Berkeley MSSE
MOLECULAR SCIENCE & SOFTWARE ENGINEERING

Lecture 07:

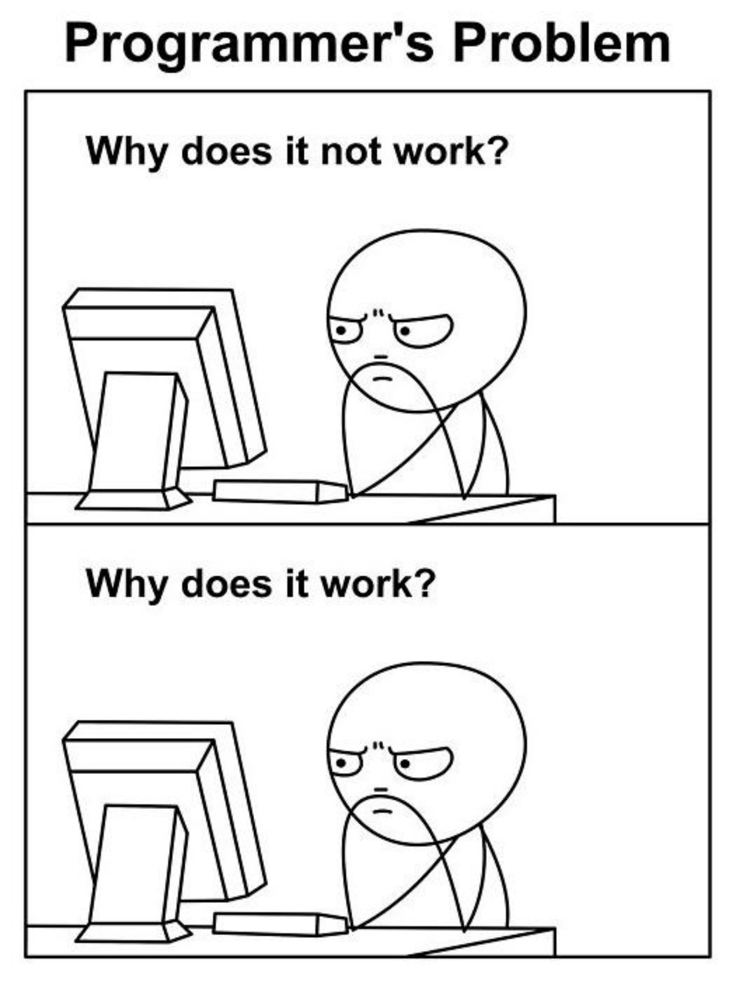
Scientific Computing

Markus Hohle

University California, Berkeley

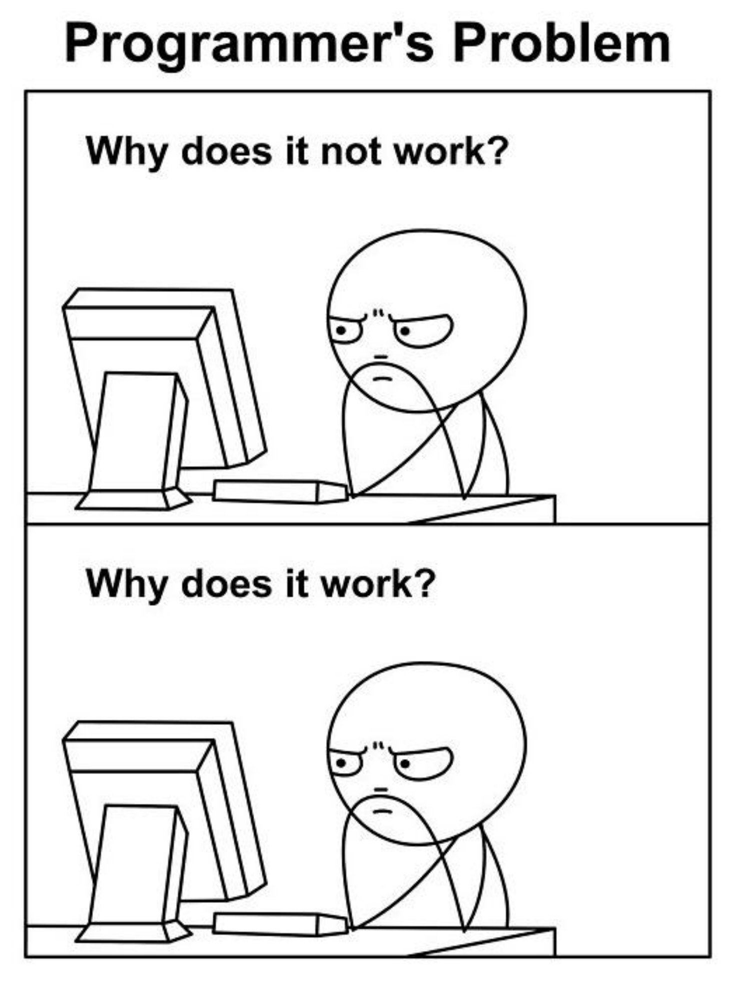
Python for Molecular Sciences

MSSE 272, 3 Units



Outline

- introduction to numpy
- linear algebra with numpy
- avoiding loops
- random numbers



Outline

- introduction to numpy

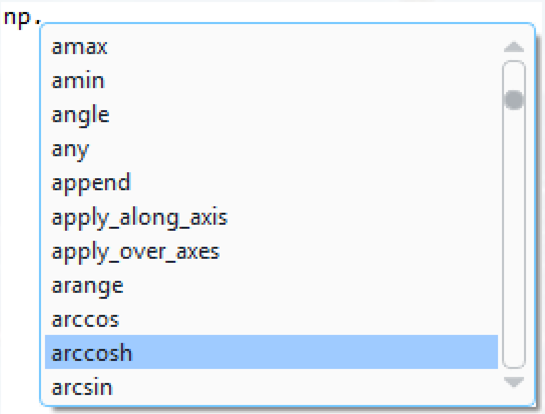
- linear algebra with numpy

- avoiding loops

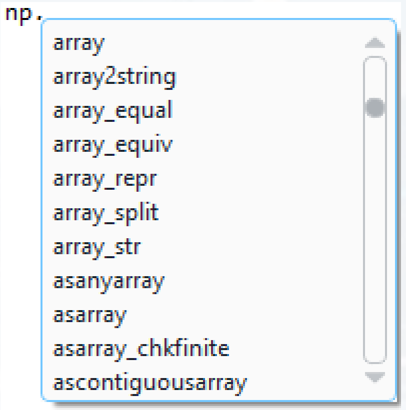
- random numbers

so far, we have used `math`
faster and more functions: `numpy`

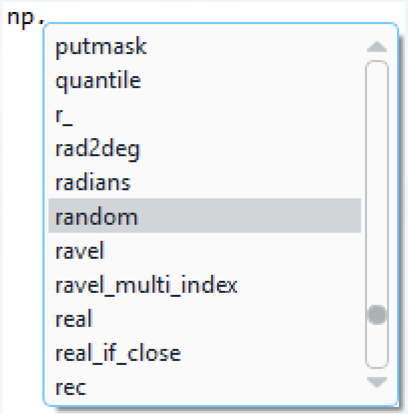
```python
import numpy as np
```



basic math function (like `math`)



manipulating/creating arrays



random number generator

…and much, much more…

type: numpy array

```
M = np.array([[1,2,5,6], [0,8,5,-4], [-4,4,6,1], [2,3,-1,0]])

print(M)
```

```
array([[ 1,  2,  5,  6],
       [ 0,  8,  5, -4],
       [-4,  4,  6,  1],
       [ 2,  3, -1,  0]])
```

```
type(M)
```

```
type(M)
numpy.ndarray
```

useful numpy commands:

```
V = np.arange(start = -1, stop = 3, step = 0.01)
```

creating an **array**, that contain floats which start at `start`, continue in steps of `step` until the stop value `stop`

V - NumPy object array

	0
0	-1
1	-0.99
2	-0.98
3	-0.97
4	-0.96
5	-0.95

useful numpy commands:

```
Z = np.zeros((5,6,7))
```

creating an **array** of any shape that only contains zeros.
Ideal for pre-allocating matrices

Z - NumPy object array

	0	1	2
0	0	0	0
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0

see also

```
Z = np.ones((5,6,7))
```

useful numpy commands:

```
I = np.eye(5)
```

creating the N x N **identity matrix**

I - NumPy object array

	0	1	2	3	4
0	1	0	0	0	0
1	0	1	0	0	0
2	0	0	1	0	0
3	0	0	0	1	0
4	0	0	0	0	1

useful numpy commands:

```
T = np.tile([1, 2, 4, 5], reps = (5, 1))
```

creating multiple replicates of an array

T - NumPy object array

	0	1	2	3
0	1	2	4	5
1	1	2	4	5
2	1	2	4	5
3	1	2	4	5
4	1	2	4	5

useful numpy commands:

```
T = np.tile([1, 2, 4, 5], reps = (1, 5))
```

creating multiple replicates of an array

T - NumPy object array

	0	1	2	3	4	5
0	1	2	4	5	1	2

math vs numpy

Task: Calculating cosine from an array!

```
A = np.arange(-1, 3, 1/1000000)
```

```
from math import cos as math_cos
from numpy import cos as np_cos
import time


t1 = time.monotonic()

for a in A:
    math_cos(a)


t2 = time.monotonic()

dt = t2 - t1
print('Total runtime: ' + str(dt) + ' seconds')
```

assigning different names in order to be able to distinguish between the different methods

0.36 sec

math vs numpy

Task: Calculating cosine from an array!

```
A = np.arange(-1, 3, 1/1000000)
```

```
from math import  cos as math_cos
from numpy import cos as np_cos
import time


t1 = time.monotonic()

for a in A:
    np_cos(a)


t2 = time.monotonic()

dt = t2 - t1
print('Total runtime: ' + str(dt) + ' seconds')
```

2.26 sec

That is 6 times slower!

math vs numpy

Task: Calculating cosine from an array!

```
A = np.arange(-1, 3, 1/1000000)
```

```
from math import  cos as math_cos
from numpy import cos as np_cos
import time


t1 = time.monotonic()

np_cos(A)

t2 = time.monotonic()

dt = t2 - t1
print('Total runtime: ' + str(dt) + ' seconds')
```

We actually don't need the loop at all!

