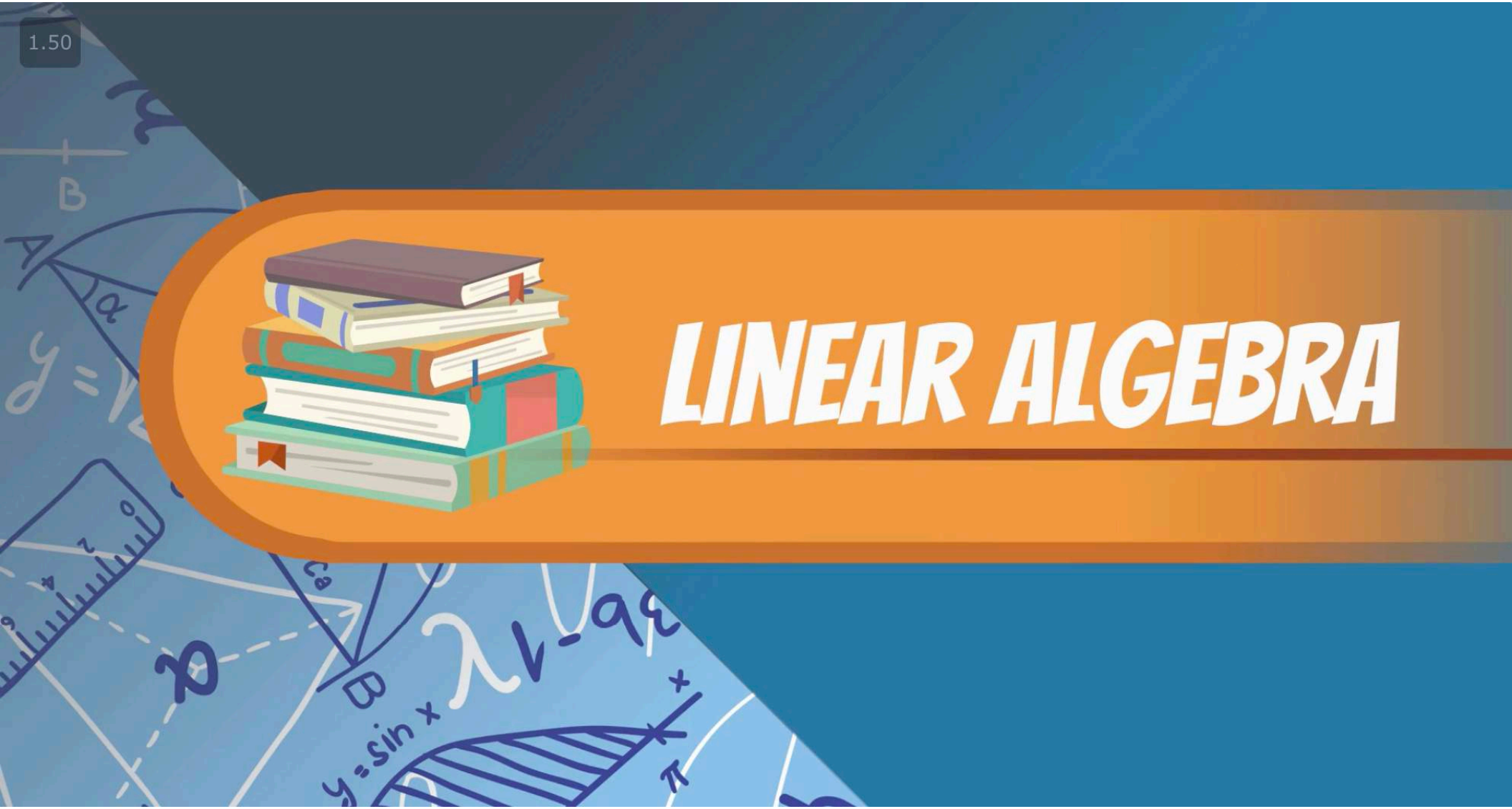
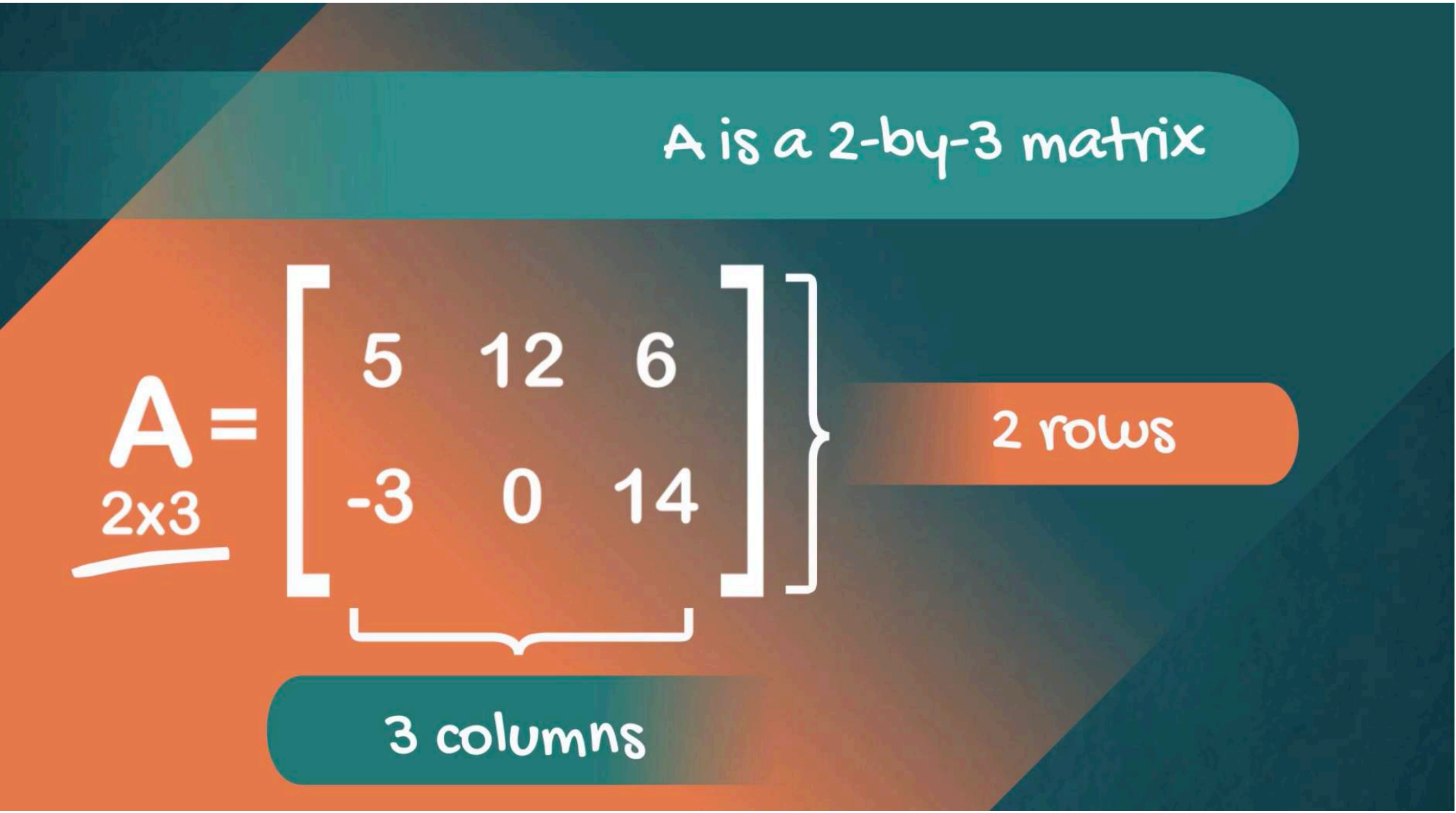

1.50
LINEAR ALGEBRA

A matrix is a collection of numbers ordered in rows and columns. Here is one.


A is a 2-by-3 matrix
A = 5 12 6 / -3 0 14
2x3
2 rows
3 columns

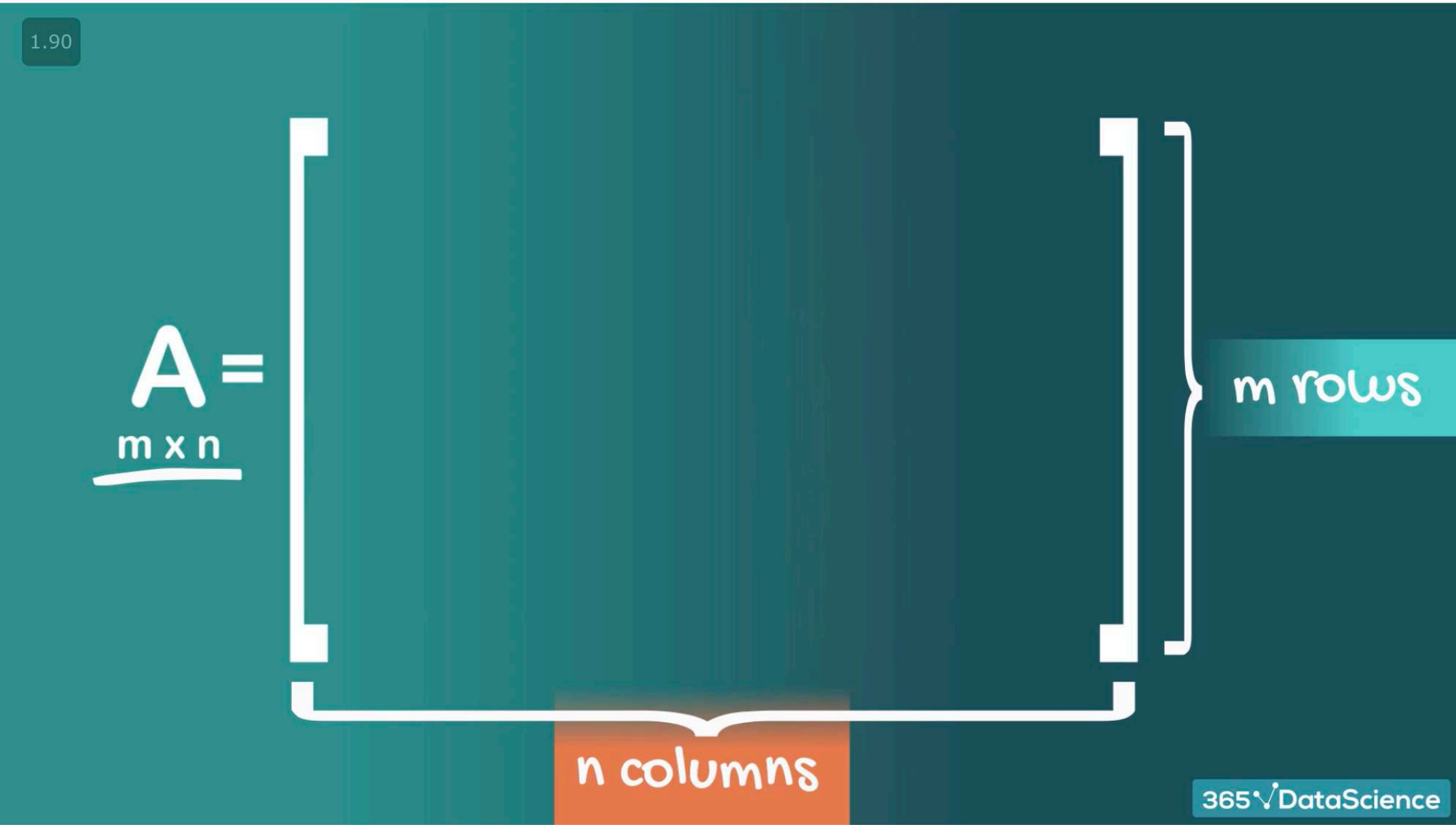
1.90

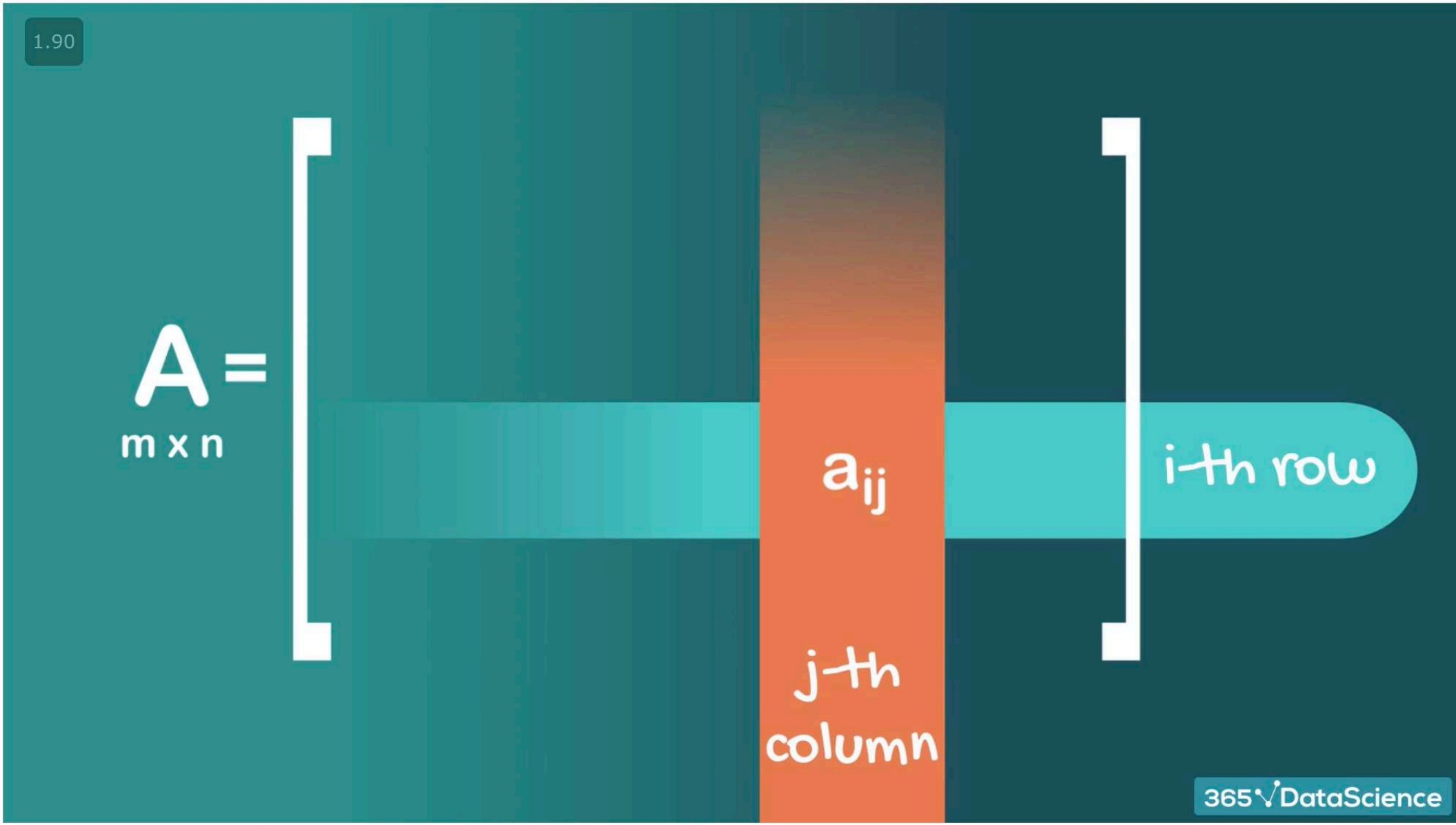
A matrix can only contain numbers, symbols, or expressions

