos\theta$, aka the `cosine similarity` , is often used in NLP (Language models)

🧱 Let's compute a `dot product` by hand!

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \cdot \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = ?$$

✅ The *size* of the left `vector` is equal to the *size* of the right `vector` ✅



📝 With `numpy` :

In [ ]:
```
np.dot(u, v)   # or...
u.dot(v)       # or...
u @ v
```

1.4) Matrices

👉 A `matrix` is a **two-dimensional** array of `scalars` .

The `symbol` of a matrix is a **bold**, *italic, uppercase* letter

Example:

$$\boldsymbol{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ a_{3,1} & a_{3,2} & \dots & a_{3,n} \\ \dots & \dots & \dots & \dots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{bmatrix}$$

$\boldsymbol{A}$ is a matrix with m rows and n columns (shape $m \times n$)

A `matrix` can be:

👉 A **collection** of `column vectors` stacked horizontally

👉 A **collection** `row vectors` stacked vertically

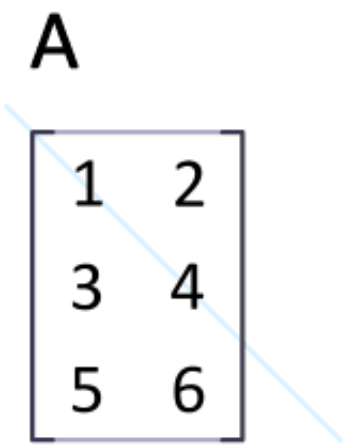
👉 Any **numerical table** (pandas dataframe...) is a matrix

💡 A **grayscale picture** is also a matrix of numbers representing the black intensity of pixels

Basic operations with matrices

👉 Transpose a matrix

The transpose of matrix $\boldsymbol{A}$ is denoted $\boldsymbol{A}^T$



👉 Add two matrices

Let $\boldsymbol{A}$ and $\boldsymbol{B}$ two matrices ⚠️ ***of identical shape.***

$\boldsymbol{A} + \boldsymbol{B}$ is a `matrix` of same shape, computed by element-wise addition.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} = \begin{bmatrix} 2 & 6 \\ 5 & 9 \\ 8 & 12 \end{bmatrix}$$

👉 Multiply a matrix by a scalar

The product $\lambda\boldsymbol{A}$ of a scalar $\lambda \in \mathbb{R}$ and a matrix $\boldsymbol{A}$ is computed by multiplying each element of $\boldsymbol{A}$ by λ.

$$3 \cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} = \begin{bmatrix} 3 & 6 \\ 9 & 12 \\ 15 & 18 \end{bmatrix}$$

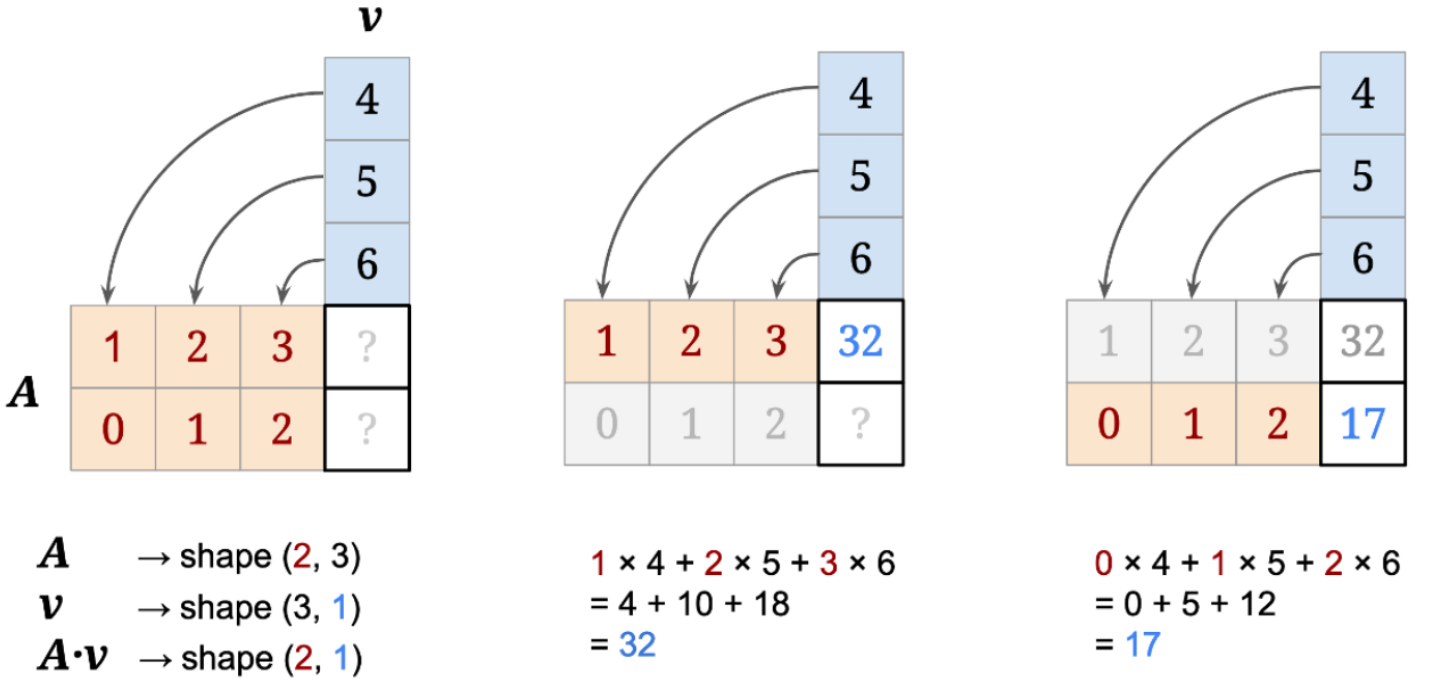
📝 With `numpy` :