0.016 sec

math vs numpy

Task: Calculating cosine from an array!

```
A = np.arange(-1, 3, 1/1000000)
```

```
from math import  cos as math_cos
from numpy import cos as np_cos
import time


t1 = time.monotonic()

math_cos(A)


t2 = time.monotonic()

dt = t2 - t1
print('Total runtime: ' + str(dt) + ' seconds')
```

not applicable: math cos is not vectorized!

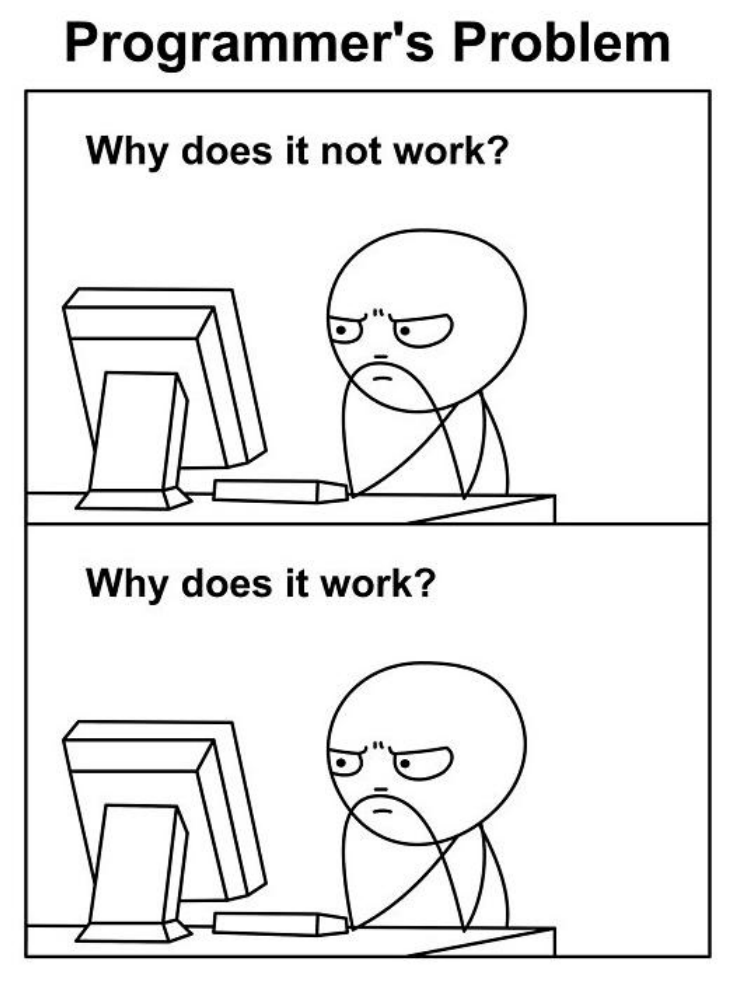
```
Traceback (most recent call last):

  Cell In[40], line 2
    math_cos(A)

TypeError: only length-1 arrays can be converted to Python scalars
```

math vs numpy

	math	numpy
loop	**0.36 sec**	**2.26 sec**
vector	**n. a.**	**0.016 sec**

math is optimized for scalars. It doesn't need arrays

arrays are actually slow but faster than a for loop → use numpy



Outline

- introduction to numpy
- **linear algebra with numpy**
- avoiding loops
- random numbers

Why do we need matrices:

- “vectorized” code is : faster
  shorter
  maintainable
  readable
  scalable

- AI/ANN: essentially matrix operations

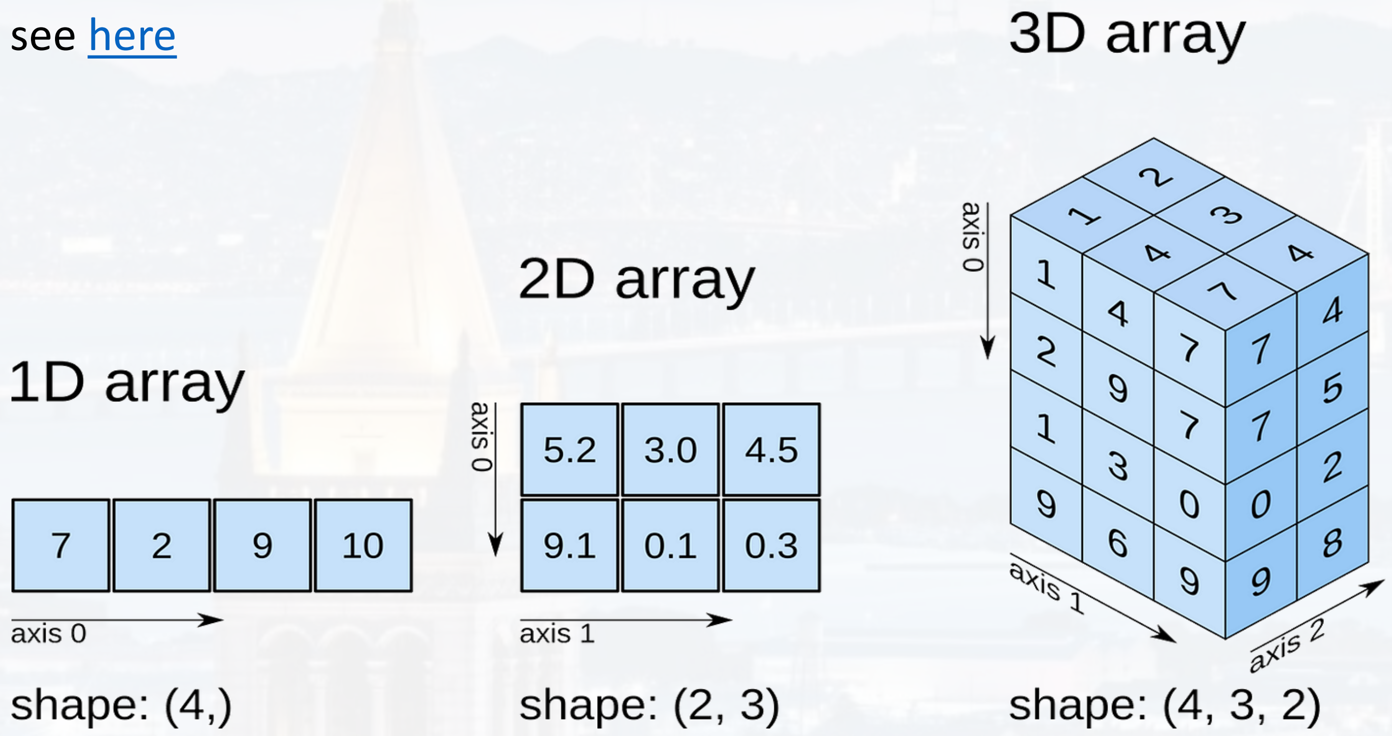
- regression, linear models etc

- easy to parallelize

in python:

```
[[ 3 -1  2  2]
 [15 -5 10 10]
 [ 0  0  0  0]
 [-9  3 -6 -6]]
```

see [here]()



… and higher

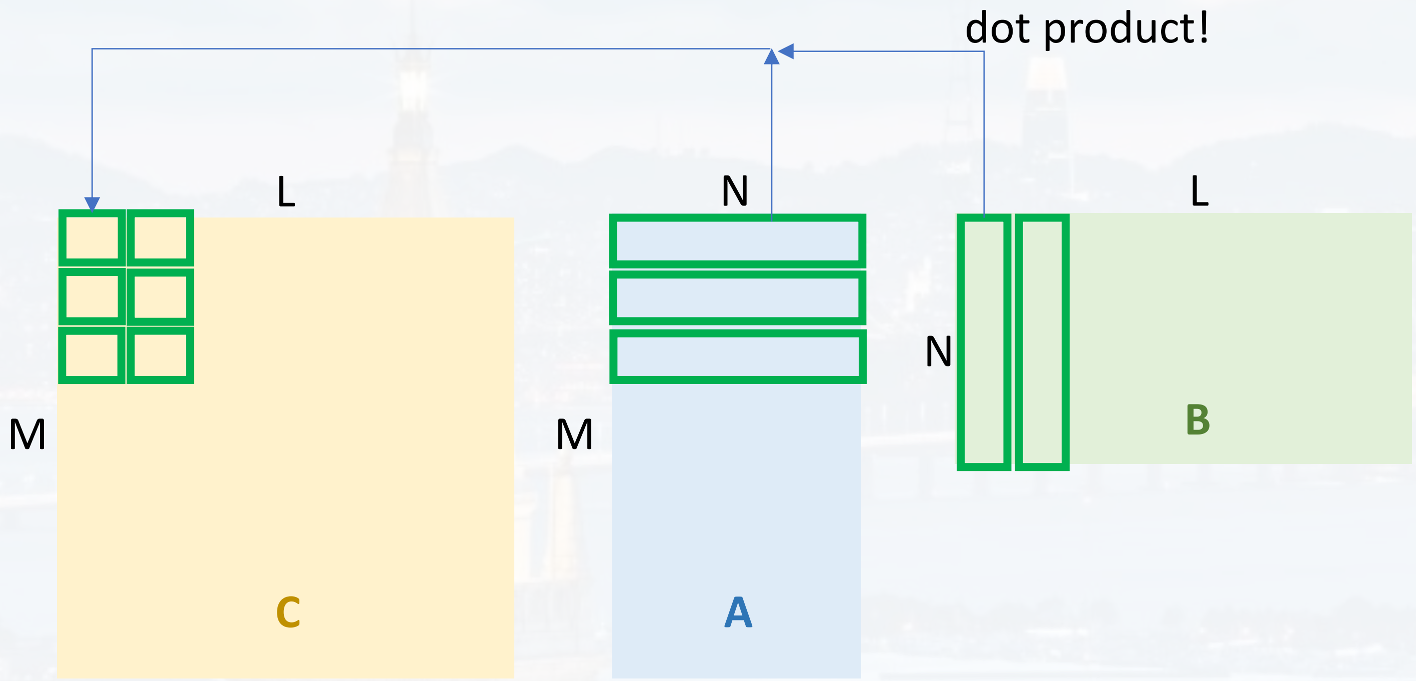
<u>note:</u>

higher dimensional arrays are called **tensor** in the CS community, but they are not the same tensors as in math & physics

the math:

actual matrix multiplication

C = A*B



$$c_{m,l} = \sum_{n=0}^{N} a_{m,n} \; b_{n,l}$$

- only works if $n_{column}(A) = n_{row}(B)$
- result: $C(n_{row}(A), n_{column}(B))$

```
v1 = np.array([1,5,0,-3])
v2 = np.array([3,-1,2,2])
```

```
1) np.dot(v1,v2)
2) np.outer(v1,v2)
```

$$\begin{pmatrix} a & b & c \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = a\alpha + b\beta + c\gamma$$

$$\begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} \begin{pmatrix} a & b & c \end{pmatrix} = \begin{pmatrix} a\alpha & \alpha b & \alpha c \\ a\beta & \beta b & \beta c \\ a\gamma & \gamma b & \gamma c \end{pmatrix}$$

```
In [75]: print(np.dot(v1,v2))
-8

In [76]: print(np.dot(v2,v1))
-8
```

```
In [78]: print(np.outer(v1,v2))
[[ 3 -1  2  2]
 [15 -5 10 10]
 [ 0  0  0  0]
 [-9  3 -6 -6]]

In [79]: print(np.outer(v2,v1))
[[ 3 15  0 -9]
 [-1 -5  0  3]
 [ 2 10  0 -6]
 [ 2 10  0 -6]]
```

```
M = np.array([[1,2,5,6], [0,8,5,-4], [-4,4,6,1],\
              [2,3,-1,0]])
```