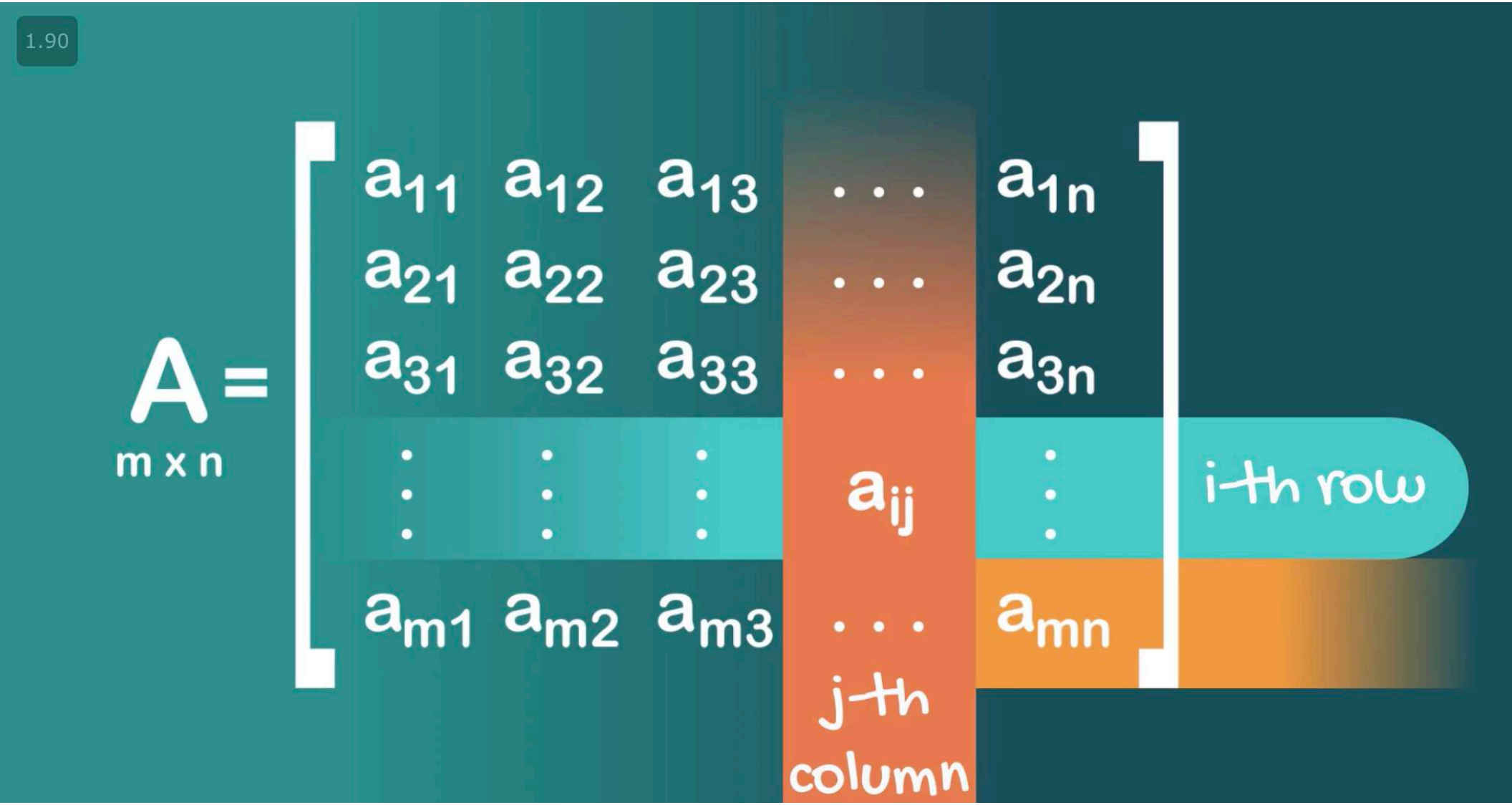
$$A_{2x3} = \begin{bmatrix} 5 & 12 & 6 \\ -3 & 0 & 14 \end{bmatrix}$$

$$B_{2x3} = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

$$C_{2x2} = \begin{bmatrix} a-x & b+5 \\ d+e & e \end{bmatrix}$$


1.90
A =
m x n
m rows
n columns
365 DataScience


1.90
A =
m x n
a_{ij}
i-th row
j-th
column
365 DataScience


1.90
A =
m x n
a_{11} a_{12} a_{13} . . . a_{1n}
a_{21} a_{22} a_{23} . . . a_{2n}
a_{31} a_{32} a_{33} . . . a_{3n}
a_{ij}
a_{m1} a_{m2} a_{m3} . . . a_{mn}
i-th row
j-th column

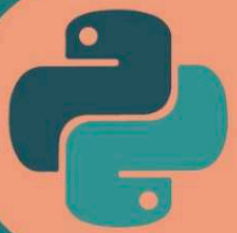
1.90

MATHEMATICS

PROGRAMING

$$A_{m \times n} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & a_{ij} & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix}$$

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0(n-1)} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1(n-1)} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{(m-1)0} & a_{(m-1)1} & a_{(m-1)2} & \cdots & a_{(m-1)(n-1)} \end{bmatrix}$$



365 DataScience

1.90

MATRICES

$$A_{m \times n} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & a_{ij} & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix}$$

rows

columns

2D

1.90

MATRICES = SCALARS

SCALARS HAVE 0 DIMENSIONS

$$\underbrace{\left[\, a_{11} \,\right]}_{\text{1 column}} \Big\} \text{1 row}$$

[15]; [1]; [2]; [-5]; []

All numbers we know from algebra are referred to as scalars in linear algebra

VECTORS

VECTORS

$$U = \begin{bmatrix} 5 \\ -2 \\ 4 \end{bmatrix}$$

3×1

3 rows

1 column

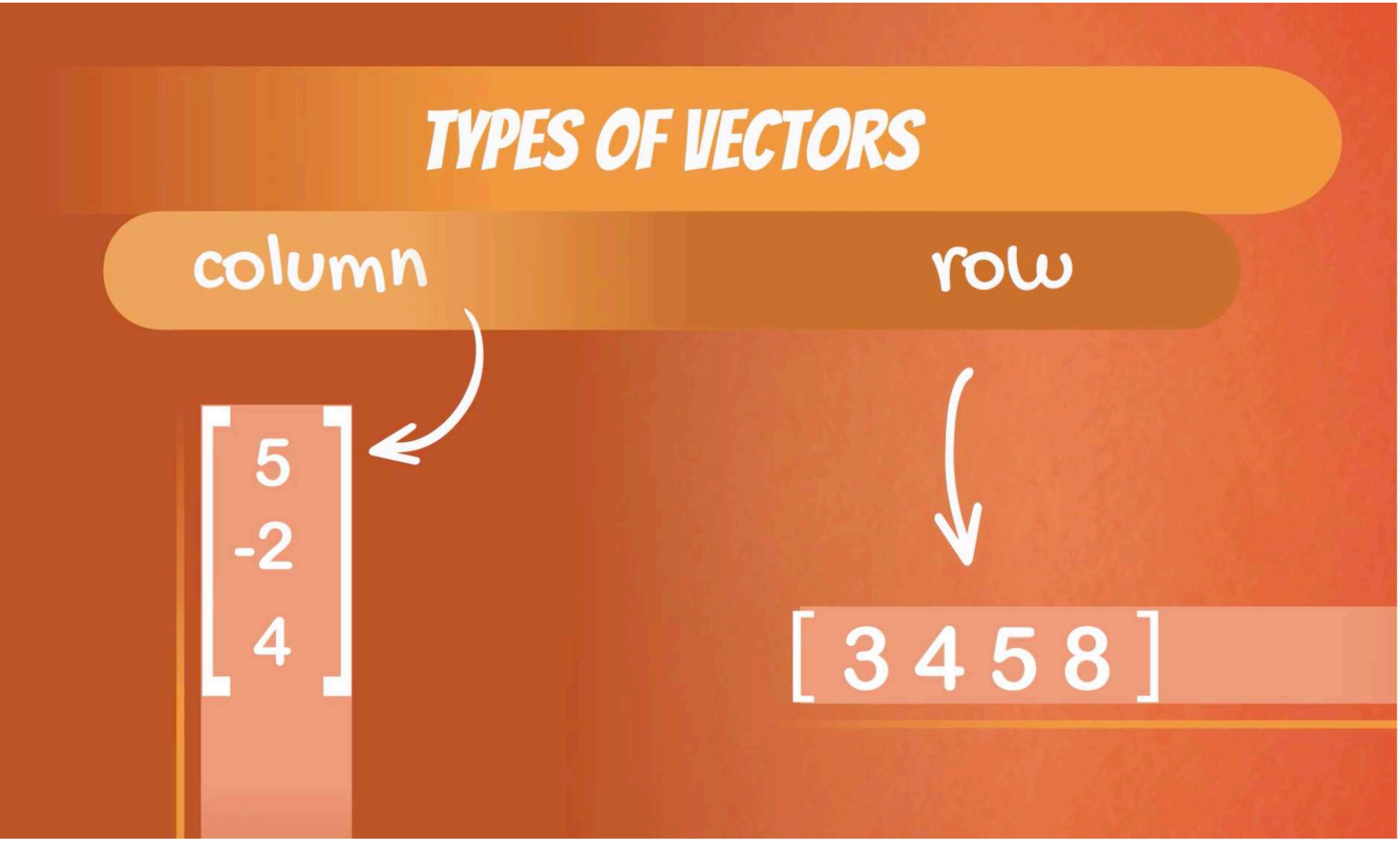
VECTORS

A vector is practically the simplest linear algebraic object

$$\begin{bmatrix} 5 \\ -2 \\ 4 \end{bmatrix}$$

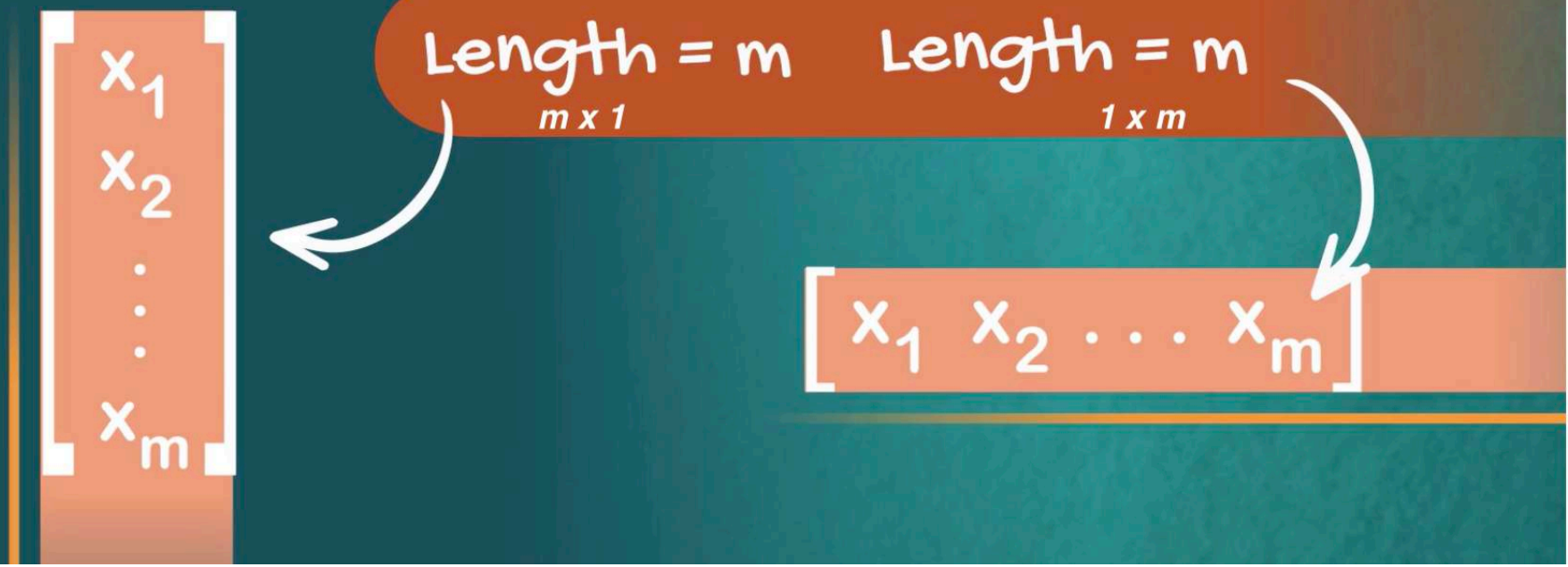
$$\begin{bmatrix} 5 & 12 & 6 \\ -3 & 0 & 14 \end{bmatrix}$$

vect. 1 vect. 2 vect. 3


TYPES OF VECTORS
column
row
5
-2
4
[3 4 5 8]

LENGTH

Df: The number of elements in a vector



SUMMARY

$$A_{m \times n} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & a_{ij} & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix}$$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

$$\begin{bmatrix} x \end{bmatrix}$$

2D Matrix
m x n

1D Vector
m x 1

0D Scalar
1 x 1

MATHEMATICS WORKS BASED ON ANALOGIES

WITH LINEAR ALGEBRA THAT'S EXTREMELY RELEVANT

LINEAR ALGEBRA AND GEOMETRY

0D Scalar
1 x 1

0D Point

LINEAR ALGEBRA AND GEOMETRY

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

1D Vector $m \times 1$

1D Line

LINEAR ALGEBRA AND GEOMETRY

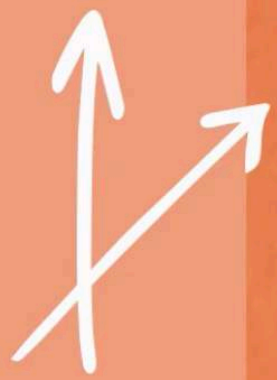
$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \quad \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