In [ ]:
```
A.shape       # (n_rows, n_columns)
A.T           # transpose of A
A + B         # addition
3 * A         # scalar multiplication
```

🔥🔥 Matrix multiplication: Matrix · Vector 🔥🔥

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

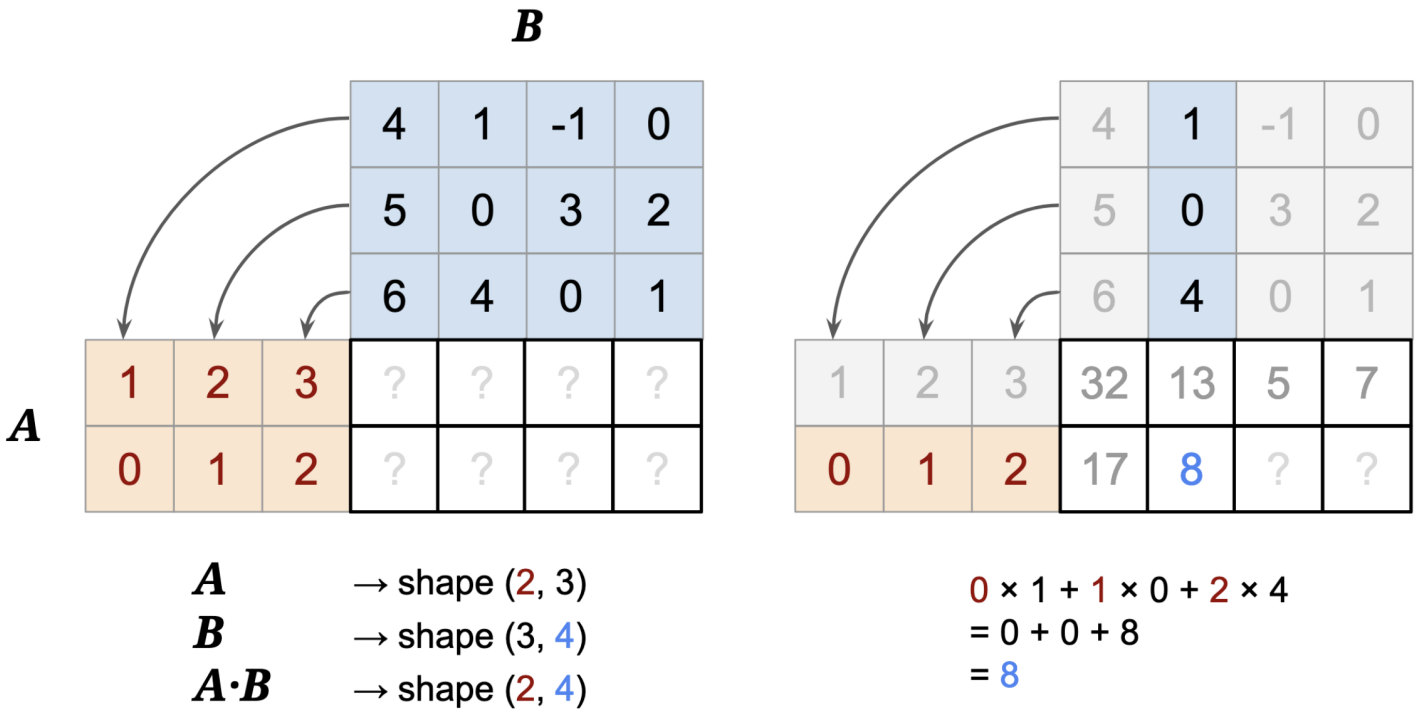
✅ The number of *columns* of the `matrix` is equal to the *size* (≈ number of "rows") of the `column vector` ✅



🔥🔥🔥 Matrix multiplication: Matrix · Matrix 🔥🔥🔥

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

✅ The number of *columns* of the left `matrix` is equal to the number of *rows* of the right `matrix` ✅



📝 With `numpy` :

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

B = np.array([[4, 1, -1, 0],
              [5, 0, 3, 2],
              [6, 4, 0, 1]])
```

In [ ]:
```
# Matrix • Matrix
np.matmul(A, B) # or...
A @ B
```

Out[ ]:
```
array([[32, 13,  5,  7],
       [17,  8,  3,  4]])
```

💥💥💥 Shape incompatibility 💥💥💥

🤔 What if...

❌ the number of `columns` of the left matrix is **NOT** equal to the number of `rows` of the right one? ❌

In [ ]:
```
print(f"{B.shape = } and {A.shape = }.")
```

```
B.shape = (3, 4) and A.shape = (2, 3).
```

In [ ]:
```
B @ A
```

```
---------------------------------------------------------------------
-------
ValueError                                Traceback (most recent cal
l last)
Cell In [28], line 1
----> 1 B @ A