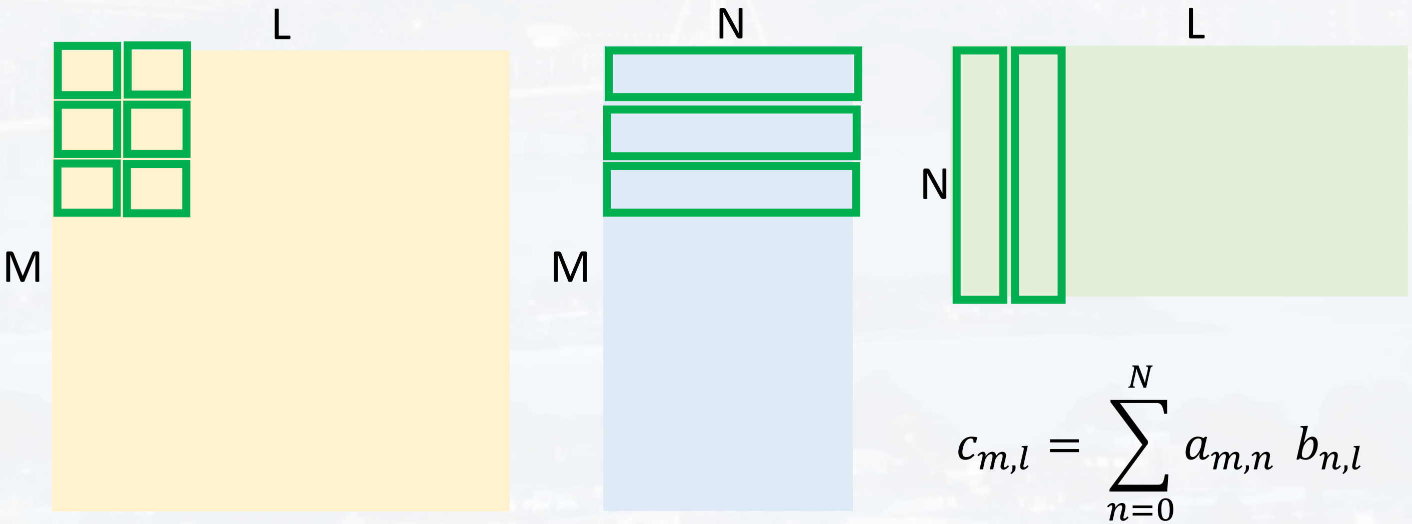
```
array([[ 1,  2,  5,  6],
       [ 0,  8,  5, -4],
       [-4,  4,  6,  1],
       [ 2,  3, -1,  0]])
```

```
np.dot(M,M)
```

```
[[ -7,  56,  39,   3],
 [-28,  72,  74, -27],
 [-26,  51,  35, -34],
 [  6,  24,  19,  -1]]
```

actual matrix multiplication

C = A*B



$$c_{m,l} = \sum_{n=0}^{N} a_{m,n}\ b_{n,l}$$

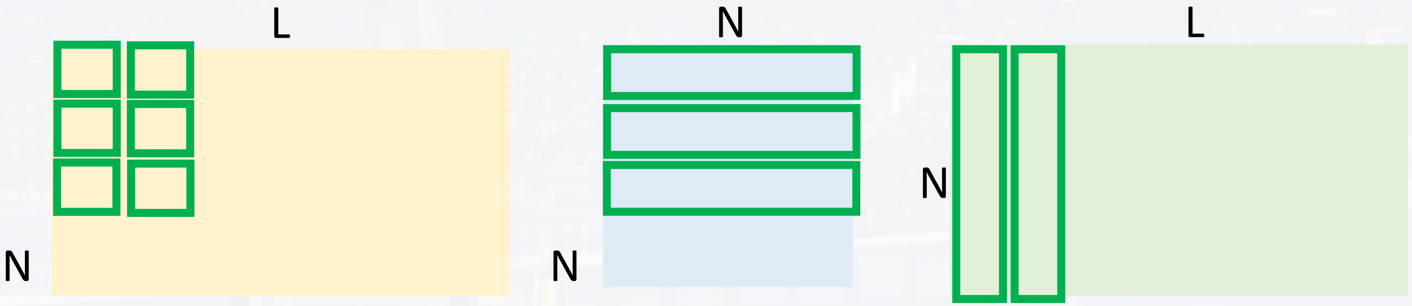
identity matrix

$$I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\mathrm{I} = \begin{pmatrix} 1 & \cdots & 0 \\ \vdots & 1 & \vdots \\ 0 & \cdots & 1 \end{pmatrix}$$

actual matrix multiplication

B = I*B



$$c_{m,l} = \sum_{n=0}^{N} a_{m,n}\ b_{n,l}$$

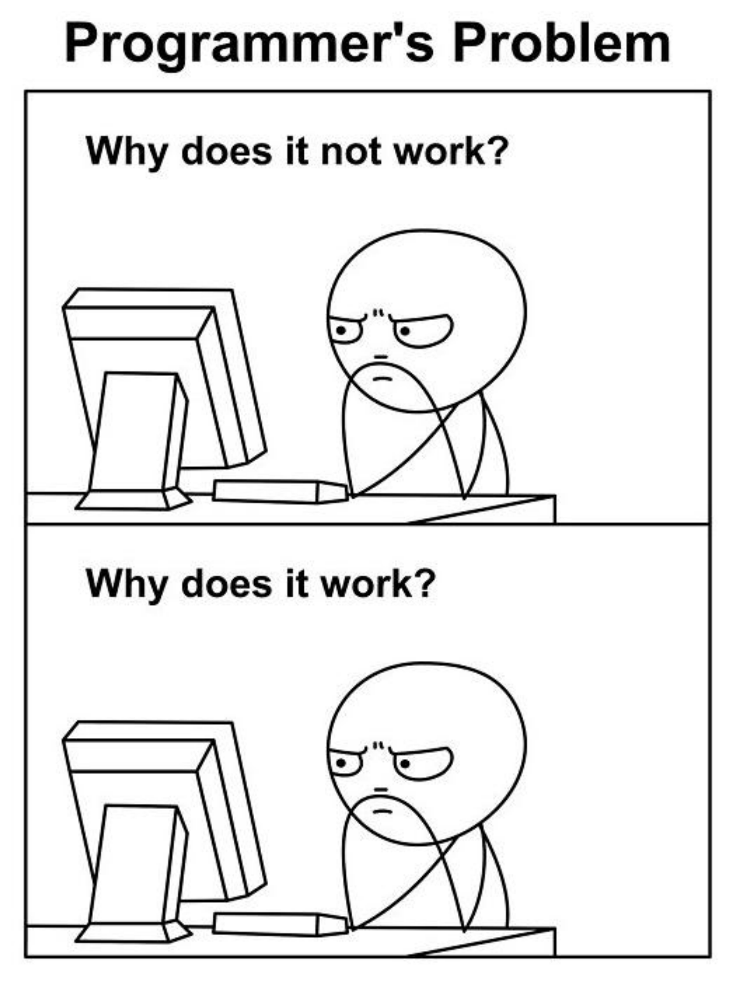
```
np.eye(5)
```

```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

```
M = np.array([[1,2,3,4], [5,6,7,8],\
              [9,10,11,12], [13,14,15,16]])
```

```
In [9]: M
Out[9]:
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
```

```
In [10]: M.transpose()
Out[10]:
array([[ 1,  5,  9, 13],
       [ 2,  6, 10, 14],
       [ 3,  7, 11, 15],
       [ 4,  8, 12, 16]])
```



Outline

- introduction to numpy
- linear algebra with numpy
- **avoiding loops**
- random numbers

We saw earlier: loops are slow → use linear algebra or numpy commands

broadcasting

"Broadcasting describes how NumPy treats arrays with **different shapes during arithmetic operations**."

last section: numpy operations following linear algebra
now: additional useful operations

```
import numpy as np

A1 = np.array([1, 3, 4, 6])
A2 = np.array([2, 4,-1, 0])

A3 = A1*A2
```

```
A3
array([ 2, 12, -4,  0])
```

automatically performs element wise multiplication

broadcasting

```
import numpy as np

A1 = np.array([1, 3, 4, 6])
A2 = np.array([2, 4,-1, 0])

A3 = A1*A2
```

```
A3
array([ 2, 12, -4,  0])
```

automatically performs element wise multiplication

```
A1.shape
(4,)

A2.shape
(4,)
```

```
A2 = A2.reshape((1, len(A2)))
```

```
A2.shape
(1, 4)

A1*A2
array([[ 2, 12, -4,  0]])
```

Works too!

broadcasting

```
import numpy as np

A1 = np.array([1, 3, 4, 6])
A2 = np.array([2, 4,-1, 0])

A3 = A1*A2
```

```
A1*3
array([ 3,  9, 12, 18])
```

Multiplying with a number (no shape!)
→ still element wise multiplication