2D Vectors $m \times 1$ $m \times 1$

2D Line *(Plane)*

LINEAR ALGEBRA AND GEOMETRY

$$\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_m & y_m \end{bmatrix}$$



2D

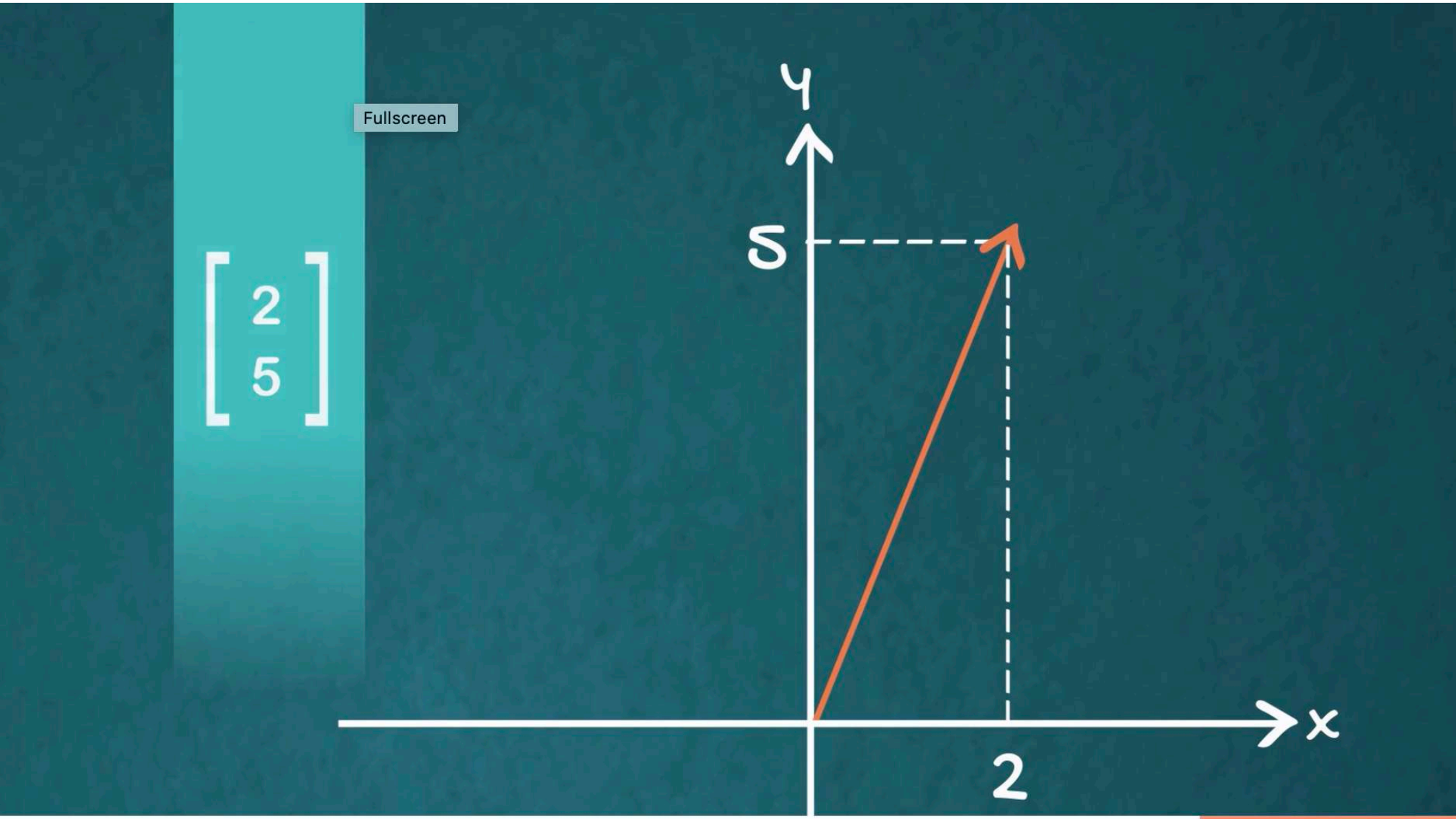
Matrix

m x 2

2D

Line

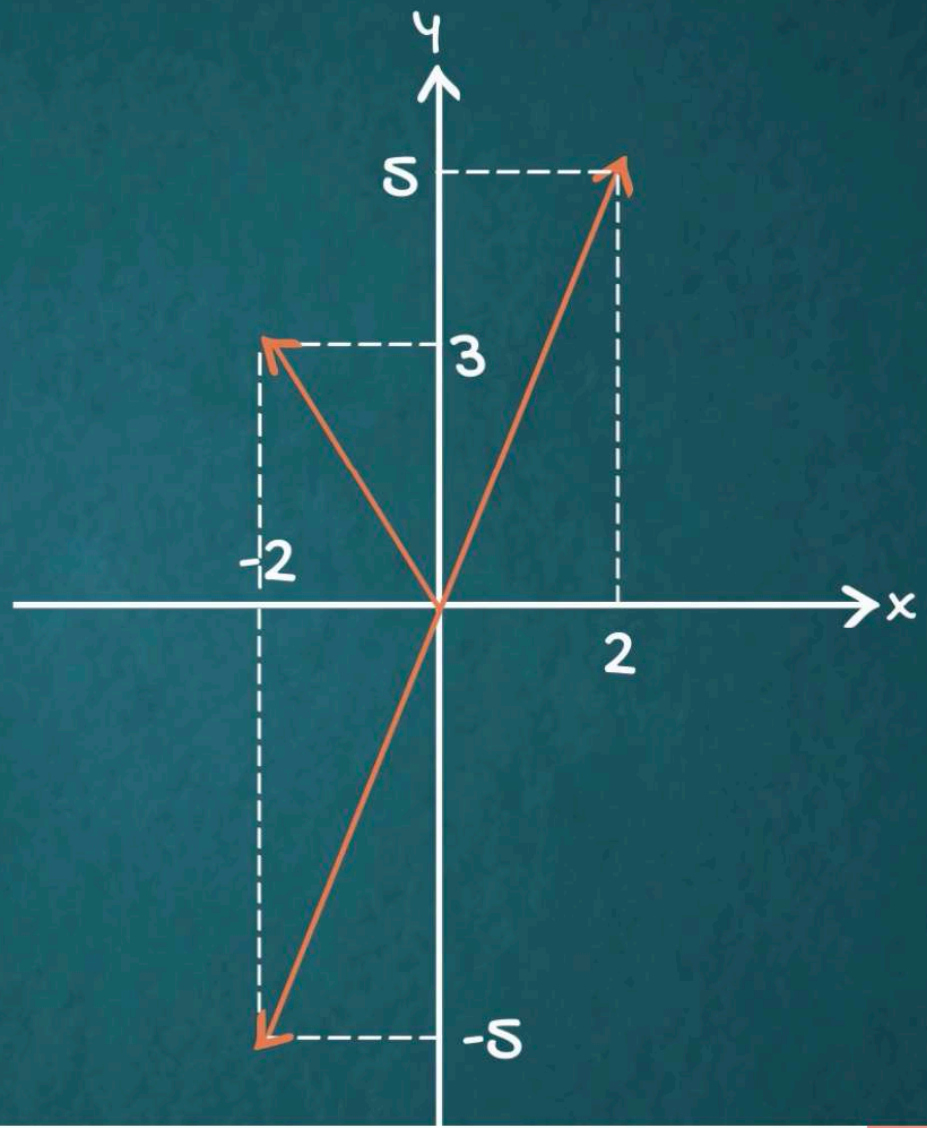
(Plane)


Fullscreen
2
5
y
5
x
2

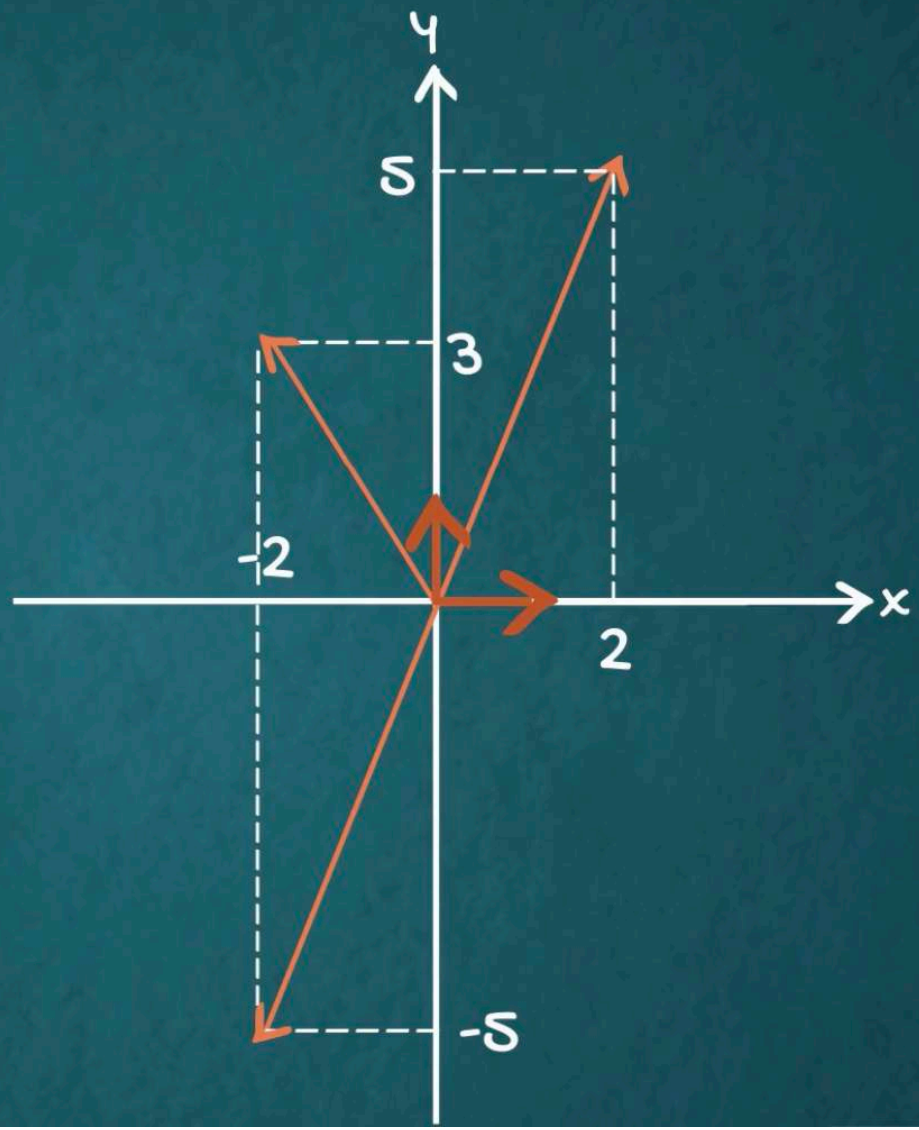
$$\begin{bmatrix} 2 \\ 5 \end{bmatrix}$$

$$\begin{bmatrix} -2 \\ -5 \end{bmatrix}$$

$$\begin{bmatrix} -2 \\ 3 \end{bmatrix}$$



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




3D VECTOR PLOTTER
BY ACADEMO.ORG
In this way, you will get a better grasp of the geometrical concept

Import the relevant libraries

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

The simplest most flexible way to work with matrices in Python is by using arrays

Declaring scalars, vectors, and matrices

Scalars

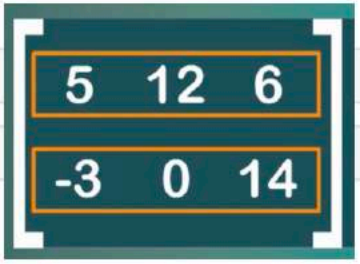
```
In [2]: s = 5

In [3]: s
Out[3]: 5
```

Vectors

By default, 'v' is a row vector


```
In [4]: v = np.array([5,-2,4])

In [5]: v
Out[5]: array([ 5, -2,  4])
```

Matrices




```
In [6]: m = np.array([[5,12,6],[-3,0,14]])

In [7]: m
Out[7]: array([[ 5, 12,  6],
               [-3,  0, 14]])
```

Data types

```
In [8]: type(s)
Out[8]: int

In [9]: type(v)
Out[9]: numpy.ndarray

In [10]: type(m)
Out[10]: numpy.ndarray

In [11]: s_array = np.array(5)
         s_array
Out[11]: array(5)

In [12]: type(s_array)
Out[12]: numpy.ndarray
```

ndarray = n-dimensional array

Data shapes

```
In [13]: m.shape
Out[13]: (2, 3)

In [14]: v.shape
Out[14]: (3,)
```

shape returns the shape (dimensions) of a variable

In the memory of the computer, this object has 3 (ordered) elements

Creating a column vector

```
In [17]: v.reshape(1,3)
Out[17]: array([[ 5, -2,  4]])

In [18]: v.reshape(3,1)
Out[18]: array([[ 5],
                [-2],
                [ 4]])
```

reshape gives an array a new shape, without changing its data

```
In [15]: s.shape
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-15-1f0fd037afc4> in <module>()
----> 1 s.shape

AttributeError: 'int' object has no attribute 'shape'

In [16]: s_array.shape
Out[16]: ()
```

0-dimensional objects don't have shape


1.00
TENSORS

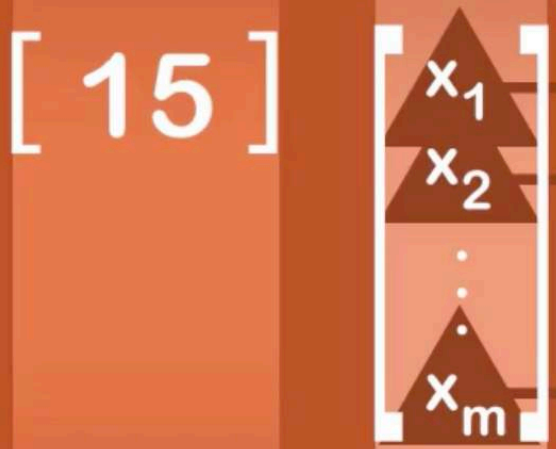
1.80

WHAT IS A TENSOR

[15]