ValueError: matmul: Input operand 1 has a mismatch in its core dimen
sion 0, with gufunc signature (n?,k),(k,m?)->(n?,m?) (size 2 is diff
erent from 4)
```

More on Matrix multiplication...

✅ `Associative` 👉 $(\boldsymbol{AB}) \cdot \boldsymbol{C} = \boldsymbol{A} \cdot (\boldsymbol{BC})$

✅ `Distributive` 👉 $\boldsymbol{A} \cdot (\boldsymbol{B} + \boldsymbol{C}) = \boldsymbol{AB} + \boldsymbol{AC}$

❌ but **NOT** `Commutative` 👉 $\boldsymbol{AB} \neq \boldsymbol{BA}$

Identity matrix

An `identity matrix` is a square matrix with a `diagonal of ones` and `zeros` everywhere else.

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

Provided the shapes of A and I are compatible:

$A \cdot I = A$ and $I \cdot A = A$

👉 `numpy.eye`

Inverse matrix

The `inverse of a matrix` A is a matrix denoted A^{-1} that satisfies:

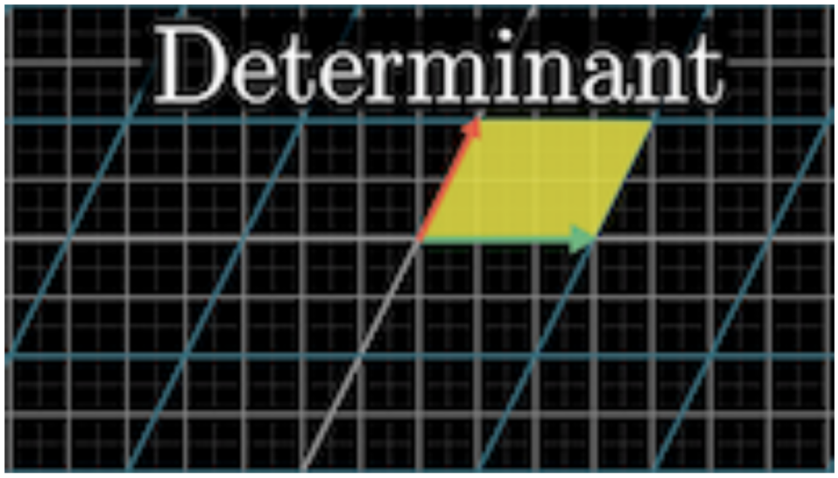
$$A^{-1} \cdot A = A \cdot A^{-1} = I$$

⚠️ Only `square matrices` can have an inverse... and [some square matrices don't have an inverse! (https://en.wikipedia.org/wiki/Invertible_matrix)](https://en.wikipedia.org/wiki/Invertible_matrix)

👉 `numpy.linalg.inv`

Determinant of a matrix

A square matrix is invertible *if and only if* its [determinant (https://en.wikipedia.org/wiki/Determinant)](https://en.wikipedia.org/wiki/Determinant) is not 0.