```
A4 = np.array([3,-5, 6])

A4*A1
```

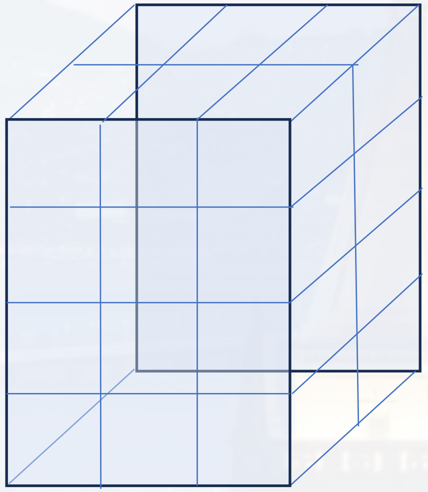
```
Traceback (most recent call last):

  Cell In[23], line 1
    A4*A1

ValueError: operands could not be broadcast together with shapes (3,) (4,)
```

broadcasting error!

What are the broadcasting rules?

What are the broadcasting rules?

Think about matrices like boxes! As long as you can turn them such a way that the edges match, broadcasting is applicable!



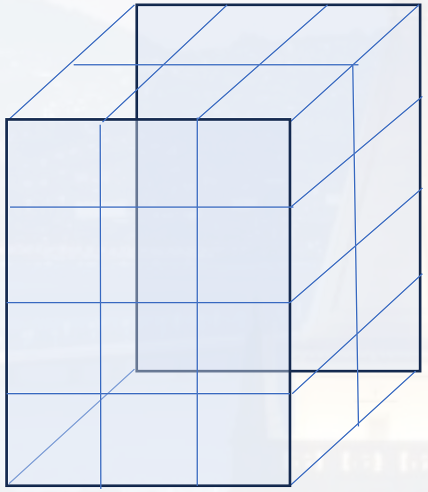
```
A1 = np.array(

[ [[1,   -1], [2,   -2], [3,   -3]],
  [[4,   -4], [5,   -5], [6,   -6]],
  [[7,   -7], [8,   -8], [9,   -9]],
  [[10, -10], [11, -11], [12, -12]] ]

              )
```

```
A1.shape
(4, 3, 2)
```

What are the broadcasting rules?

broadcasting

Think about matrices like boxes! As long as you can turn them such a way that the edges match, broadcasting is applicable!



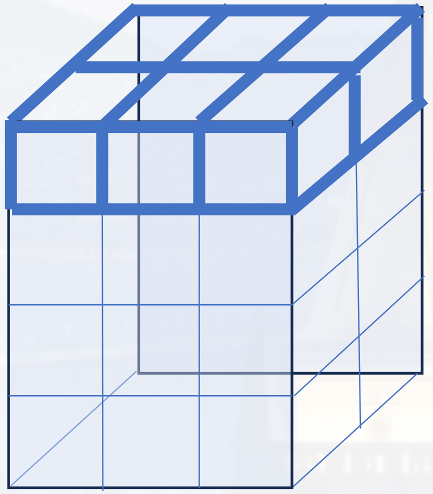
```
A1 = np.array(

[ [[1,   -1], [2,   -2], [3,   -3]],
  [[4,   -4], [5,   -5], [6,   -6]],
  [[7,   -7], [8,   -8], [9,   -9]],
  [[10, -10], [11, -11], [12, -12]] ]

)
```

```
A1.shape
(4, 3, 2)
```

What are the broadcasting rules?

broadcasting

Think about matrices like boxes! As long as you can turn them such a way that the edges match, broadcasting is applicable!



```
A1 = np.array(

[ [[1,   -1], [2,   -2], [3,   -3]],
  [[4,   -4], [5,   -5], [6,   -6]],
  [[7,   -7], [8,   -8], [9,   -9]],
  [[10, -10], [11, -11], [12, -12]] ]

            )
```

```
A1.shape
(4, 3, 2)
```

	0	1
0	1	-1
1	2	-2
2	3	-3

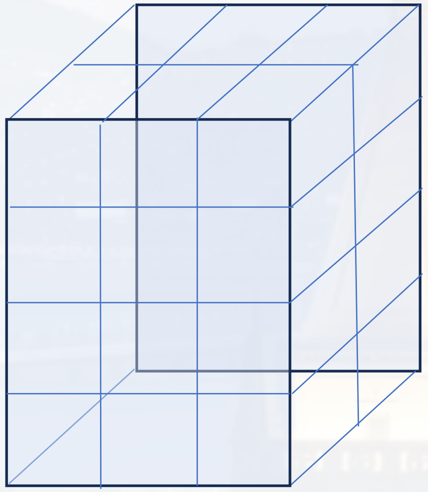
Axis: 0 Shape: (4,

Slicing: [0, :, :]

What are the broadcasting rules?

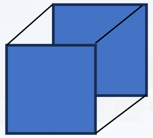
broadcasting

Think about matrices like boxes! As long as you can turn them such a way that the edges match, broadcasting is applicable!



```
A1 = np.array([   [[1,   -1], [2,   -2], [3,   -3]],
                  [[4,   -4], [5,   -5], [6,   -6]],
                  [[7,   -7], [8,   -8], [9,   -9]],
                  [[10, -10], [11, -11], [12, -12]] ])
```

```
A1.shape
(4, 3, 2)
```



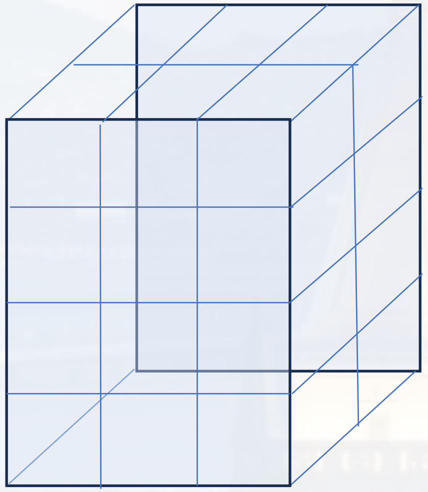
adding/multiplying
an object of shape 0, (1,) or (1,1)

Always fits any part of the box → works!

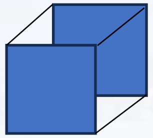
What are the broadcasting rules?

broadcasting

Think about matrices like boxes! As long as you can turn them such a way that the edges match, broadcasting is applicable!



```
A1 = np.array([   [[1,   -1], [2,   -2], [3,   -3]],
                  [[4,   -4], [5,   -5], [6,   -6]],
                  [[7,   -7], [8,   -8], [9,   -9]],
                  [[10, -10], [11, -11], [12, -12]] ])
```



```
a = 3
```

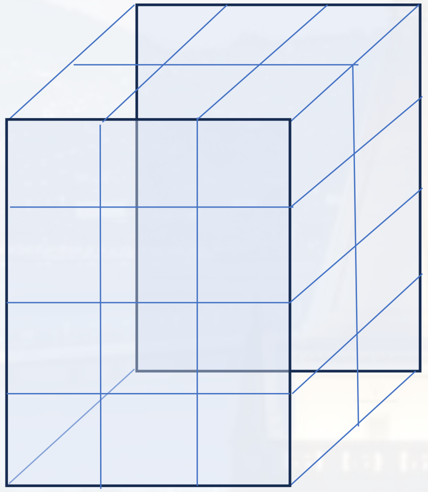
```
A1.shape
(4, 3, 2)
```

adding/multiplying
an object of shape `0`, `(1,)` or `(1,1)`

```
In [4]: A1 * a
Out[4]:
array([[[  3,  -3],
        [  6,  -6],
        [  9,  -9]],

       [[ 12, -12],
        [ 15, -15],
        [ 18, -18]],

       [[ 21, -21],
        [ 24, -24],
        [ 27, -27]],

       [[ 30, -30],
        [ 33, -33],
        [ 36, -36]]])
```

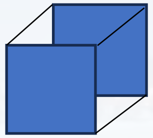
What are the broadcasting rules?

Think about matrices like boxes! As long as you can turn them such a way that the edges match, broadcasting is applicable!



```
A1 = np.array([   [[1,   -1], [2,   -2], [3,   -3]],
                  [[4,   -4], [5,   -5], [6,   -6]],
                  [[7,   -7], [8,   -8], [9,   -9]],
                  [[10, -10], [11, -11], [12, -12]] ])
```

```
A1.shape
(4, 3, 2)
```



```
a = np.array([3])
```

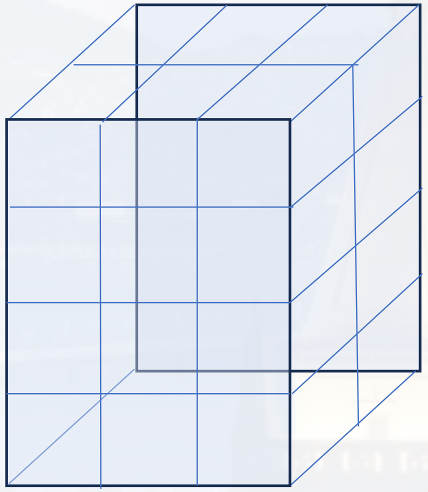
```
a.shape
(1,)
```

adding/multiplying
an object of shape 0, (1,) or (1,1)

```
In [4]: A1 * a
Out[4]:
array([[[  3,  -3],
        [  6,  -6],
        [  9,  -9]],

       [[ 12, -12],
        [ 15, -15],
        [ 18, -18]],

       [[ 21, -21],
        [ 24, -24],
        [ 27, -27]],

       [[ 30, -30],
        [ 33, -33],
        [ 36, -36]]])
```

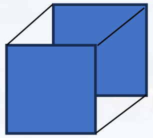
What are the broadcasting rules?

Think about matrices like boxes! As long as you can turn them such a way that the edges match, broadcasting is applicable!