- A vector of lenght 1
- A 1X1 matrix
- Scalar

WHAT IS A TENSOR

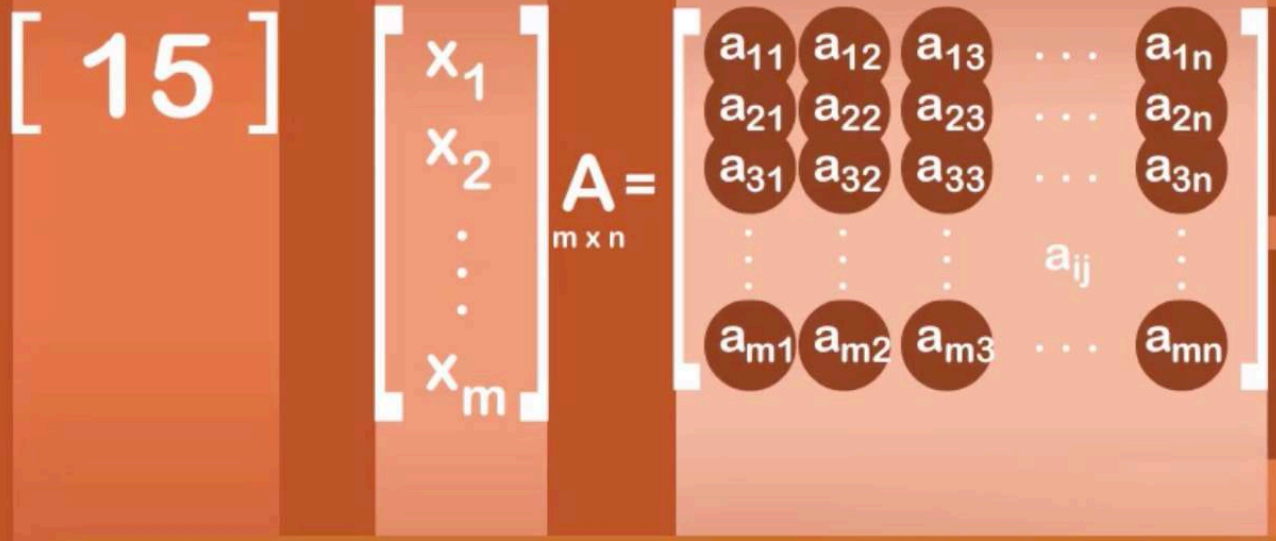


- each element is a scalar
- m x 1 or 1 x m matrix

Scalar 1x1 Vector

1.80

WHAT IS A TENSOR



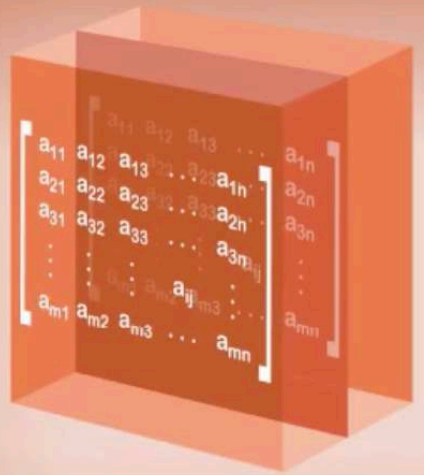
- a collection of vectors
- a collection of scalars

Scalar 1X1 | Vector m X 1 | Matrix m x n

1.80

TENSORS ARE SIMPLY A GENERALIZATION OF THE CONCEPTS WE HAVE SEEN SO FAR

$$[\,15\,] \qquad \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \qquad A_{m \times n} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & a_{ij} & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix}$$



RANK 0	RANK 1	RANK 2	RANK 3
Tensor	Tensor	Tensor	Tensor
1x1	m x 1	m x n	k x m x n

Tensors can be stored in ndarrays, and that's how we often deal with the issue

Import the relevant libraries

In [1]:
```
import numpy as np
```

Creating a tensor

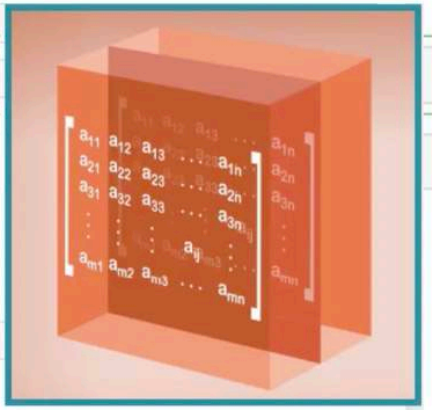
In [2]:
```
m1 = np.array([[5,12,6],[-3,0,14]])
m1
```

Out[2]:
```
array([[ 5, 12,  6],
       [-3,  0, 14]])
```

In [3]:
```
m2 = np.array([[9,8,7],[1,3,-5]])
m2
```

Out[3]:
```
array([[ 9,  8,  7],
       [ 1,  3, -5]])
```

In [ ]:
```
t = np.array([m1,m2])
```



In [5]:
```
t
```

Out[5]:
```
array([[[ 5, 12,  6],
        [-3,  0, 14]],

       [[ 9,  8,  7],
        [ 1,  3, -5]]])
```

Checking its shape

In [6]:
```
t.shape
```

Out[6]:
```
(2, 2, 3)
```

Manually creating a tensor

In [7]:
```
t_manual = np.array([[[ 5, 12,  6], [-3,  0, 14]], [[ 9,  8,  7], [ 1,  3, -5]]])
```

ADDITION AND SUBTRACTION

ADDITION

Very easy

1 Condition:

The two matrices must have the same dimensions

$$\underset{2x3}{M_1} + \underset{2x3}{M_2} = \underset{2x3}{X}$$

$$\begin{bmatrix} 5 & 12 & 6 \\ -3 & 0 & 14 \end{bmatrix} + \begin{bmatrix} 9 & 8 & 7 \\ 1 & 3 & -5 \end{bmatrix} = \begin{bmatrix} 5+9 & 12+8 & 6+7 \\ -3+1 & 0+3 & 14-5 \end{bmatrix}$$

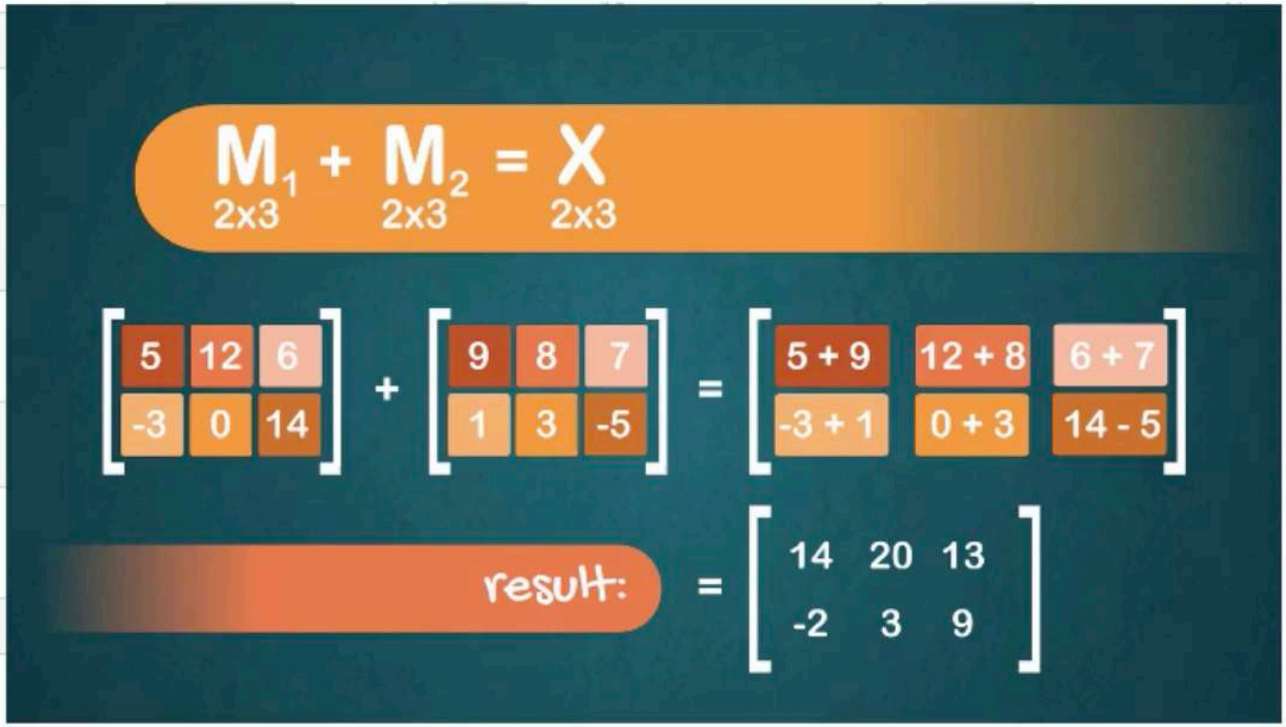
result: $= \begin{bmatrix} 14 & 20 & 13 \\ -2 & 3 & 9 \end{bmatrix}$

Import the relevant libraries

```
In [1]: import numpy as np
```

Addition

```
In [2]: m1 = np.array([[5,12,6],[-3,0,14]])
        m1

Out[2]: array([[ 5, 12,  6],
               [-3,  0, 14]])

In [3]: m2 = np.array([[9,8,7],[1,3,-5]])
        m2

Out[3]: array([[ 9,  8,  7],
               [ 1,  3, -5]])

In [4]: m1 + m2

Out[4]: array([[14, 20, 13],
               [-2,  3,  9]])
```



```
       [-340.21,    1.06],
       [  30.41,  424.99]])
```

```
In [11]: m5 - m6
```

```
Out[11]: array([[-108.8 , -453.02],
                [ 137.07, 1198.96],
                [ -26.19, -190.89]])
```

Adding vectors together

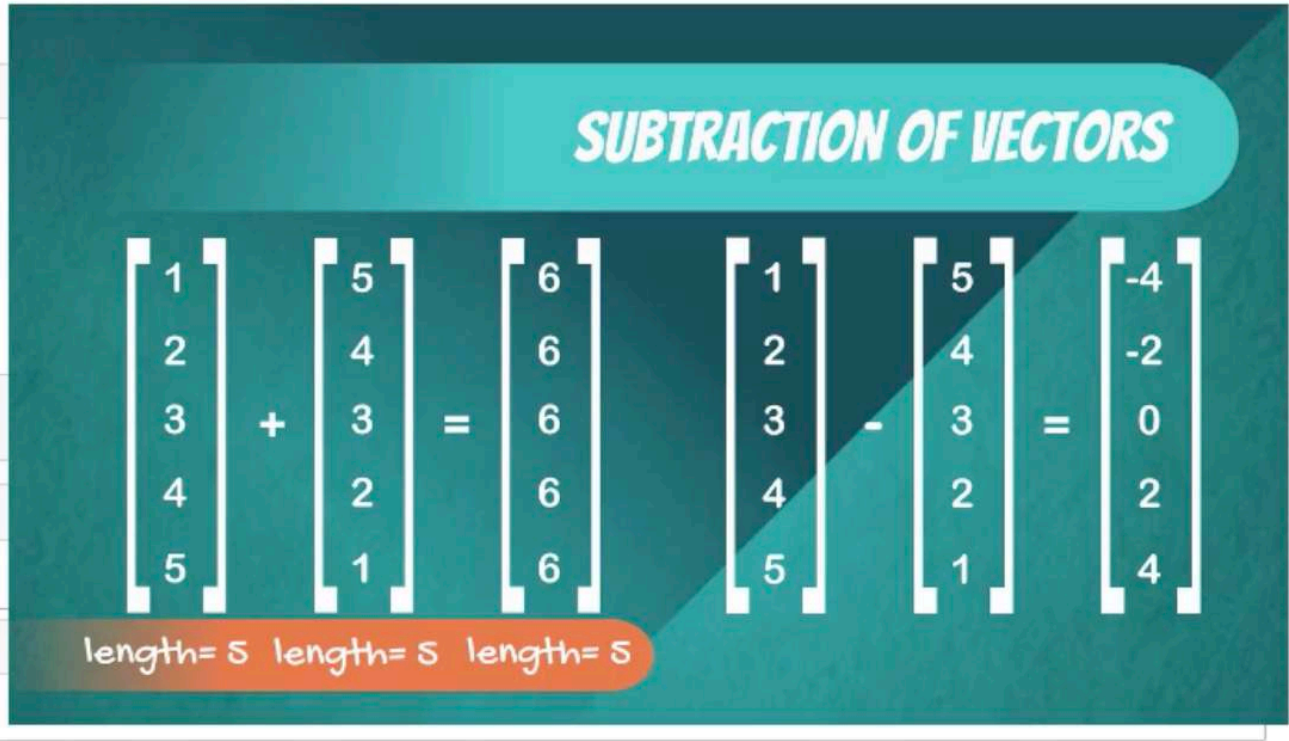
```
In [12]: v1 = np.array([1,2,3,4,5])
         v2 = np.array([5,4,3,2,1])
```

```
In [13]: v1 + v2
```

```
Out[13]: array([6, 6, 6, 6, 6])
```

```
In [14]: v1 - v2
```

```
Out[14]: array([-4, -2,  0,  2,  4])
```