[Linear transformations and matrices (https://www.youtube.com/watch?v=kYB8IZa5AuE&list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab&index=3)](https://www.youtube.com/watch?v=kYB8IZa5AuE&list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab&index=3) by `3blue1brown` 🥰

👉 `numpy.linalg.det`

1.5) Matrix multiplication 🤝 `data science`

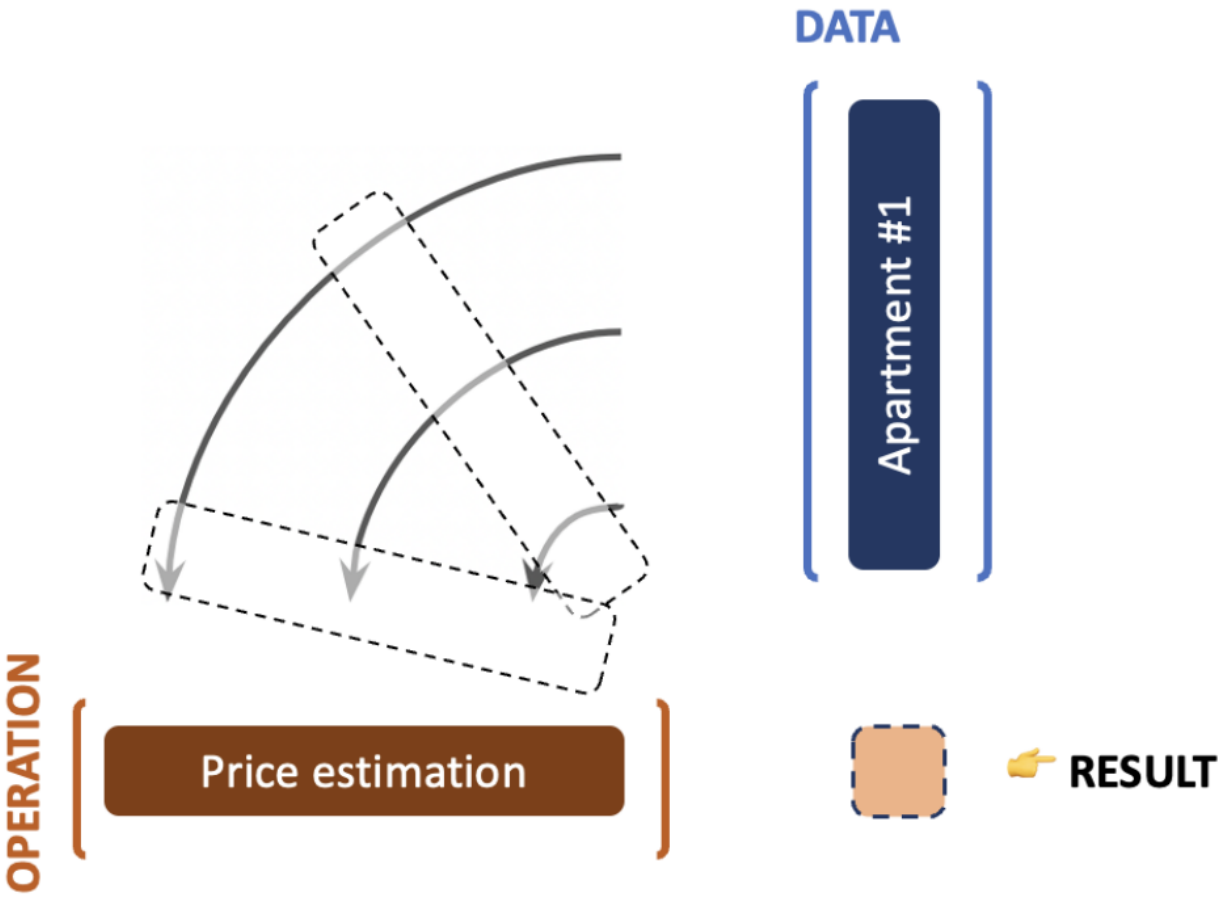
1.5.1) Concise code and efficient computation

$$y = a_1 x_1 + a_2 x_2 + \ldots + a_n x_n$$

... is equivalent to

$$y = \boldsymbol{a} \cdot \boldsymbol{x}$$

👉 Estimate the `price` of one apartment (y) by a linear combination (weights = $\boldsymbol{a}$) of the `features` ($\boldsymbol{x}$)
🔥 Vector • Vector 🔥



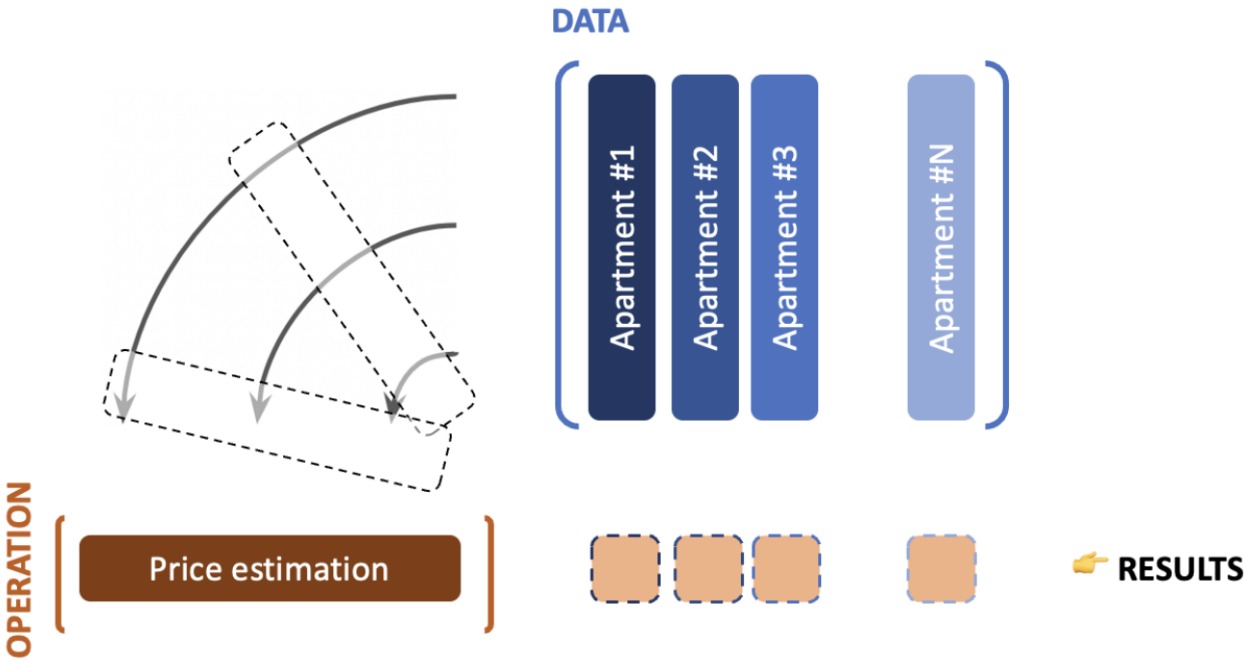
$$y = a_1 x_1 + a_2 x_2 + \ldots + a_n x_n = \boldsymbol{a} \cdot \boldsymbol{x}$$

$$\begin{cases} y_1 = a_1\ x_{1,1} + a_2\ x_{1,2} + \ldots + a_n\ x_{1,n} \\ y_2 = a_1\ x_{2,1} + a_2\ x_{2,2} + \ldots + a_n\ x_{2,n} \\ \quad\ldots \qquad\qquad \ldots \\ y_m = a_1\ x_{m,1} + a_2\ x_{m,2} + \ldots + a_n\ x_{m,n} \end{cases}$$

is equivalent to

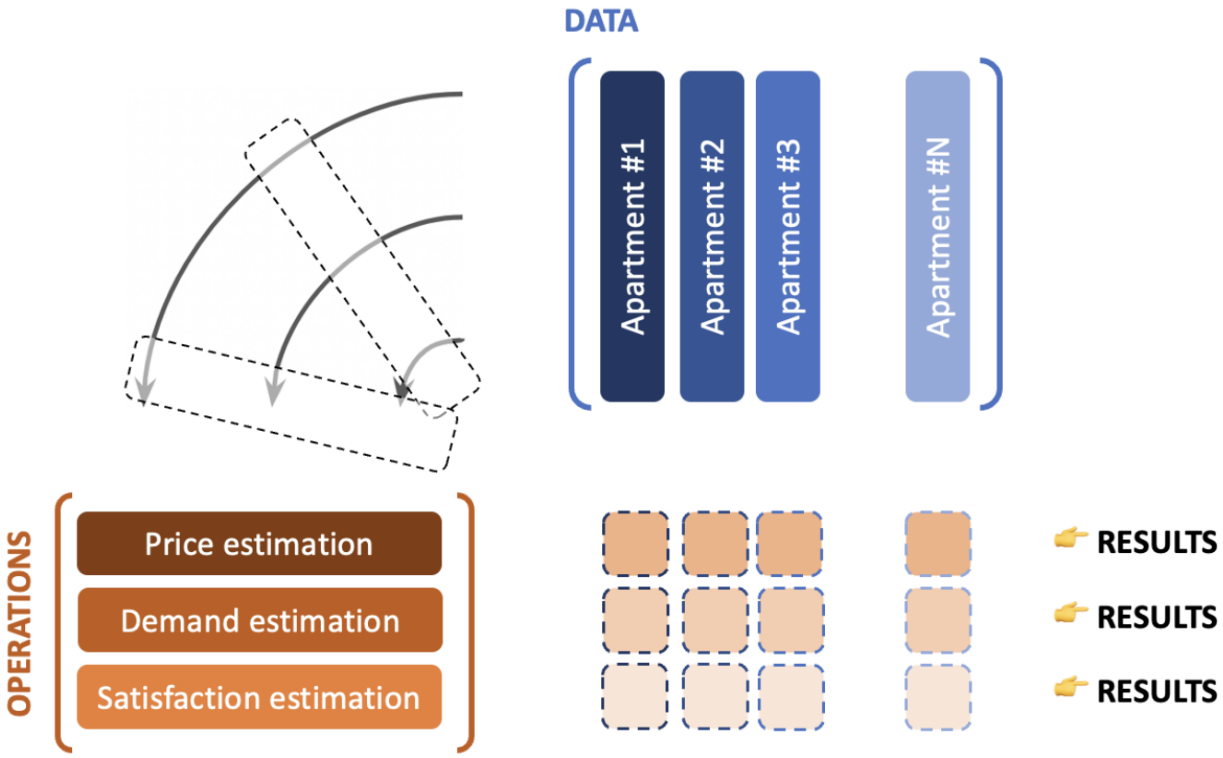
$$y = \boldsymbol{a} \cdot \boldsymbol{X}$$

👉 Estimate the `price` of *many* apartments ($\boldsymbol{y}$) 🔥🔥 Vector • Matrix 🔥🔥



$$y = a \cdot X$$

👉 Perform *many* operations on *many* apartments 🔥🔥🔥 Matrix • Matrix 🔥🔥🔥



$$\boldsymbol{Y} = \boldsymbol{A} \cdot \boldsymbol{X}$$

One last thing:

GPUs ❤️ `matrix multiplication`