```
A1 = np.array([   [[1,   -1], [2,   -2], [3,   -3]],
                  [[4,   -4], [5,   -5], [6,   -6]],
                  [[7,   -7], [8,   -8], [9,   -9]],
                  [[10, -10], [11, -11], [12, -12]] ])
```

```
A1.shape
(4, 3, 2)
```



```
a = np.array([[3]])
```

```
a.shape
(1, 1)
```

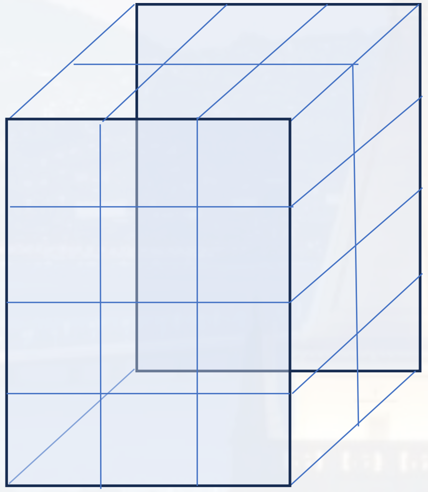
adding/multiplying
an object of shape `0`, `(1,)` or `(1,1)`

```
In [4]: A1 * a
Out[4]:
array([[[  3,  -3],
        [  6,  -6],
        [  9,  -9]],

       [[ 12, -12],
        [ 15, -15],
        [ 18, -18]],

       [[ 21, -21],
        [ 24, -24],
        [ 27, -27]],

       [[ 30, -30],
        [ 33, -33],
        [ 36, -36]]])
```

What are the broadcasting rules?

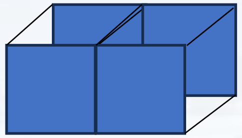
broadcasting

Think about matrices like boxes! As long as you can turn them such a way that the edges match, broadcasting is applicable!



```
A1 = np.array([   [[1,   -1], [2,   -2], [3,   -3]],
                  [[4,   -4], [5,   -5], [6,   -6]],
                  [[7,   -7], [8,   -8], [9,   -9]],
                  [[10, -10], [11, -11], [12, -12]] ])
```

```
array([[[ 3,  3],
        [ 6,  6],
        [ 9,  9]],

       [[12, 12],
        [15, 15],
        [18, 18]],

       [[21, 21],
        [24, 24],
        [27, 27]],

       [[30, 30],
        [33, 33],
        [36, 36]]])
```



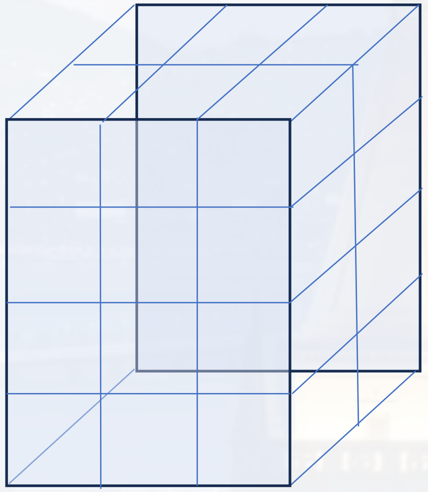
```
a = np.array([3, -3])
```

```
A1.shape
(4, 3, 2)
```

adding/multiplying
an object of shape (2,)

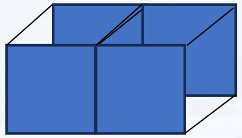
What are the broadcasting rules?

Think about matrices like boxes! As long as you can turn them such a way that the edges match, broadcasting is applicable!



```
A1 = np.array([   [[1,   -1], [2,   -2], [3,   -3]],
                  [[4,   -4], [5,   -5], [6,   -6]],
                  [[7,   -7], [8,   -8], [9,   -9]],
                  [[10, -10], [11, -11], [12, -12]] ])
```

```
A1.shape
(4, 3, 2)
```



```
a = np.array([3, -3])
```

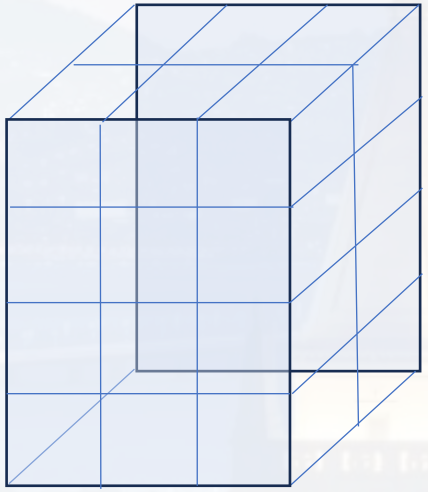
adding/multiplying an object of shape (2,)

Works, because a was just oriented the right way!

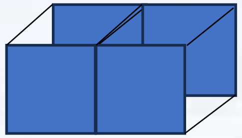
```
array([[[ 3,   3],
        [ 6,   6],
        [ 9,   9]],

       [[12,  12],
        [15,  15],
        [18,  18]],

       [[21,  21],
        [24,  24],
        [27,  27]],

       [[30,  30],
        [33,  33],
        [36,  36]]])
```

What are the broadcasting rules?

Think about matrices like boxes! As long as you can turn them such a way that the edges match, broadcasting is applicable!



```
A1 = np.array([   [[1,   -1], [2,   -2], [3,   -3]],
                  [[4,   -4], [5,   -5], [6,   -6]],
                  [[7,   -7], [8,   -8], [9,   -9]],
                  [[10, -10], [11, -11], [12, -12]] ])
```



```
a = np.array([[3], [-3]])
```

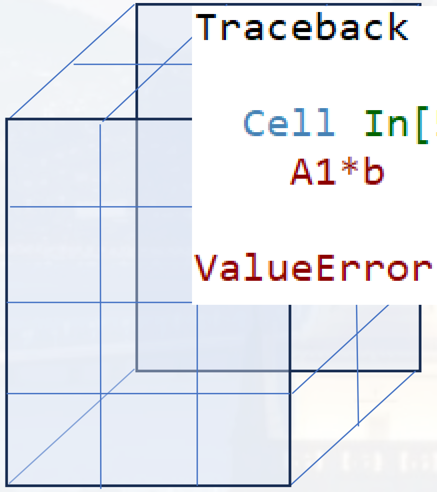
```
A1.shape
(4, 3, 2)
```

adding/multiplying
an object of shape (2,1)

What are the broadcasting rules?

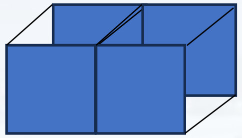
broadcasting

Think about matrices like boxes! As long as you can turn them such a way that the edges match, broadcasting is applicable!



```
Traceback (most recent call last):

  Cell In[59], line 1
    A1*b

ValueError: operands could not be broadcast together with shapes (4,3,2) (2,1)
```

```
A1.shape
(4, 3, 2)
```



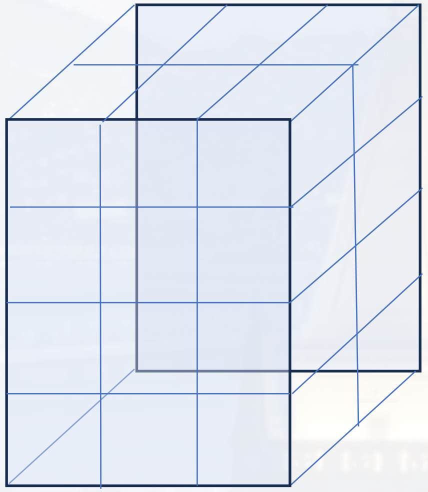
```
a = np.array([[3], [-3]])
```

adding/multiplying
an object of shape (2,1)

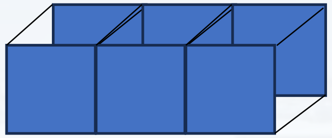
What are the broadcasting rules?

broadcasting

Think about matrices like boxes! As long as you can turn them such a way that the edges match, broadcasting is applicable!



```
A1 = np.array([   [[1,   -1], [2,   -2], [3,   -3]],
                  [[4,   -4], [5,   -5], [6,   -6]],
                  [[7,   -7], [8,   -8], [9,   -9]],
                  [[10, -10], [11, -11], [12, -12]] ])
```



```
a = np.array([3, 0, -3])
```

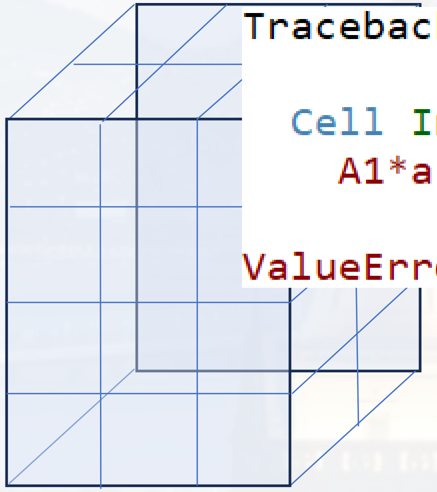
```
A1.shape
(4, 3, 2)
```

adding/multiplying
an object of shape (3,)

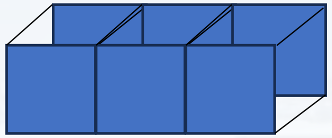
What are the broadcasting rules?

broadcasting

Think about matrices like boxes! As long as you can turn them such a way that the edges match, broadcasting is applicable!



```
Traceback (most recent call last):

  Cell In[63], line 1
    A1*a

ValueError: operands could not be broadcast together with shapes (4,3,2) (3,)
```



```
a = np.array([3, 0, -3])
```

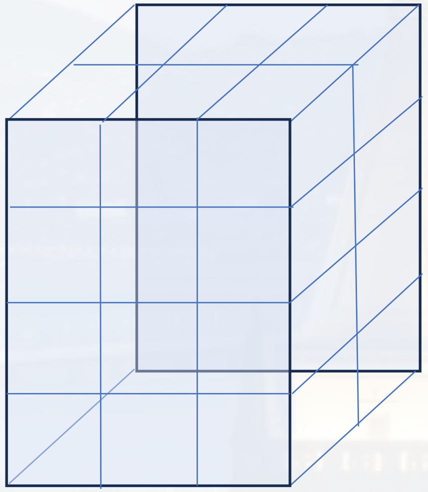
```
A1.shape
(4, 3, 2)
```

adding/multiplying
an object of shape (3,)

What are the broadcasting rules?

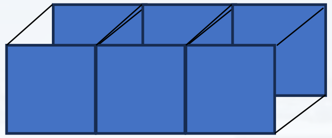
broadcasting

Think about matrices like boxes! As long as you can turn them such a way that the edges match, broadcasting is applicable!