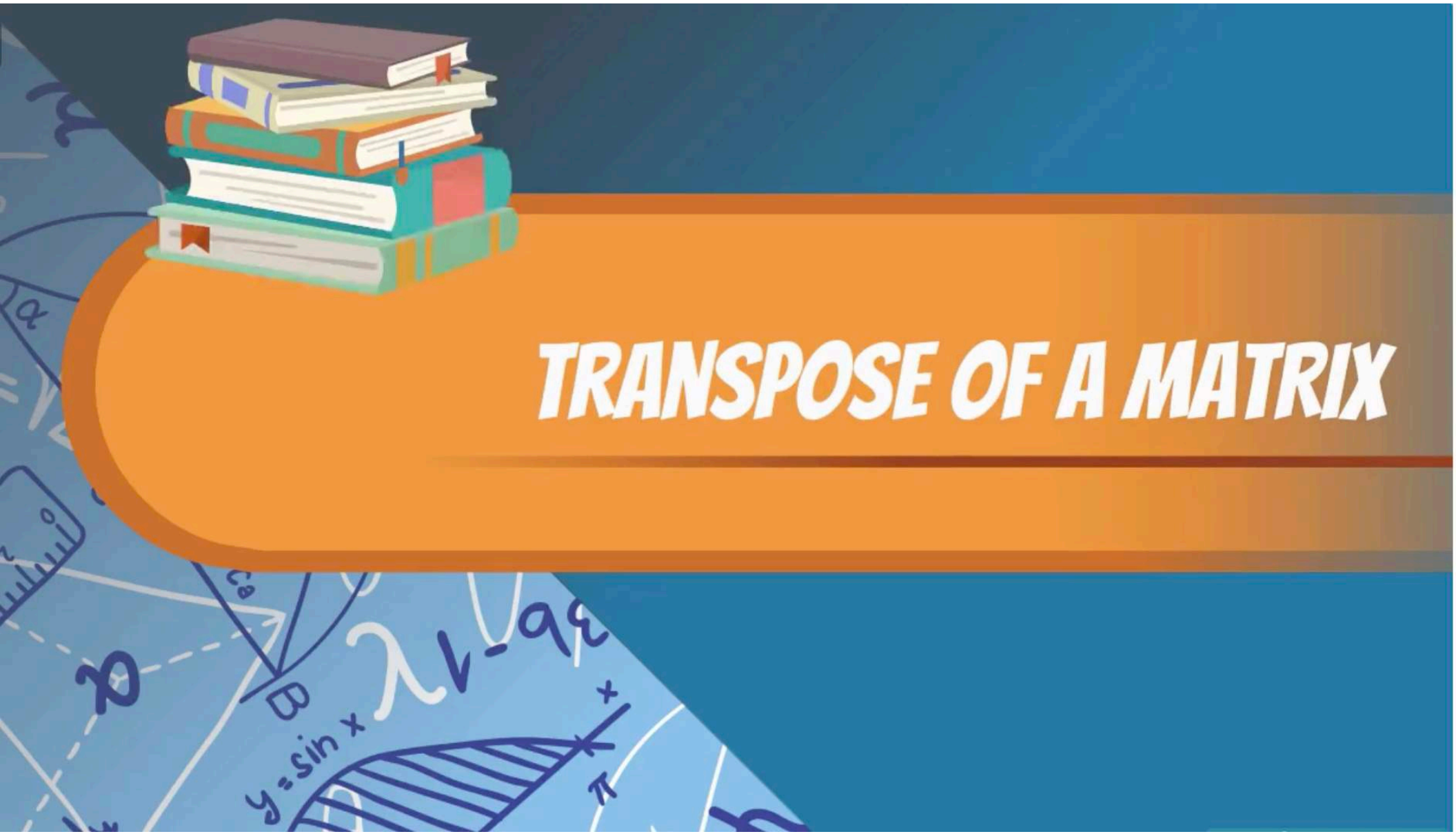


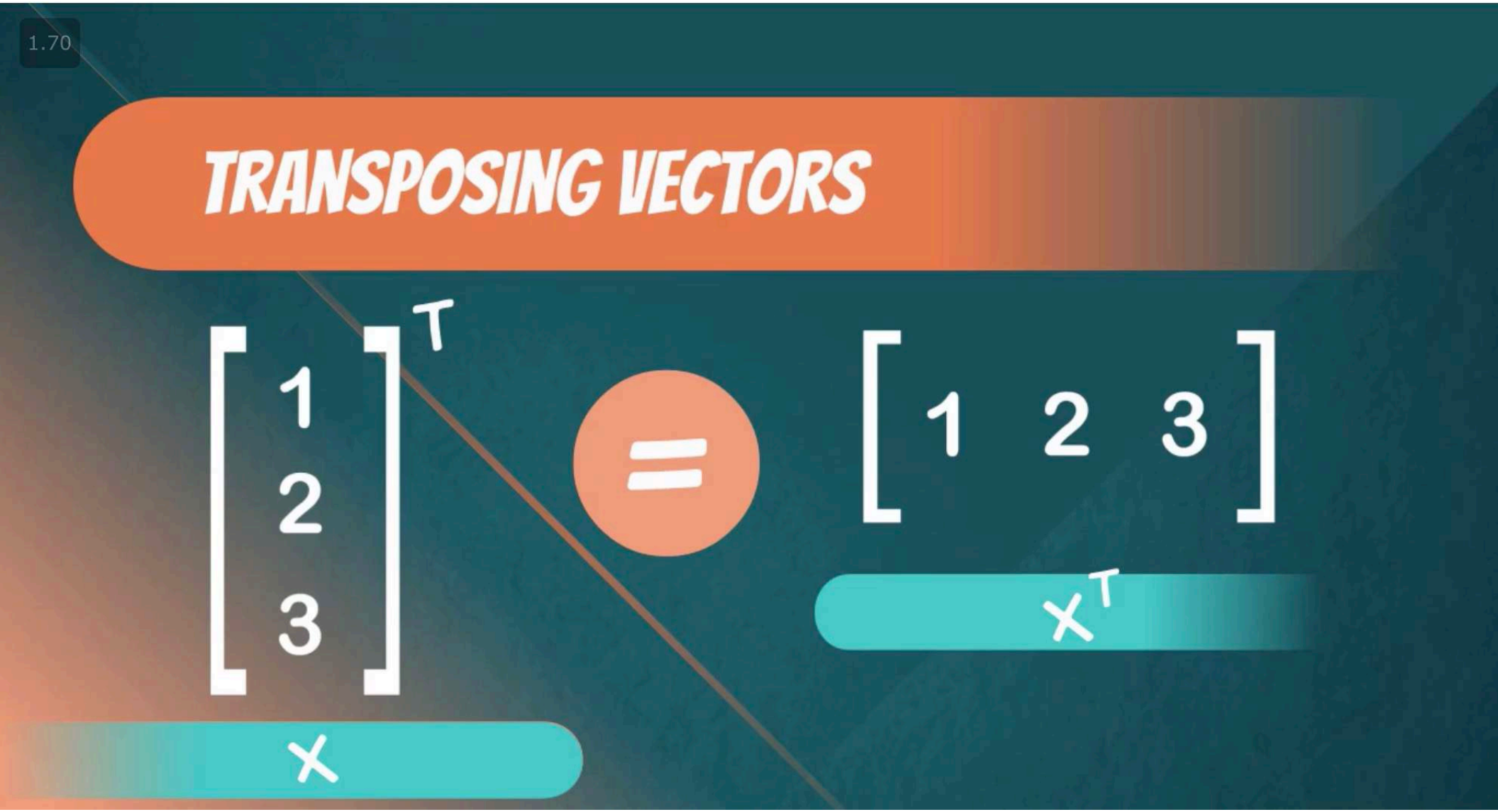
This result has a meaning in terms of arrays, but not in terms of linear algebra

Exceptions (addition with a scalar)

Adding scalars to matrices

```
In [11]: m1
Out[11]: array([[ 5, 12,  6],
                [-3,  0, 14]])

In [12]: m1 + 1
Out[12]: array([[ 6, 13,  7],
                [-2,  1, 15]])

In [13]: v1
Out[13]: array([1, 2, 3, 4, 5])

In [14]: v1 + 1
Out[14]: array([2, 3, 4, 5, 6])

In [15]: m1 + np.array([1])
Out[15]: array([[ 6, 13,  7],
                [-2,  1, 15]])

In [16]: v1 + np.array([1])
Out[16]: array([2, 3, 4, 5, 6])
```


TRANSPOSE OF A MATRIX


1.70
TRANSPOSING VECTORS
1
2
3
T
=
1 2 3
X
X T

TRANSPOSING VECTORS

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

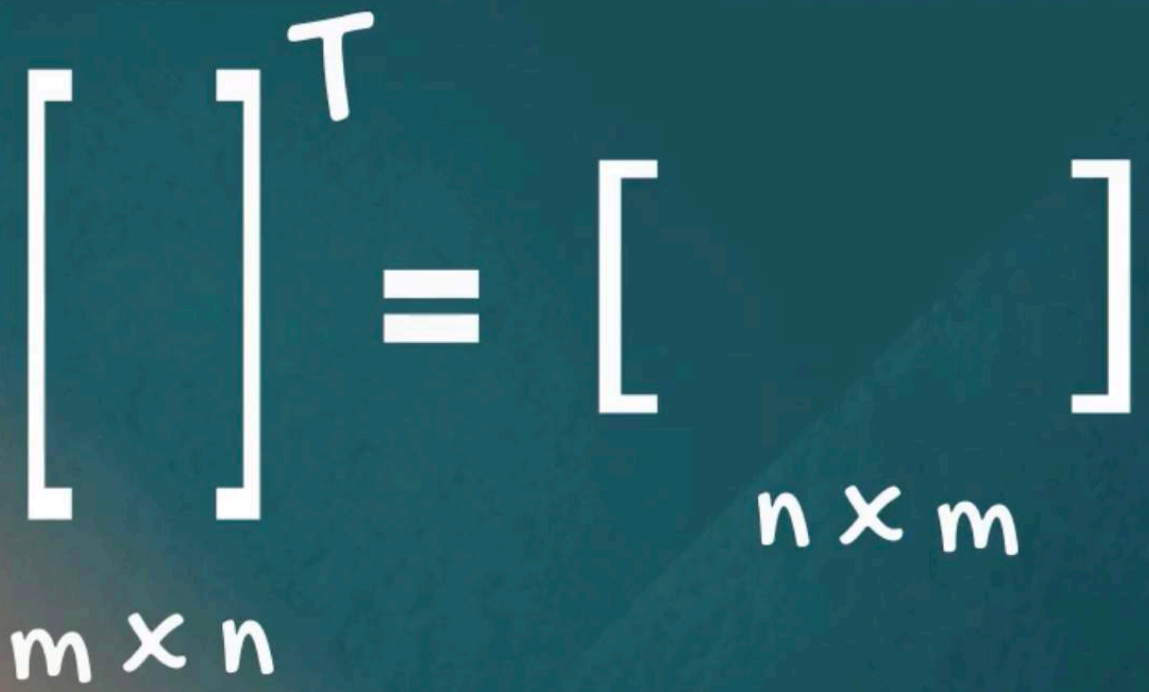
x^T

$(x^T)^T = x$

TRANSPOSING VECTORS

- The values are not changing or transforming; only their position is
- Transposing the same vector (object) twice yields the initial vector (object)
- A 3x1 matrix transposed is a 1x3 matrix

TRANSPOSING MATRICES



1.70

TRANSPOSING MATRICES

$$A_{2\times 3} = \begin{bmatrix} 5 & 12 & 6 \\ -3 & 0 & 14 \end{bmatrix}$$

$$A^{T}_{3\times 2} = \begin{bmatrix} 5 & -3 \\ 12 & 0 \\ 6 & 14 \end{bmatrix}$$

1.70

TRANSPOSING MATRICES

$$A_{2\times3} = \begin{bmatrix} 5 & 12 & 6 \\ -3 & 0 & 14 \end{bmatrix}$$

$$A^{T}_{3\times2} = \begin{bmatrix} 5 & -3 \\ 12 & 0 \\ 6 & 14 \end{bmatrix}$$

TRANSPOSING MATRICES

$$A_{2x3} = \begin{bmatrix} 5 & 12 & 6 \\ -3 & 0 & 14 \end{bmatrix}$$

$$A^T_{3x2} = \begin{bmatrix} 5 & -3 \\ 12 & 0 \\ 6 & 14 \end{bmatrix}$$

$$A_{2x3} = \begin{bmatrix} 5 & 12 & 6 \\ -3 & 0 & 14 \end{bmatrix}$$

$$A^T_{3x2} = \begin{bmatrix} 5 & -3 \\ 12 & 0 \\ 6 & 14 \end{bmatrix}$$

Import the relevant libraries

```
In [1]: import numpy as np
```

Transposing matrices

```
In [6]: C = np.array([[4,-5],[8,12],[-2,-3],[19,0]])
        C

Out[6]: array([[ 4, -5],
               [ 8, 12],
               [-2, -3],
               [19,  0]])

In [7]: C.T

Out[7]: array([[ 4,  8, -2, 19],
               [-5, 12, -3,  0]])
```

array.T returns the transpose of an array (matrix)

Transposing vectors

```
In [10]: x = np.array([1,2,3])
         x

Out[10]: array([1, 2, 3])

In [11]: x.T

Out[11]: array([1, 2, 3])

In [12]: x.shape

Out[12]: (3,)
```

In Python, 1D arrays don't really get transposed (in the memory of the computer)

```
In [13]: x_reshaped = x.reshape(1,3)
         x_reshaped

Out[13]: array([[1, 2, 3]])

In [14]: x_reshaped.T

Out[14]: array([[1],
                [2],
                [3]])
```


DOT PRODUCT

1.80

SCALAR MULTIPLICATION

$$[6] . [5] = [30]$$

$$[10] . [-2] = [-20]$$

1.80

VECTOR MULTIPLICATION

$$\begin{bmatrix} 2 \\ 8 \\ -4 \end{bmatrix} \times \begin{bmatrix} 1 \\ -7 \\ 3 \end{bmatrix}$$

Condition: They must have the same length

1.80

DOT PRODUCT

$$\begin{bmatrix} 2 \\ 8 \\ -4 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ -7 \\ 3 \end{bmatrix} = \begin{bmatrix} -66 \end{bmatrix}$$

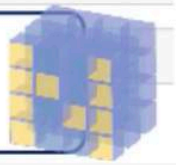
the 'sum of the products of the corresponding elements'

Vector Vector Scalar

Import relevant libraries

In [1]:
```
import numpy as np
```

np.dot() returns the dot product of two objects



Dot product

In [2]:
```
x = np.array([2,8,-4])
y = np.array([1,-7,3])
```

In [3]:
```
np.dot(x,y)
```

Out[3]: -66

In [4]:
```
u = np.array([0,2,5,8])
v = np.array([20,3,4,-1])
```

In [5]:
```
np.dot(u,v)
```

0x(20) + (2x3) + (5x4) + 8x(-1)

Out[5]: 18

In [ ]:

In [ ]:

Scalar * Scalar

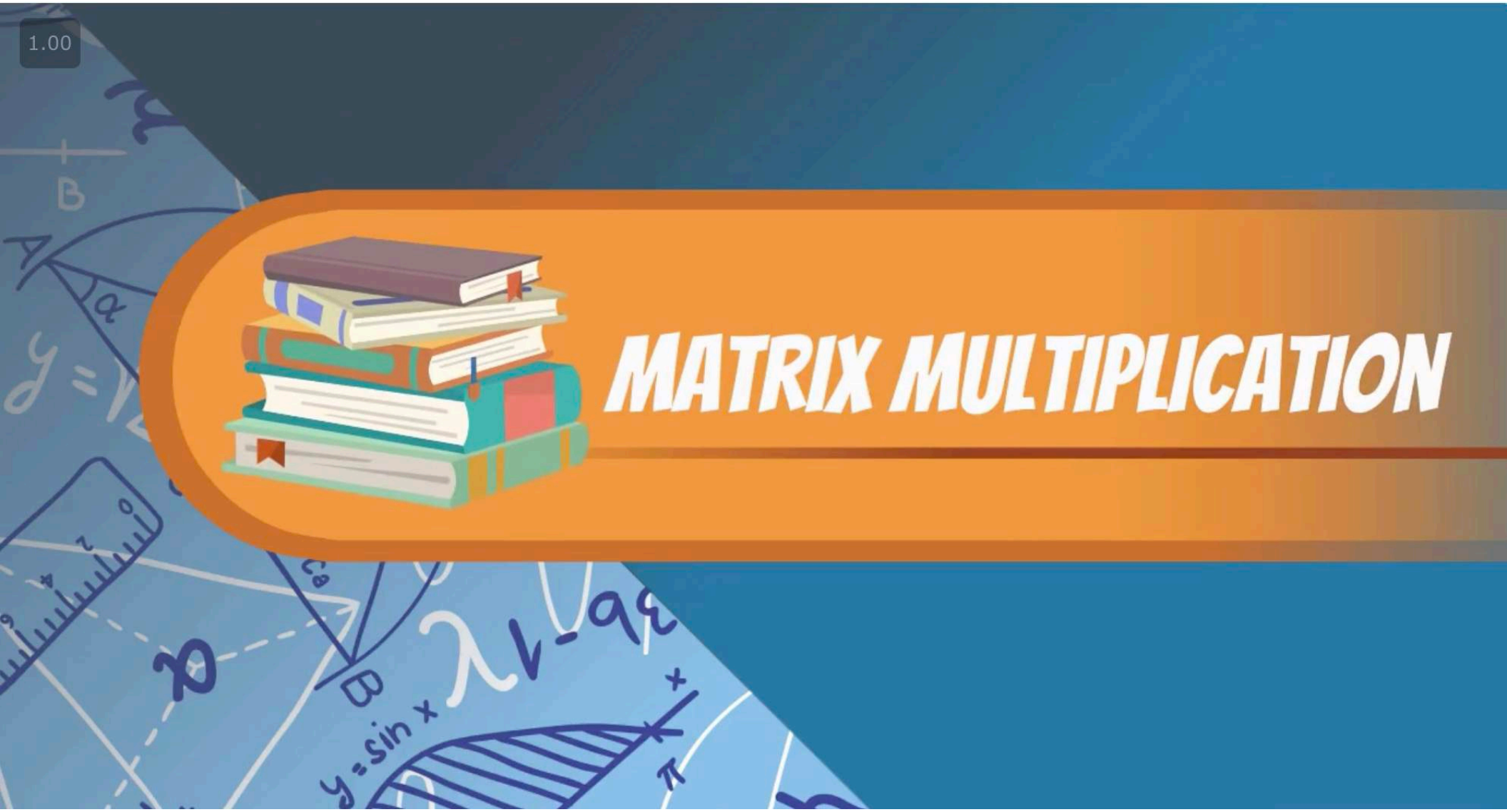
```
In [6]: np.dot(5,6)
Out[6]: 30

In [7]: np.dot(10,-2)
Out[7]: -20
```

Scalar * Vector

We get a vector with the same lenght!

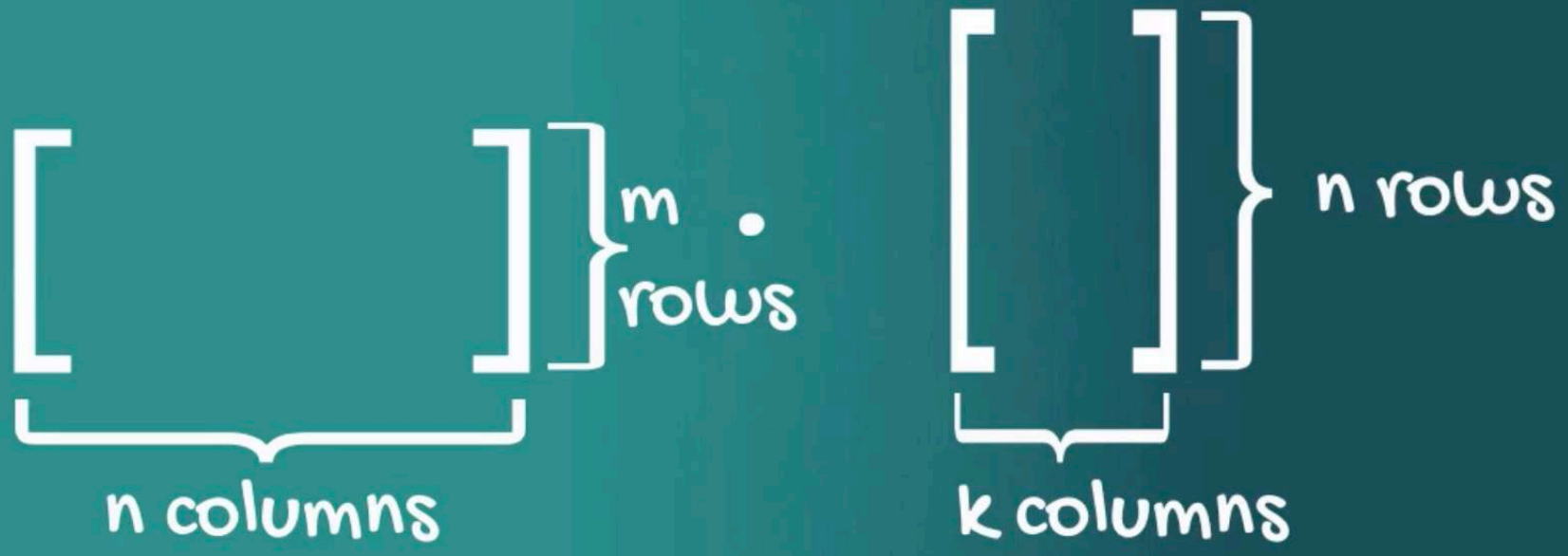
```
In [8]: x
Out[8]: array([ 2,  8, -4])

In [9]: 5*x
Out[9]: array([ 10,  40, -20])
```

The initial vector had been SCALED 5 times (thus the name)


1.00
MATRIX MULTIPLICATION

1.90

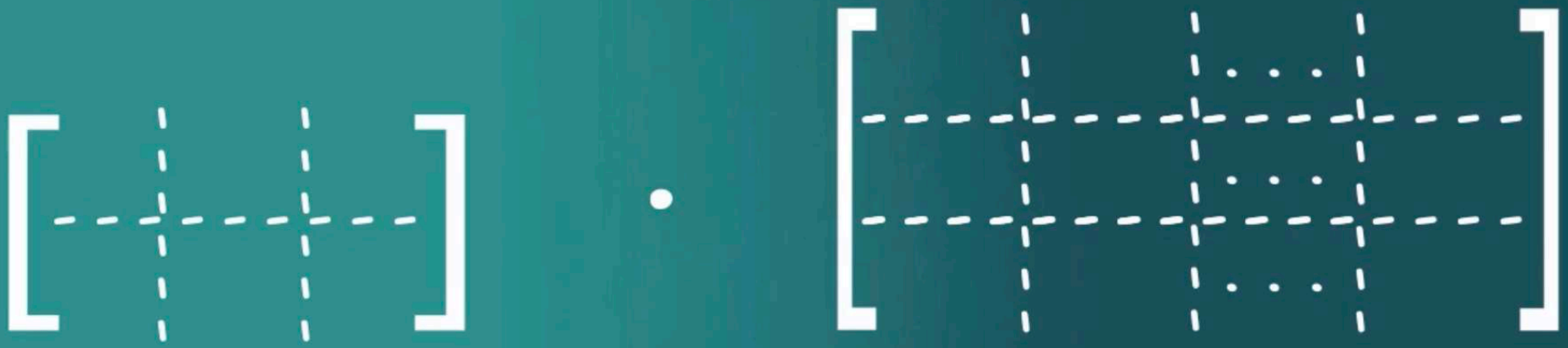
DOT PRODUCT



Condition: We can only multiply an m x n with an n x k matrix

1.90

DOT PRODUCT



2 X 3

3 X k

Condition: We can only multiply an m x n with an n x k matrix

1.90

DOT PRODUCT

$A_{2x3} \cdot B_{3x6} = C_{2x6}$

$A_{3x4} \cdot B_{4x2} = C_{3x2}$

$A_{100x300} \cdot B_{300x3} = C_{100x3}$

1.90

DOT PRODUCT

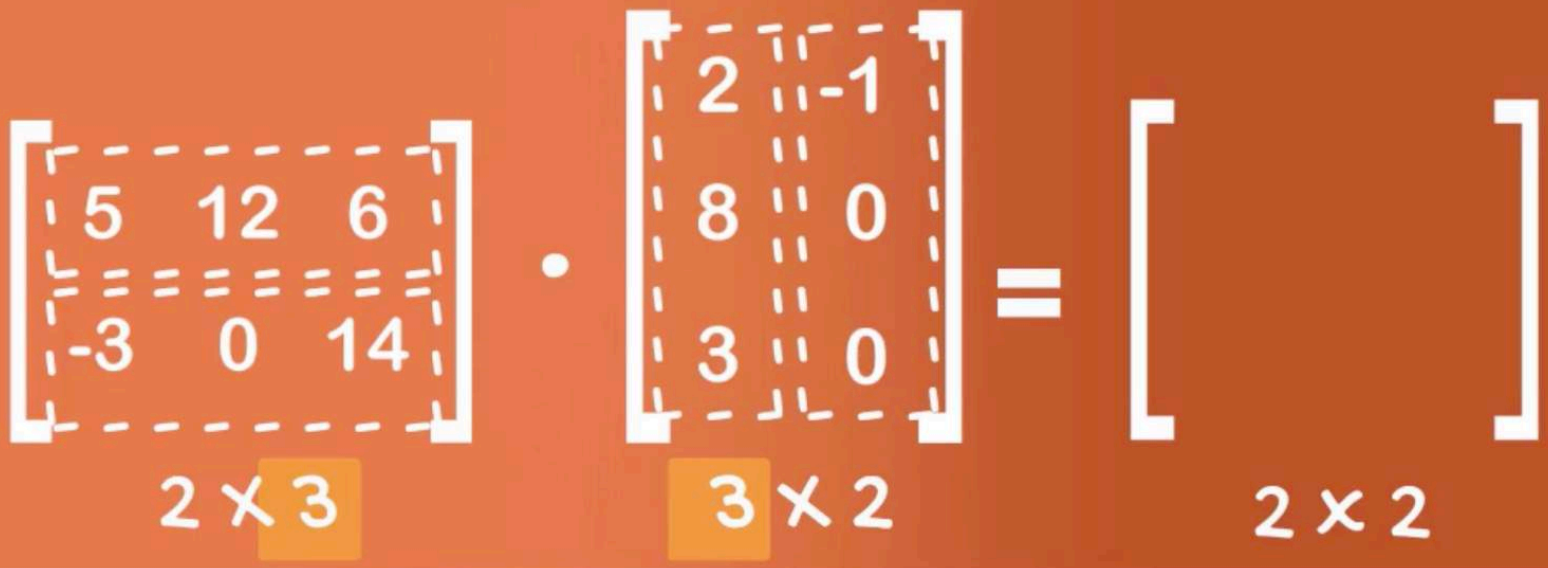
$$\underset{1 \times 3}{\begin{bmatrix} 2 & 8 & -4 \end{bmatrix}} \cdot \underset{3 \times 1}{\begin{bmatrix} 1 \\ -7 \\ 3 \end{bmatrix}} = \begin{bmatrix} -66 \end{bmatrix}$$

Scalar

Compatibility: they need to have matching forms

1.90

DOT PRODUCT



1) Matrices are nothing more than a collection of vectors

2) When we have a dot product, we always multiply a row vector times a column vector

DOT PRODUCT

$$\begin{bmatrix} 5 & 12 & 6 \\ -3 & 0 & 14 \end{bmatrix} \cdot \begin{bmatrix} 2 & -1 \\ 8 & 0 \\ 3 & 0 \end{bmatrix} = \begin{bmatrix} 124 & \\ & \end{bmatrix}$$

2×3 $\quad$ 3×2 $\quad$ 2×2

$$5 \times 2 + 12 \times 8 + 6 \times 3$$

DOT PRODUCT

$$\begin{bmatrix} 5 & 12 & 6 \\ -3 & 0 & 14 \end{bmatrix} \cdot \begin{bmatrix} 2 & -1 \\ 8 & 0 \\ 3 & 0 \end{bmatrix} = \begin{bmatrix} 124 & -5 \\ & \end{bmatrix}$$

$2 \times 3 \qquad 3 \times 2 \qquad 2 \times 2$

$$(5 \times -1) + (12 \times 0) + (6 \times 0)$$

DOT PRODUCT

$$\begin{bmatrix} 5 & 12 & 6 \\ -3 & 0 & 14 \end{bmatrix} \cdot \begin{bmatrix} 2 & -1 \\ 8 & 0 \\ 3 & 0 \end{bmatrix} = \begin{bmatrix} 124 & -5 \\ 36 & \end{bmatrix}$$

2×3 $\quad$ 3×2 $\quad$ 2×2

$$(-3 \times 2) + (0 \times 8) + (14 \times 3)$$

1.90

DOT PRODUCT

$$\begin{bmatrix} 5 & 12 & 6 \\ -3 & 0 & 14 \end{bmatrix} \cdot \begin{bmatrix} 2 & -1 \\ 8 & 0 \\ 3 & 0 \end{bmatrix} = \begin{bmatrix} 124 & -5 \\ 36 & 3 \end{bmatrix}$$