1.5.2) Solving a system of linear equations

Solving the father/daughter problem (again)

$$\begin{cases} 1 \cdot x_1 + (-1) \cdot x_2 = 22 \\ 1 \cdot x_1 + (-2) \cdot x_2 = 10 \end{cases}$$

$\iff$ $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$ ... where: $\boldsymbol{A} = \begin{bmatrix} 1 & -1 \\ 1 & -2 \end{bmatrix}$, $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ and $\boldsymbol{b} = \begin{bmatrix} 22 \\ 10 \end{bmatrix}$

👉 Multiply both sides of the equation with $\boldsymbol{A}^{-1}$ on the left: $\boldsymbol{A}^{-1} \cdot (\boldsymbol{A} \cdot \boldsymbol{x}) = \boldsymbol{A}^{-1} \cdot \boldsymbol{b}$

👉 Notice that: $\boldsymbol{A^{-1} \cdot (A \cdot x) = (A^{-1} \cdot A) \cdot x = I_2 \cdot x = x}$

👉 finally: $\boldsymbol{x = A^{-1} \cdot b}$

📝 With `NumPy` :

In [ ]:
```
A = np.array([[1, -1], [1, -2]])
b = np.array([[22], [10]])
```

In [ ]:
```
np.linalg.inv(A) @ b
```

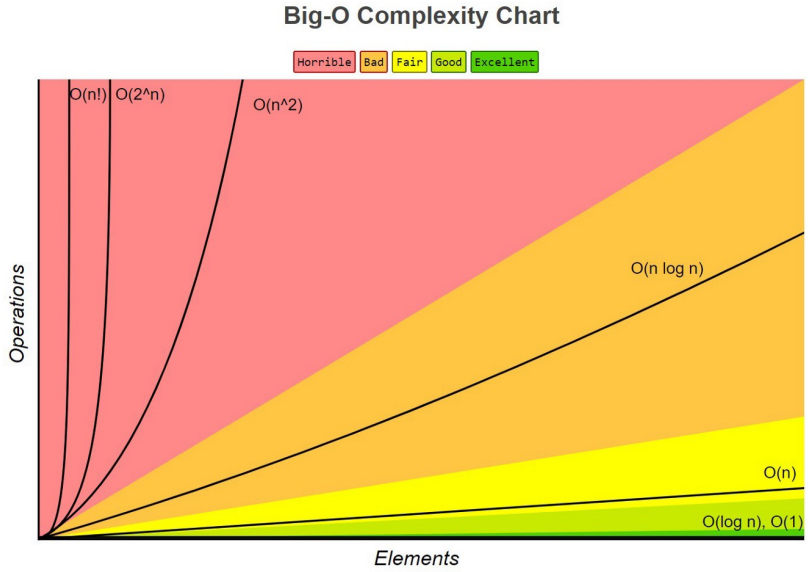
Out[ ]:
```
array([[34.],
       [12.]])
```

How did Numpy compute A^{-1} ?

We can use the **Gaussian elimination** (https://en.wikipedia.org/wiki/Gaussian_elimination#Finding_the_inverse_of_a_matrix) algorithm which has a $O(n^3)$ **complexity**.

This complexity can still be improved (https://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations#Matrix_algebra) but we won't reach a complexity that is better than $O(n^{2.3})$ 😱