```
A1 = np.array([   [[1,   -1], [2,   -2], [3,   -3]],
                  [[4,   -4], [5,   -5], [6,   -6]],
                  [[7,   -7], [8,   -8], [9,   -9]],
                  [[10, -10], [11, -11], [12, -12]] ])
```

```
a = np.array([[3], [0],[ -3]])
```



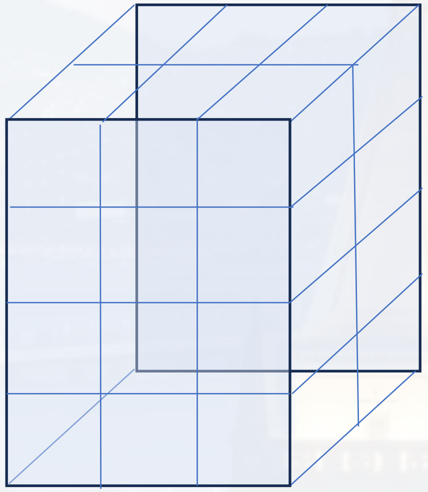
```
A1.shape
(4, 3, 2)
```

adding/multiplying
an object of shape (3,1)

What are the broadcasting rules?

broadcasting

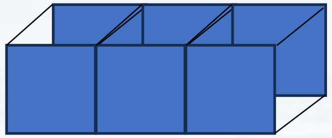
Think about matrices like boxes! As long as you can turn them such a way that the edges match, broadcasting is applicable!



```
A1 = np.array([   [[1,   -1], [2,   -2], [3,   -3]],
                  [[4,   -4], [5,   -5], [6,   -6]],
                  [[7,   -7], [8,   -8], [9,   -9]],
                  [[10, -10], [11, -11], [12, -12]]
```

```
A1.shape
(4, 3, 2)
```

```
a = np.array([[3], [0],[ -
```

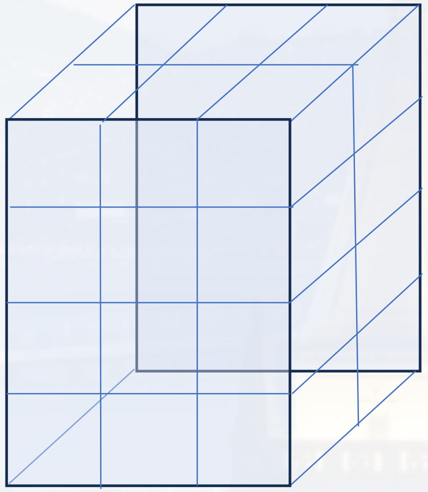


adding/multiplying
an object of shape (3,1)

```
array([[[  3,  -3],
        [  0,   0],
        [ -9,   9]],

       [[ 12, -12],
        [  0,   0],
        [-18,  18]],

       [[ 21, -21],
        [  0,   0],
        [-27,  27]],

       [[ 30, -30],
        [  0,   0],
        [-36,  36]]])
```

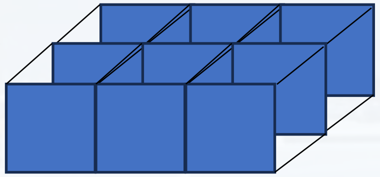
What are the broadcasting rules?

Think about matrices like boxes! As long as you can turn them such a way that the edges match, broadcasting is applicable!



```
A1 = np.array([   [[1,   -1], [2,   -2], [3,   -3]],
                  [[4,   -4], [5,   -5], [6,   -6]],
                  [[7,   -7], [8,   -8], [9,   -9]],
                  [[10, -10], [11, -11], [12, -12]] ])
```

```
a = np.array([[3, 0, -3], [2, 0, -2]])
```



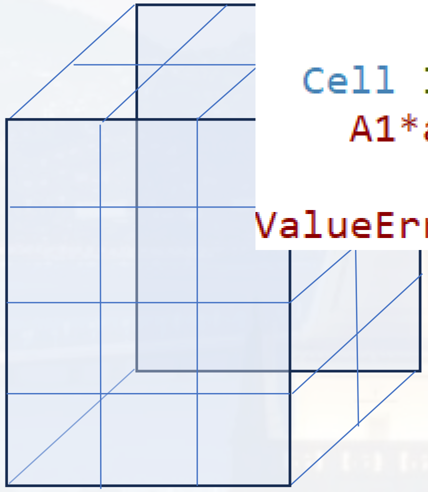
```
A1.shape
(4, 3, 2)
```

adding/multiplying
an object of shape (2,3)

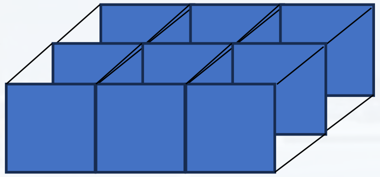
What are the broadcasting rules?

Think about matrices like boxes! As long as you can turn them such a way that the edges match, broadcasting is applicable!

```
In [8]: A1*a
Traceback (most recent call last):

  Cell In[8], line 1
    A1*a

ValueError: operands could not be broadcast together with shapes (4,3,2) (2,3)
```



```
a = np.array([[3, 0, -3], [2, 0, -2]])
```



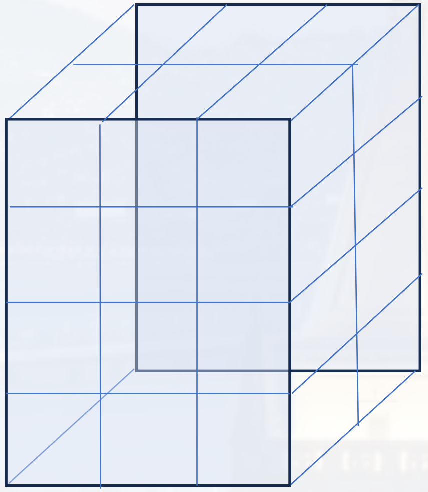
```
A1.shape
(4, 3, 2)
```

adding/multiplying
an object of shape (2,3)

What are the broadcasting rules?

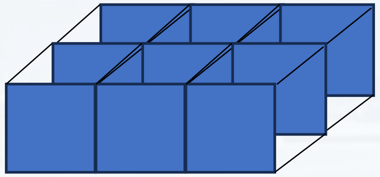
broadcasting

Think about matrices like boxes! As long as you can turn them such a way that the edges match, broadcasting is applicable!



```
A1 = np.array([   [[1,   -1], [2,   -2], [3,   -3]],
                  [[4,   -4], [5,   -5], [6,   -6]],
                  [[7,   -7], [8,   -8], [9,   -9]],
                  [[10, -10], [11, -11], [12, -12]] ])
```

```
A1.shape
(4, 3, 2)
```

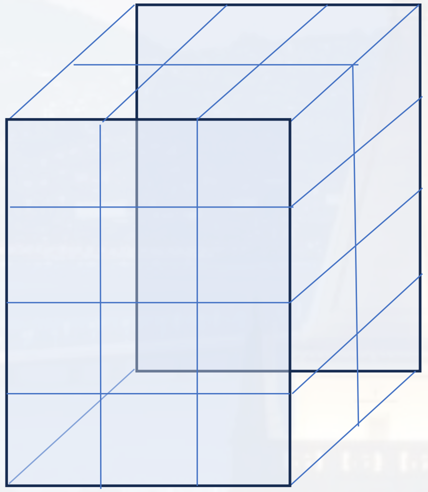


```
a = np.array([[3, 0, -3], [2, 0, -2]])

a = a.reshape((3,2))
```

adding/multiplying
an object of shape (3,2)

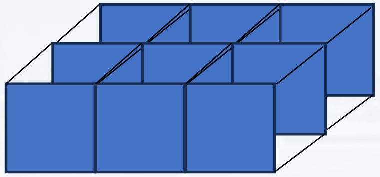
What are the broadcasting rules?

Think about matrices like boxes! As long as you can turn them such a way that the edges match, broadcasting is applicable!



```
A1 = np.array([   [[1,   -1], [2,   -2], [3,   -3]],
                  [[4,   -4], [5,   -5], [6,   -6]],
                  [[7,   -7], [8,   -8], [9,   -9]],
                  [[10, -10], [11, -11], [12, -12]]
```

```
a = np.array([[3, 0, -3], [2, 0, -2]])
```



```
a = a.reshape((3,2))
```

```
A1.shape
(4, 3, 2)
```

adding/multiplying
an object of shape (3,2)

```
array([[[  3,   0],
        [ -6,  -4],
        [  0,   6]],

       [[ 12,   0],
        [-15, -10],
        [  0,  12]],

       [[ 21,   0],
        [-24, -16],
        [  0,  18]],

       [[ 30,   0],
        [-33, -22],
        [  0,  24]]])
```

What are the broadcasting rules?

note: these broadcasting rules work for addition/subtraction too

```
ValueError: operands could not be broadcast together with shapes (4,3,2) (2,1)
```

didn't work because `(2, 1)` doesn't match `(4, 3, 2)`

solution: `(2,)`, it matches `(4, 3, 2)`

```
ValueError: operands could not be broadcast together with shapes (4,3,2) (3,)
```

didn't work because `(3,)` doesn't match `(4, 3, 2)`

solution: `(3, 1)`, it matches `(4, 3, 2)` two times

```
ValueError: operands could not be broadcast together with shapes (4,3,2) (2,3)
```

didn't work because `(2, 3)` doesn't match `(4, 3, 2)`

solution: reshape to `(3, 2)`, it matches `(4, 3, 2)` more useful numpy commands to avoid loops:

statistics

```
np.max()
np.min()
np.mean()
np.median()
np.std()
np.argwhere()
np.historgram()
...
```

rearranging arrays

```
np.reshape()
np.sort()
np.transpose()
np.hstack()
np.vstack()
np.tile()
np.arange()
...
```

resetting values