2×3 $\quad$ 3×2 $\quad$ 2×2

Row vectors determine the row, column vectors -> the column

Compatibility: they need to have matching forms

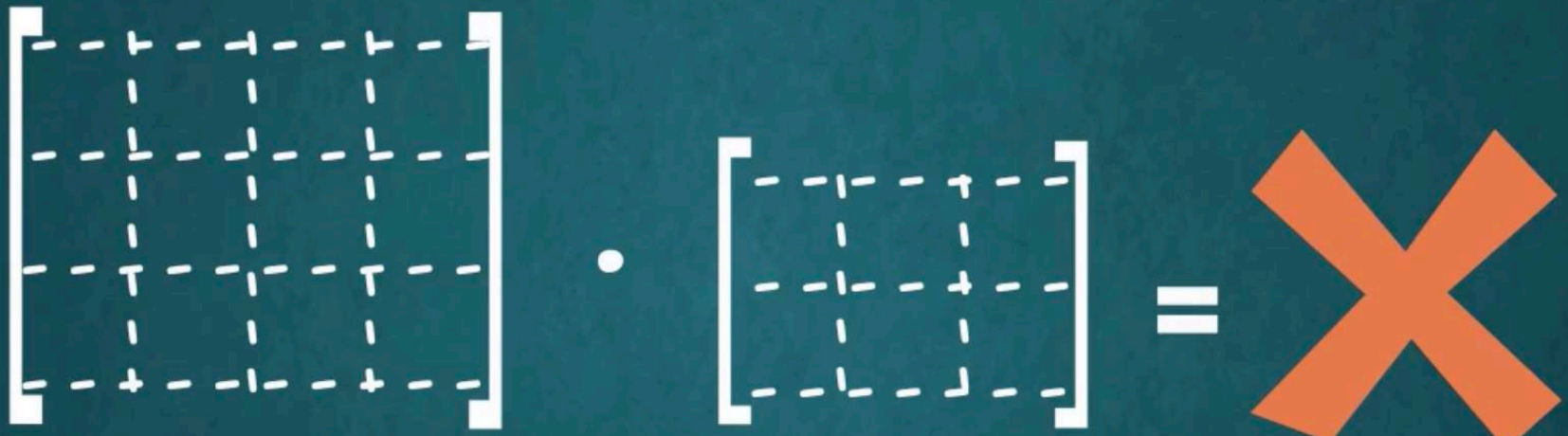
DOT PRODUCT

$$\begin{bmatrix} -12 & 5 & -5 & 1 & 6 \\ 6 & -2 & 0 & 0 & -3 \\ 10 & 2 & 0 & 8 & 0 \\ 9 & -4 & 8 & 3 & -6 \end{bmatrix} \cdot \begin{bmatrix} 6 & -1 \\ 8 & -4 \\ 2 & -2 \\ 7 & 4 \\ -6 & -9 \end{bmatrix} = \begin{bmatrix} & \\ & \\ & \\ & \end{bmatrix}$$

4×5 $\quad$ 5×2 $\quad$ 4×2

1.90

DOT PRODUCT



NOT COMPATIBLE

Scalar * Matrix

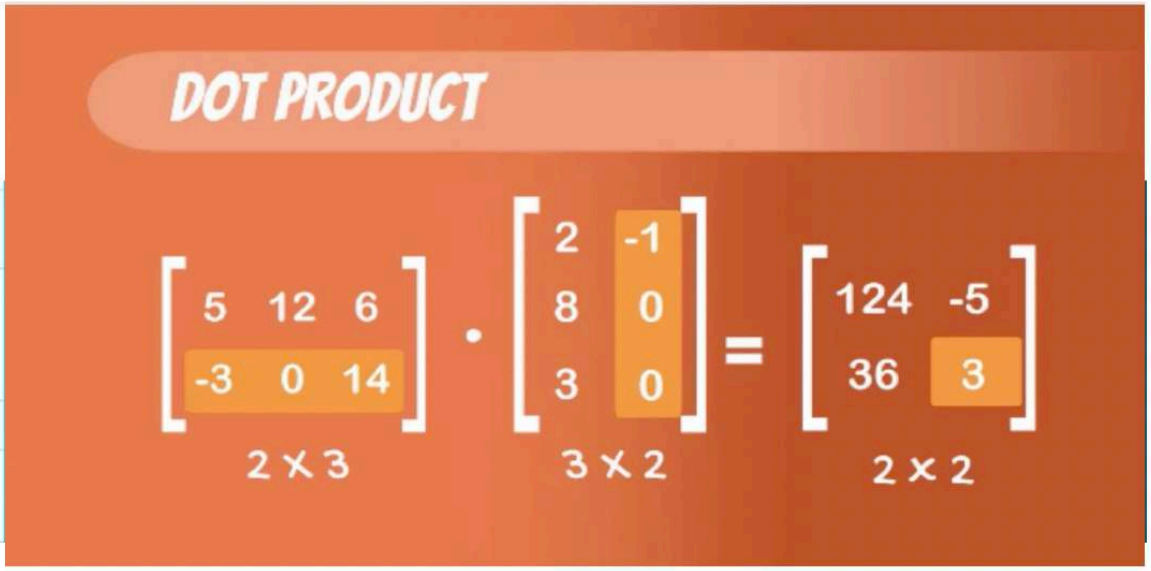
```
In [9]: A = np.array([[5,12,6],[-3,0,14]])
        A

Out[9]: array([[ 5, 12,  6],
               [-3,  0, 14]])

In [ ]: 3*A

Out[10]: array([[15, 36, 18],
                [-9,  0, 42]])
```

The initial matrix had been SCALED 3 times

Matrix * Matrix

Example 1

```
In [11]: B = np.array([[2,-1],[8,0],[3,0]])
         B

Out[11]: array([[ 2, -1],
                [ 8,  0],
                [ 3,  0]])

In [12]: np.dot(A,B)

Out[12]: array([[124,  -5],
                [ 36,   3]])
```



WHY IS LINEAR ALGEBRA ACTUALLY USEFUL?

Applications in data science

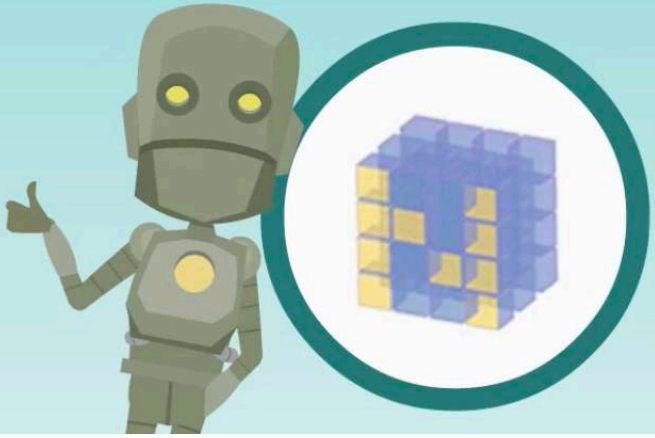
- Vectorized code
- Image recognition
- Dimensionality reduction