Computational complexity - "Big O" 🤔

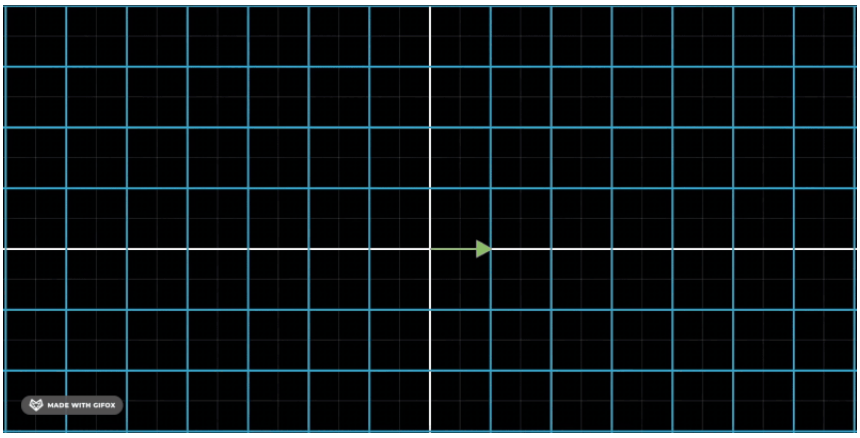


- $O(1)$ - get first element of a list
- $O(log(n))$ - find position of given element in a sorted list (divide & conquer)
- $O(n)$ - print each elements of a list (simple loop)
- $O(nlog(n))$ - sort a list
- $O(n^2)$ - nested `for` loop
- $O(2^n)$ - password cracking (brute force) or recursive functions

📚 Towards Data Science - understanding time complexity (https://medium.com/towards-data-science/understanding-time-complexity-with-python-examples-2bda6e8158a7)

👉 Recent research from DeepMind (https://www.deepmind.com/blog/discovering-novel-algorithms-with-alphatensor)

1.5.3) Change of basis and Projection



Linear transformations and matrices (https://www.youtube.com/watch?v=kYB8IZa5AuE&list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab&index=3) by `3blue1brown` 🥰

`Matrix Multiplication` can represent a Change of Basis (https://en.wikipedia.org/wiki/Change_of_basis) or a Projection (https://en.wikipedia.org/wiki/Projection_%28linear_algebra%29).

$\boldsymbol{i}$ becomes $\begin{bmatrix} a & c \\ b & d \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix}$ while $\boldsymbol{j}$ becomes $\begin{bmatrix} a & c \\ b & d \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} c \\ d \end{bmatrix}$

💡 See PCA in the `Unsupervised Learning` Lecture (Week 5)

2) Calculus

We will use:

- **derivatives** and **partial derivatives** in `optimization` problems
- **integrals** for `probability` distribution

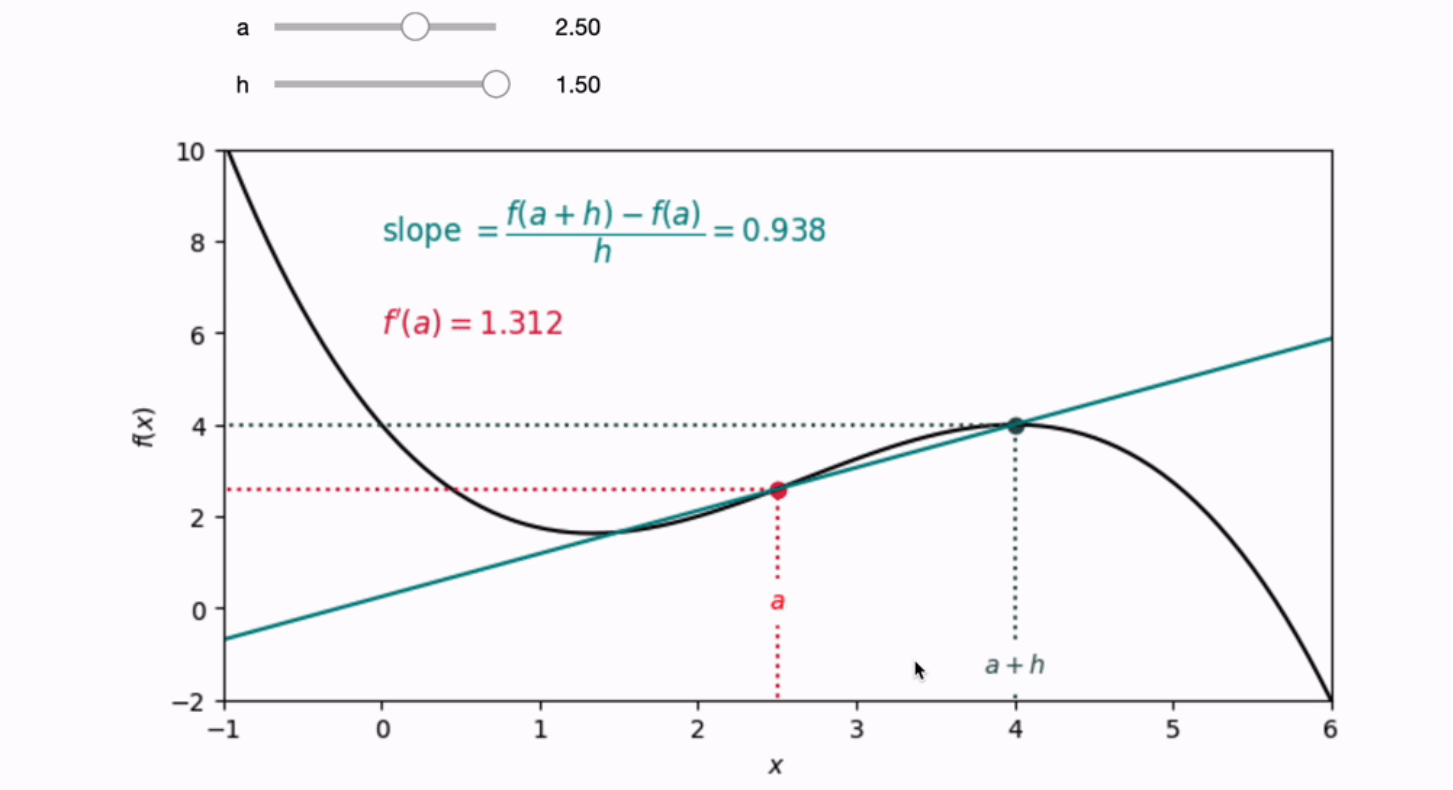
2.1) Derivatives

🤔 When x changes, how does $f(x)$ changes? This information is provided by f' the derivative (https://en.wikipedia.org/wiki/Derivative) of f

👉 Let $f: \mathbb{R} \to \mathbb{R}$ a function

$$f'(a) = \frac{df}{dx}(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$$

💡 The derivative at one point is the local **slope** of the curve



2.2) Partial derivatives

🤔 When x and y change, how does $g(x, y)$ changes? In that case we split the problem!

Let $g: \mathbb{R}^2 \to \mathbb{R}$ a continuous bivariate function. The [partial derivatives (https://en.wikipedia.org/wiki/Partial_derivative)](https://en.wikipedia.org/wiki/Partial_derivative) of g are:

$$\frac{\partial g}{\partial x}(x_A, y_A) = \lim_{h \to 0} \frac{g(x_A + h, y_A) - g(x_A, y_A)}{h}$$

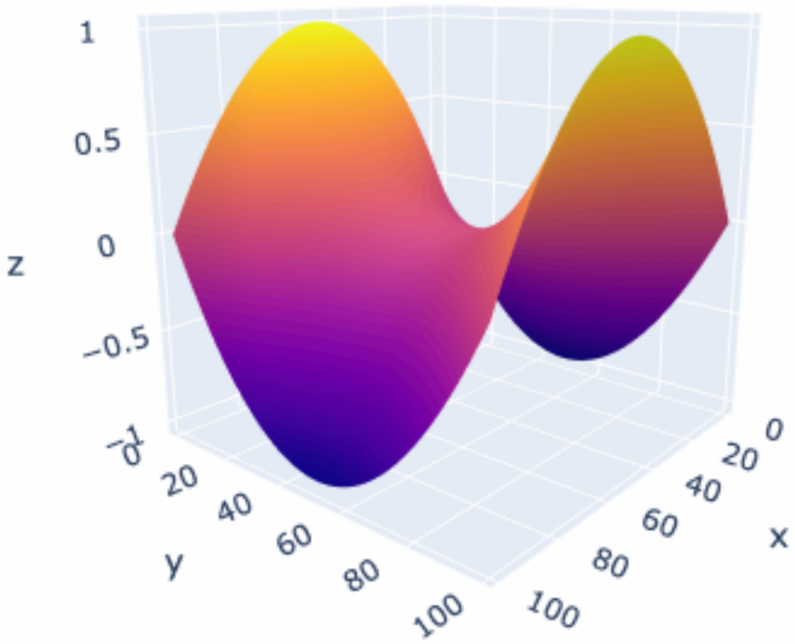
and

$$\frac{\partial g}{\partial y}(x_A, y_A) = \lim_{h \to 0} \frac{g(x_A, y_A + h) - g(x_A, y_A)}{h}$$

👉 the `partial derivatives` at one point are the **slopes** (or rates of change) with respect to the "x" direction and to the "y" direction

In [ ]:
```
import plotly.graph_objects as go
fun = lambda x, y: x**2 - y**2
x, y = np.meshgrid(np.linspace(-1, 1, 100), np.linspace(-1, 1, 100))
z = fun(x, y)