```
np.clip()
np.abs()
np.round()
np.eye()
...
```

math functions

```
np.exp()
np.sin()
np.cos()
np.tan()
np.arcsin()
np.arccos()
np.arctan()
np.arctanh()
...
```

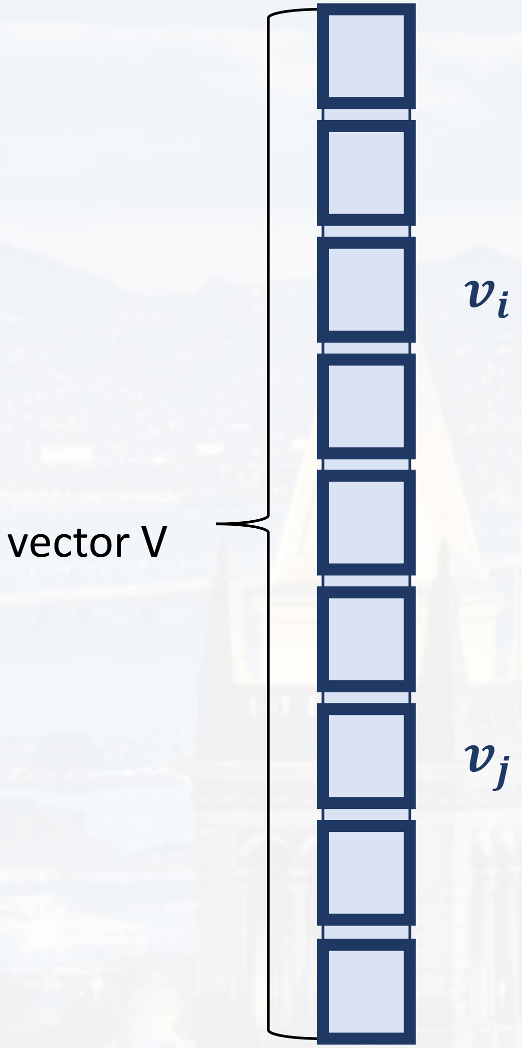
Let's combine some of those commands

```
np.zeros()
np.arange()
np.tile()
np.transpose()
```

for a particular example about avoiding loops!

We saw earlier: loops are slow → use lin algebra or numpy commands

calculating distances d

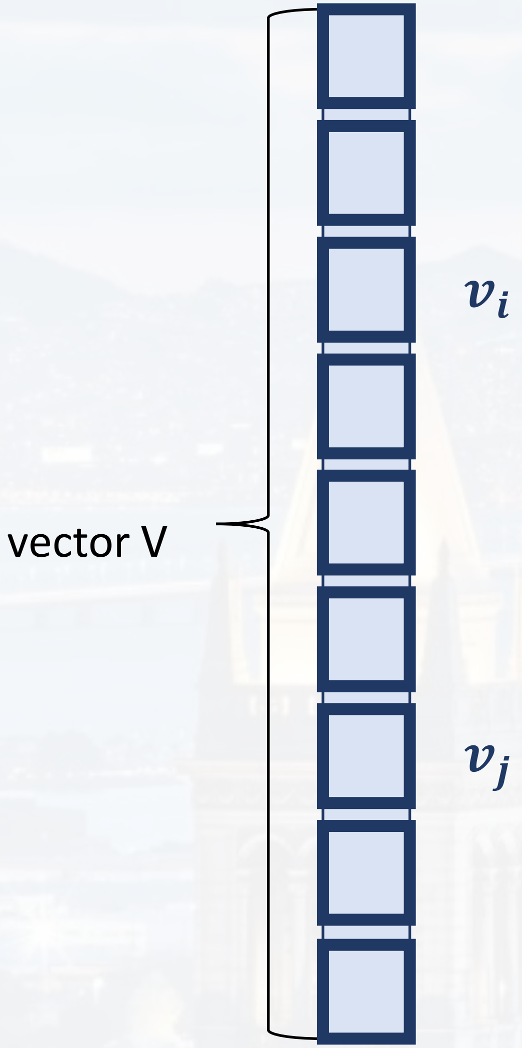


calculating the distance between each element

$$d(v_i, v_j) = (v_i - v_j)^2$$

We saw earlier: loops are slow → use lin algebra or numpy commands

calculating distances d



calculating the distance between each element

$$d(v_i, v_j) = (v_i - v_j)^2$$

efficiency:

vector of length N → N*N operations

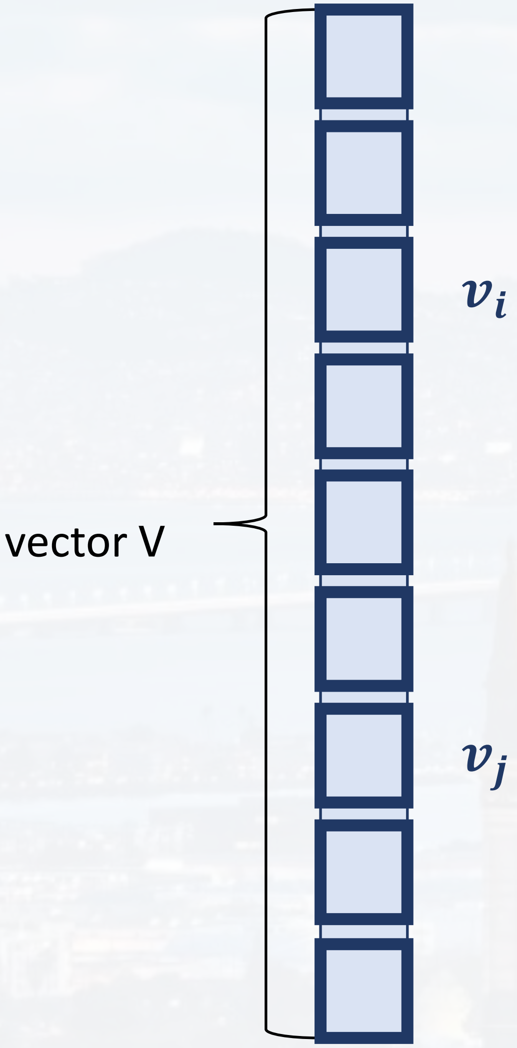
we know that the diagonal $d(v_i, v_i) = 0$ → N fewer operations

only half the operations are necessary: $d(v_i, v_j) = d(v_j, v_i)$

→ instead of N*N operations: only (N-1)*(N-1)/2 needed

We saw earlier: loops are slow → use lin algebra or numpy commands

calculating distances d



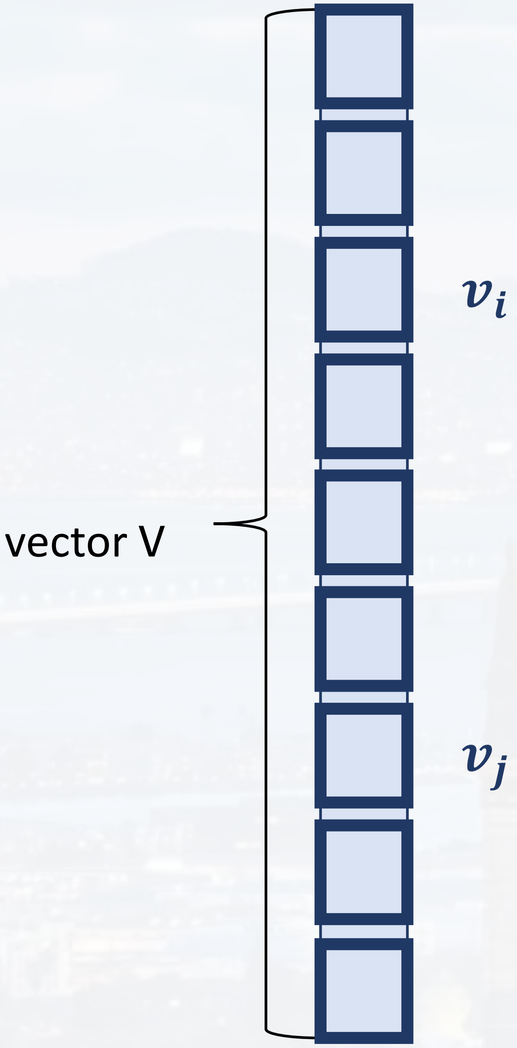
1) normal loop

0.11 sec

```
V = np.arange(start = 0, stop = 500)
N = len(V)

D = np.zeros((N,N))

t1 = time.monotonic()

for i in range(N):
    for j in range(N):
        D[i,j] = (V[i] - V[j])**2

t2 = time.monotonic()
dt = t2 - t1
print('Total runtime: ' + str(dt) + ' seconds')
```

We saw earlier: loops are slow → use lin algebra or numpy commands

calculating distances d



1) normal loop

0.11 sec

```
V = np.arange(start = 0, stop = 500)
N = len(V)
```

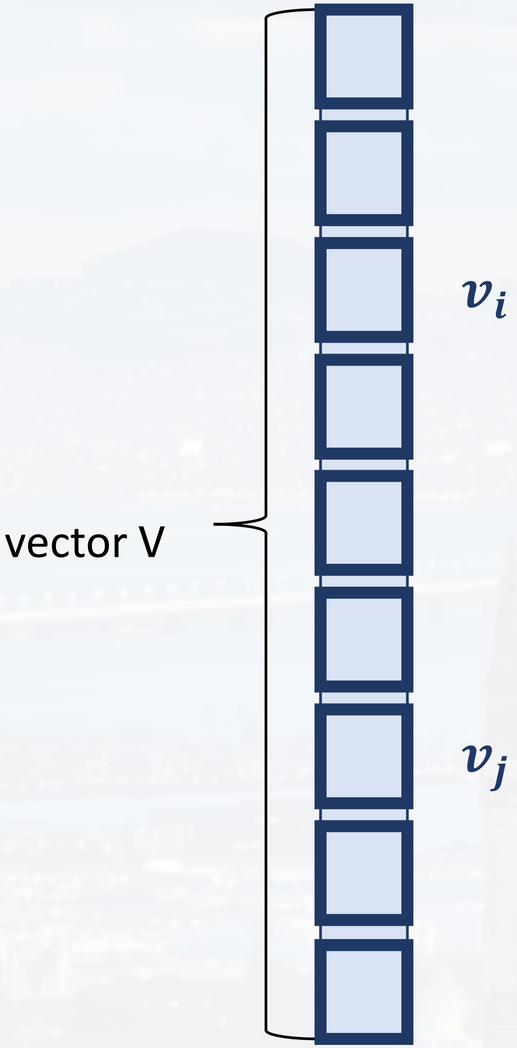
	0	1	2	3
0	0	1	4	9
1	1	0	1	4
2	4	1	0	1
3	9	4	1	0
4	16	9	4	1