1.90

VECTORIZED CODE

Whenever we are using linear algebra
to compute many values simultaneously

much faster



NUMPY

VECTORIZED CODE

Whenever we are using linear algebra
to compute many values simultaneously

1.90

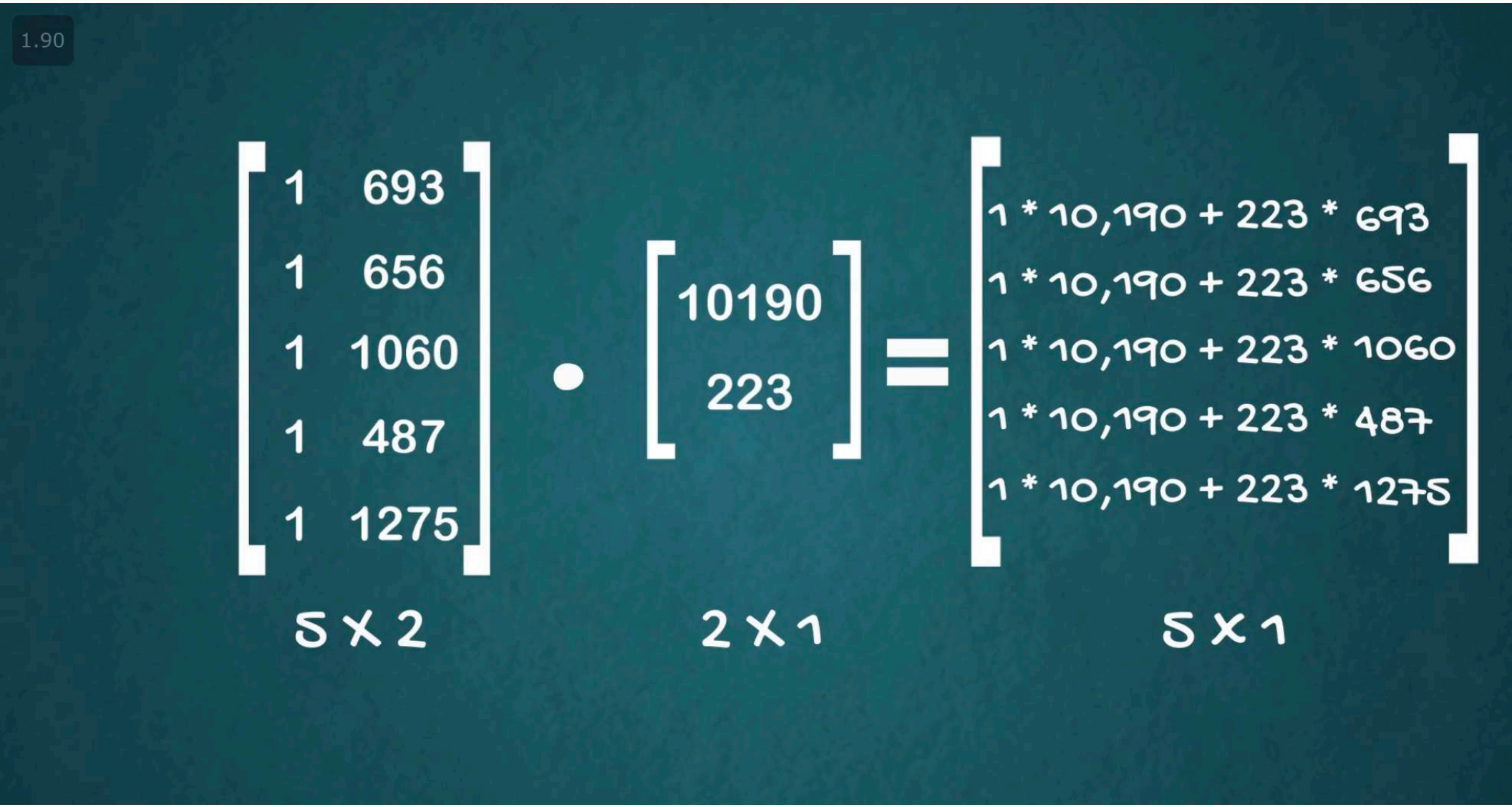
10,190 + 223 *  = 

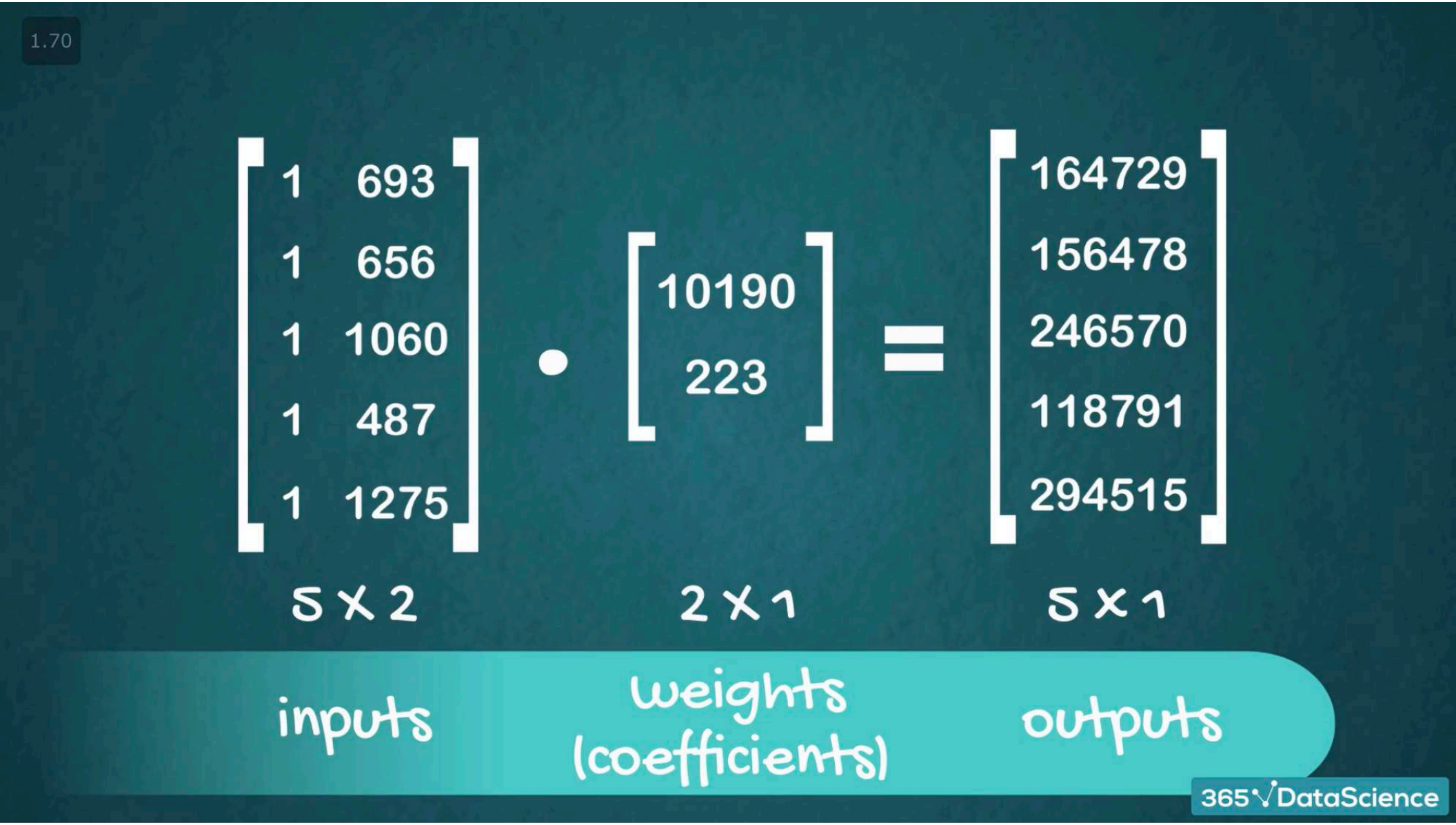
MANUALLY

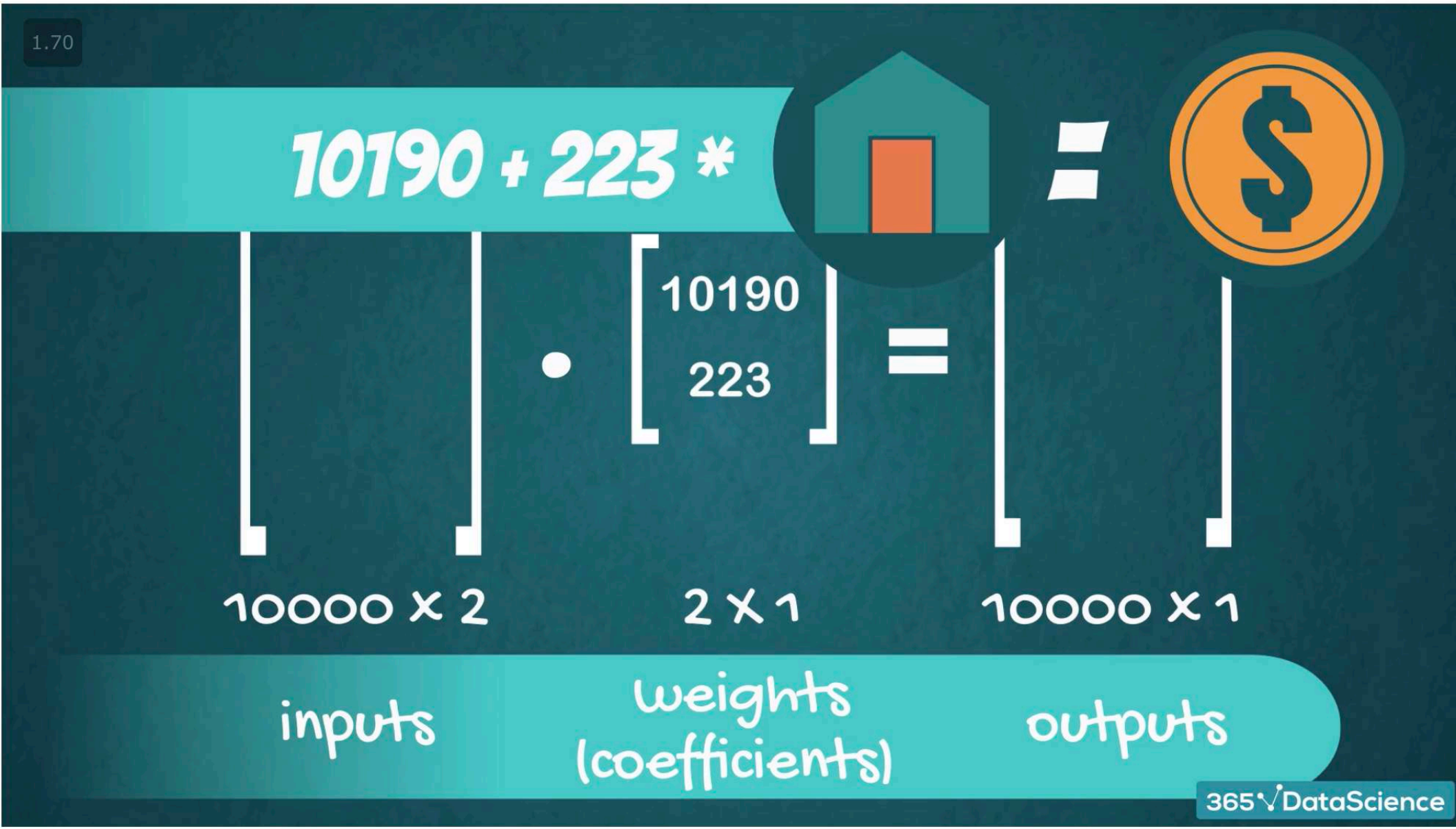
10,190 + 223 * 693 = 164729
10,190 + 223 * 656 = 156478
10,190 + 223 * 1060 = 246570
10,190 + 223 * 487 = 118791
10,190 + 223 * 1275 = 294515

WITH A LOOP

```
for i in range (100):
    print (10190 + 223 * size [ i ])
```


1.90
1 693
1 656
1 1060
1 487
1 1275
5 X 2
10190
223
2 X 1
1 * 10,190 + 223 * 693
1 * 10,190 + 223 * 656
1 * 10,190 + 223 * 1060
1 * 10,190 + 223 * 487
1 * 10,190 + 223 * 1275
5 X 1

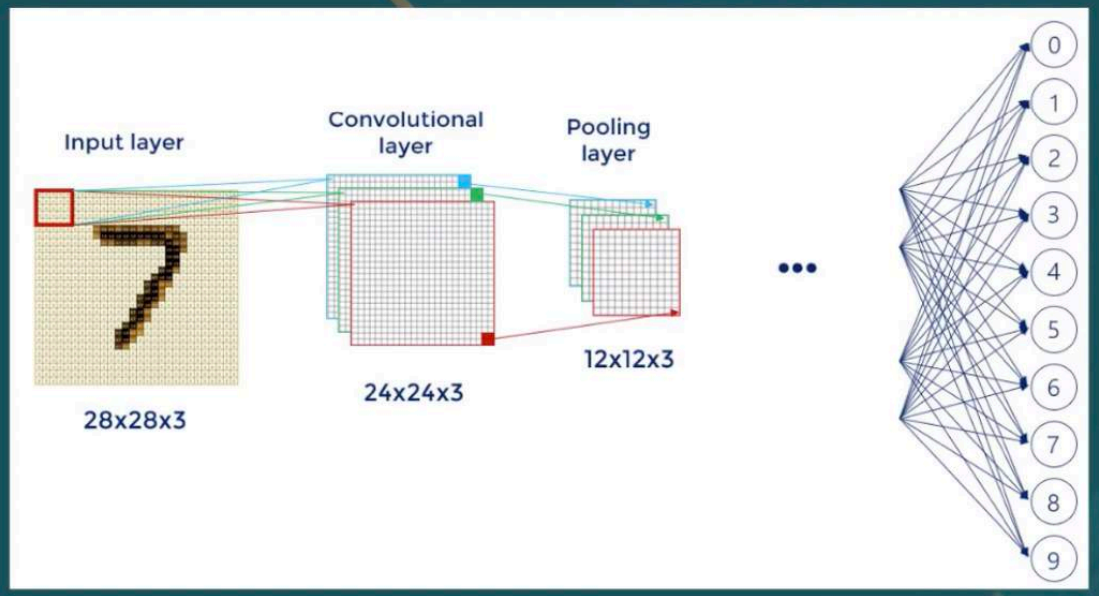


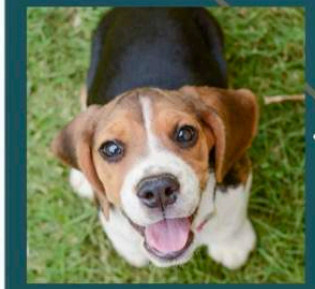

10190 + 223 *
=
10190
223
10000 X 2
2 X 1
10000 X 1
inputs
weights
(coefficients)
outputs
365 DataScience

1.90

IMAGE RECOGNITION

Deep learning
CNNS (convolutional neural networks)





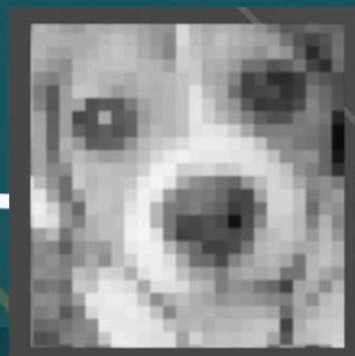
400 x 400

DOG

1.90

IMAGE RECOGNITION

- Zoom in until we see pixels
- Zoom out to reach the photo



400 x 400

Height = 400

400 x 400

Wight = 400

- 256 shades of gray
- 0 = totally white
  255 = totally black
- shows the intensity of grey

1.90

IMAGE RECOGNITION

how to represent colour

400 x 400

R	G	B		RGB scale
255	0	0	›	
0	255	0	›	
0	0	255	›	
138	106	214	›	

1:90

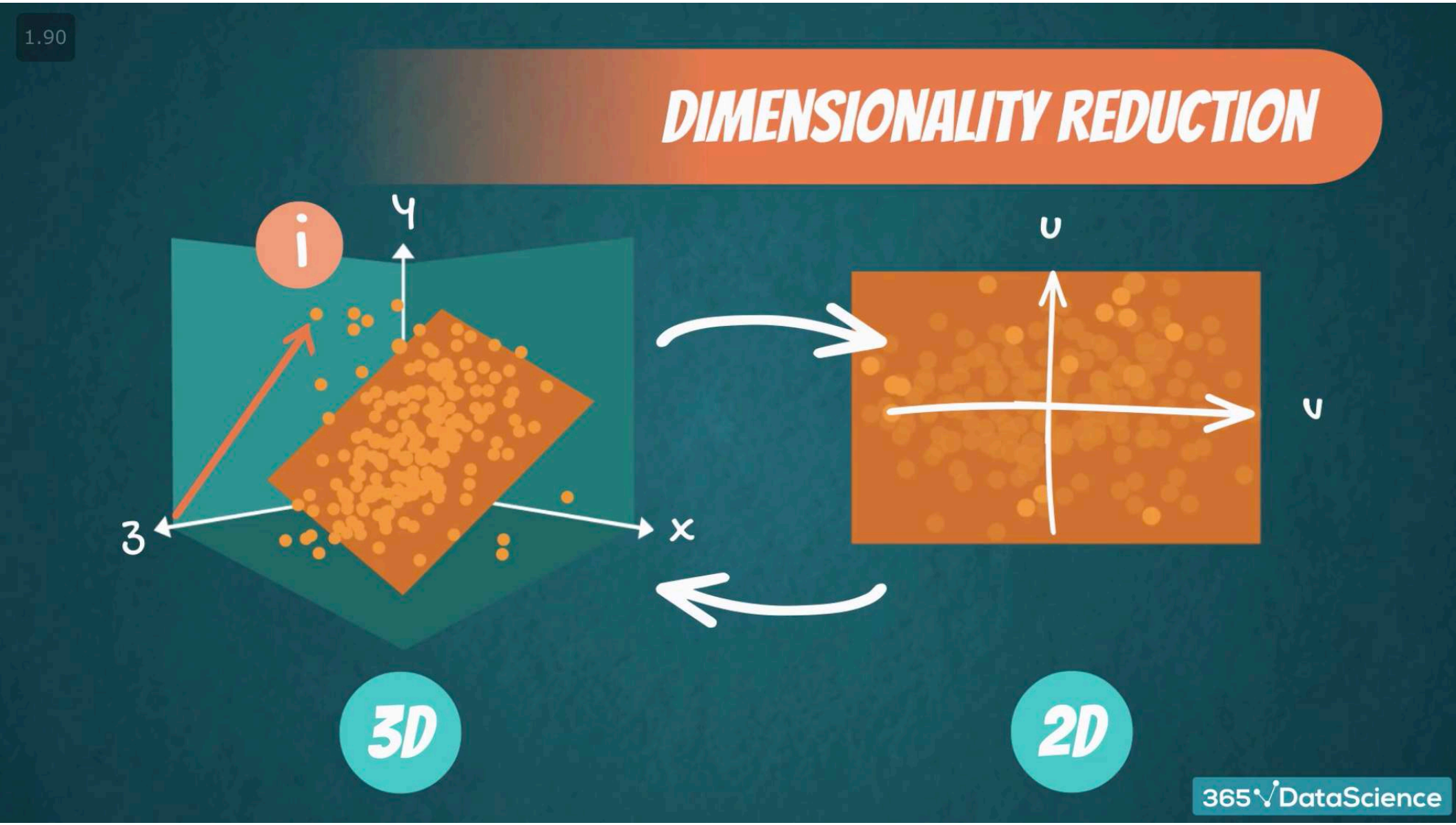
IMAGE RECOGNITION

How to represent colour

- RGB scale
- Tensors

3 x 400 x 400

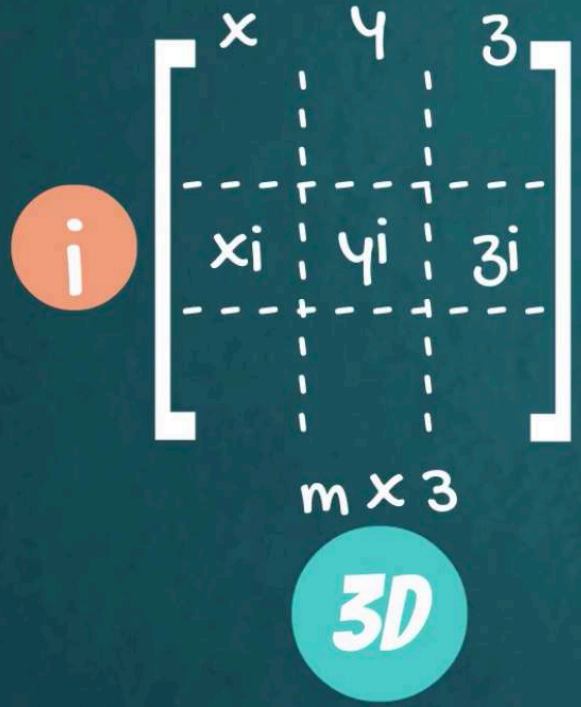
1.90
DIMENSIONALITY REDUCTION
(xi, yi, 3i)
i
3
x
i
x
y
3
xi
yi
3i
m x 3


1.90
DIMENSIONALITY REDUCTION
i
y
z
x
u
v
3D
2D
365 DataScience

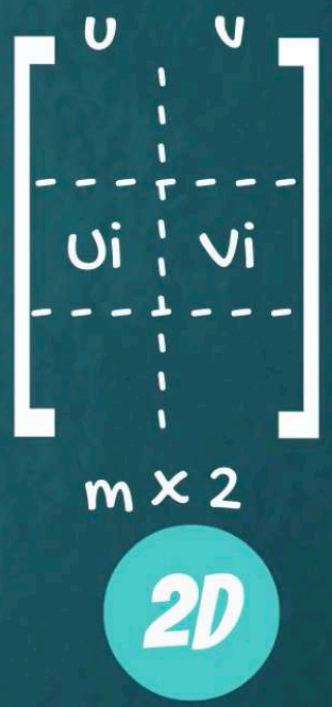
1.90

DIMENSIONALITY REDUCTION

INITIAL



TRANSFORMED



DIMENSIONALITY REDUCTION

PLEASE RATE FROM 1 TO 5

1)I feel comfortable around people

1 2 3 4 5

2)I easily make friends

1 2 3 4 5

3)I like going out

1 2 3 4 5

⋮

SO)

TROUGH LINEAR ALGEBRA

LEVEL OF EXTROVERSION