fig = go.Figure(go.Surface(x=x, y=y, z=z)); fig.show()
```



💡 The gradient vector (https://en.wikipedia.org/wiki/Gradient) contains all the `partial derivatives` of a **multi-variate function**.

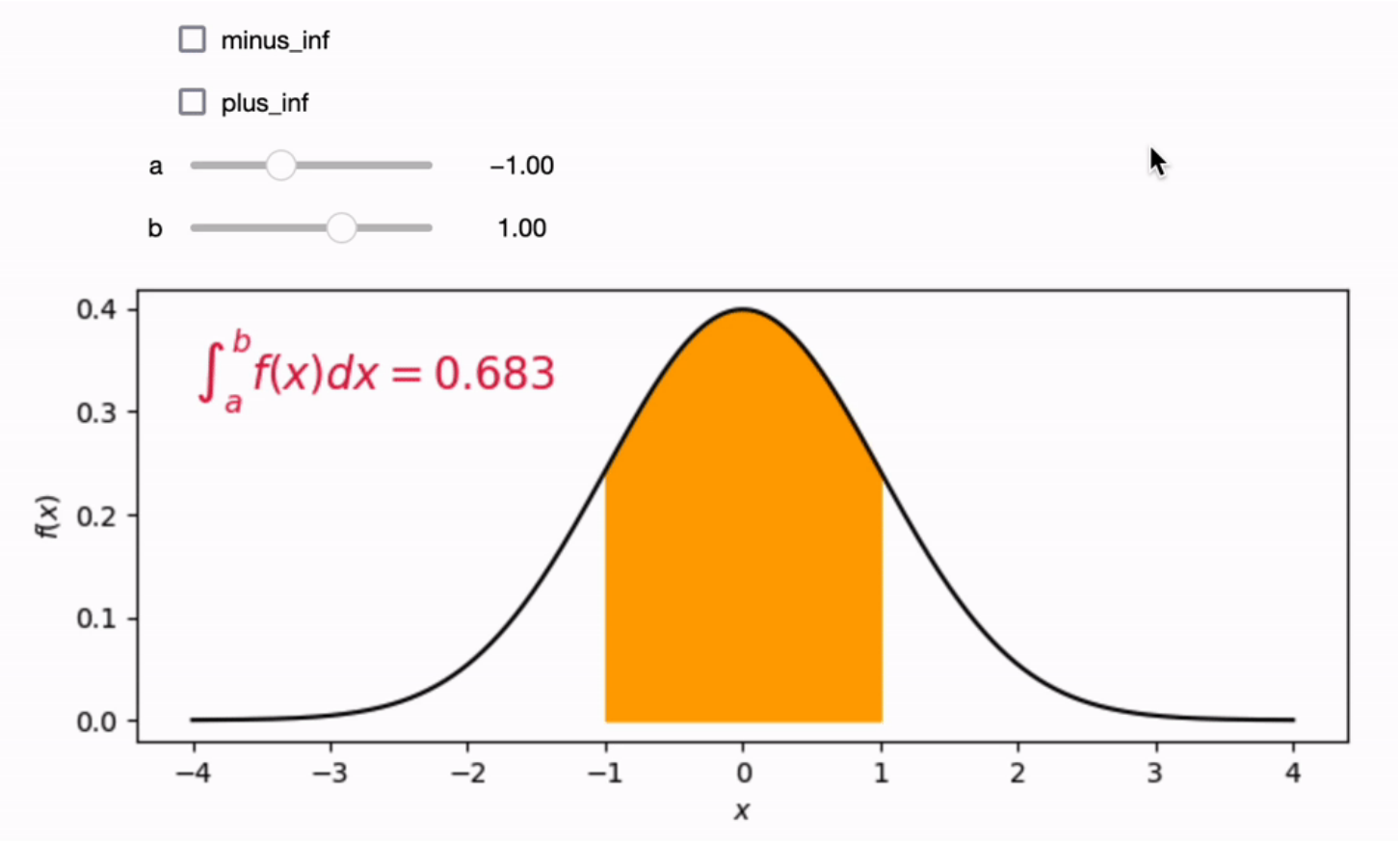
$$\nabla h(p) = \begin{bmatrix} \frac{\partial h}{\partial x_1}(p) \\ \frac{\partial h}{\partial x_2}(p) \\ \dots \\ \frac{\partial h}{\partial x_n}(p) \end{bmatrix}$$

2.3) Integrals

👉 The integral (https://en.wikipedia.org/wiki/Integral) of f from a to b is denoted as:

$$\int_a^b f(x)dx$$

💡 It can be viewed as the *area under the curve* $y = f(x)$ delimited by the verticals $x = a$ and $x = b$



Bibliography

📚 Hadrien Jean - Linear Algebra Series (https://hadrienj.github.io/deep-learning-book-series-home/) (Blog) + Essential Math for Data Science (https://www.essentialmathfordatascience.com/) (O'Reilly)

📺 3Blue1Brown - Essence of Linear Algebra (https://www.youtube.com/watch?v=fNk_zzaMoSs&list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab) (3h) and Essence of Calculus (https://www.youtube.com/watch?v=WUvTyaaNkzM&list=PLZHQObOWTQDMsr9K-rj53DwVRMYO3t5Yr) (4h)

📚 David Bessis - Mathematica (https://www.seuil.com/ouvrage/mathematica-david-bessis/9782021493979) (🇫🇷 2022) - Will be translated in 🇬🇧 - summary in a blog post 🇬🇧 (https://www.startup-book.com/2022/03/)

Your turn!