```
print('Total runtime: ' + str(dt) + ' seconds')
```

avoiding loops

We saw earlier: loops are slow → use lin algebra or numpy commands

calculating distances d



1) normal loop

0.11 sec

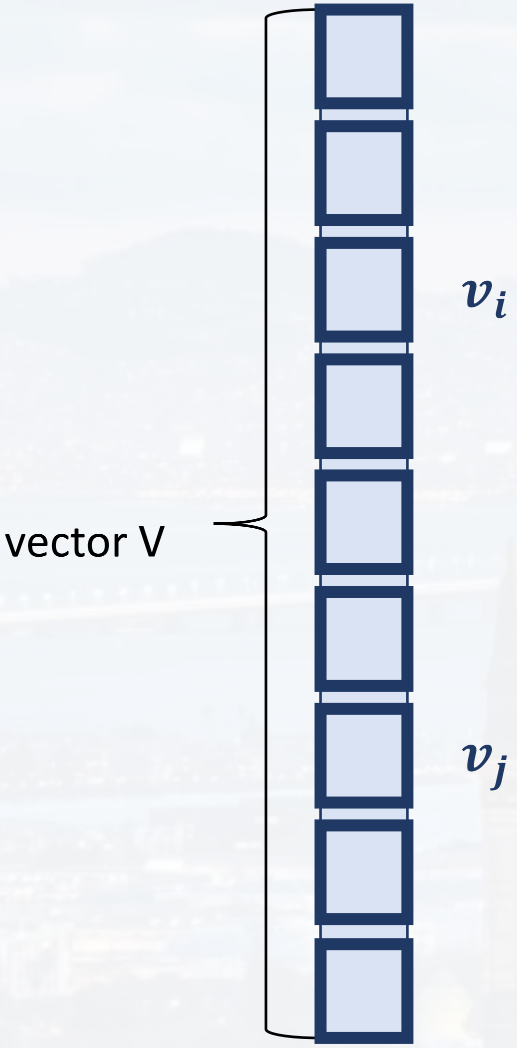
```
V = np.arange(start = 0, stop = 500)
N = len(V)
```

	0	1	2	3
0	0	1	4	9
1	1	0	1	4
2	4	1	0	1
3	9	4	1	0
4	16	9	4	1

```
print('Total runtime: ' + str(dt) + ' seconds')
```

We saw earlier: loops are slow → use lin algebra or numpy commands

calculating distances d



2) efficient loop

0.05 sec

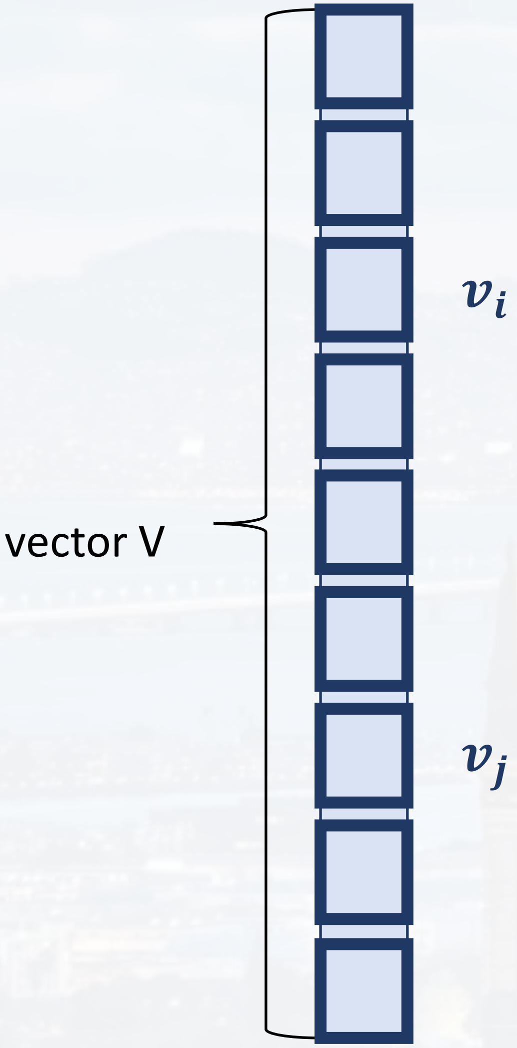
```
V = np.arange(start = 0, stop = 500)
N = len(V)
D = np.zeros((N,N))

t1 = time.monotonic()

for i in range(N):
    for j in range(i):
        D[i,j] = (V[i] - V[j])**2

D += D.transpose()

t2 = time.monotonic()
dt = t2 - t1
print('Total runtime: ' + str(dt) + ' seconds')
```

filling only half the matrix!

adding the other half

We saw earlier: loops are slow → use lin algebra or numpy commands

calculating distances d



3) no loop

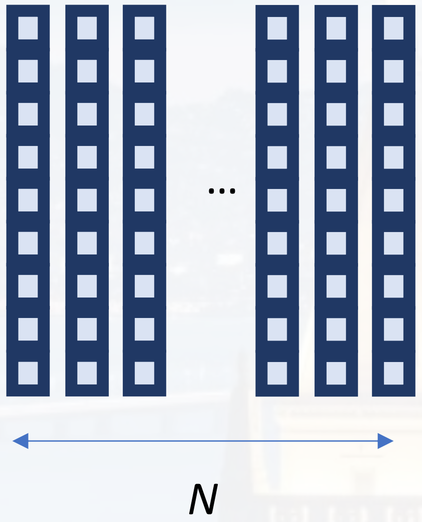
0.0003 sec

```
V = np.arange(start = 0, stop = 500)
N = len(V)

t1 = time.monotonic()


Rep = np.tile(V, (N,1))
D   = (Rep - Rep.transpose())**2


t2 = time.monotonic()
dt = t2 - t1
print('Total runtime:  ' + str(dt) + ' seconds')
```

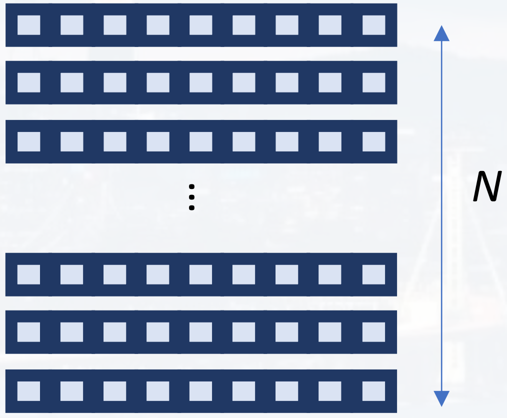
We saw earlier: loops are slow → use lin algebra or numpy commands

calculating distances d

```
Rep = np.tile(V, (N,1))
```

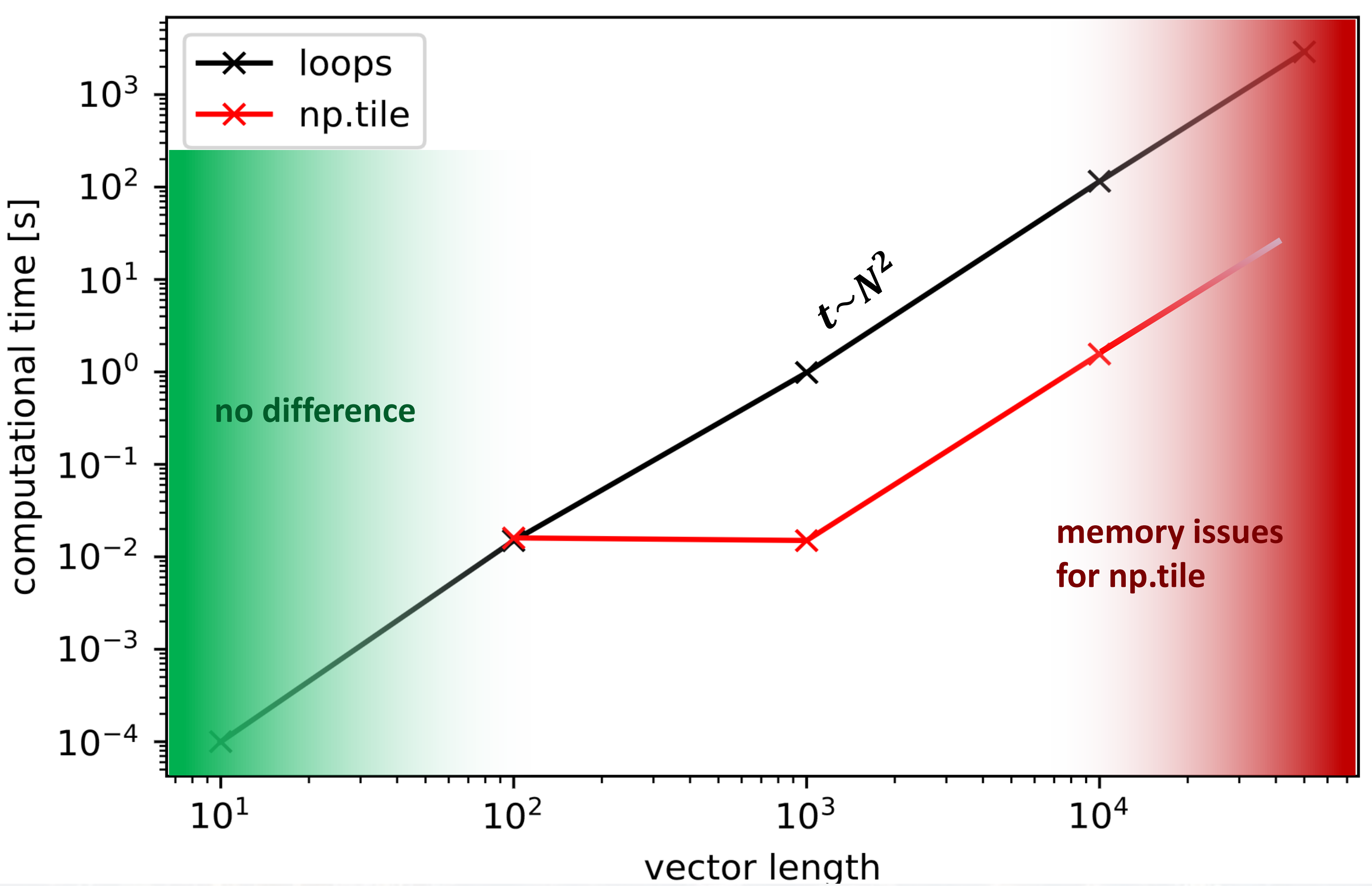


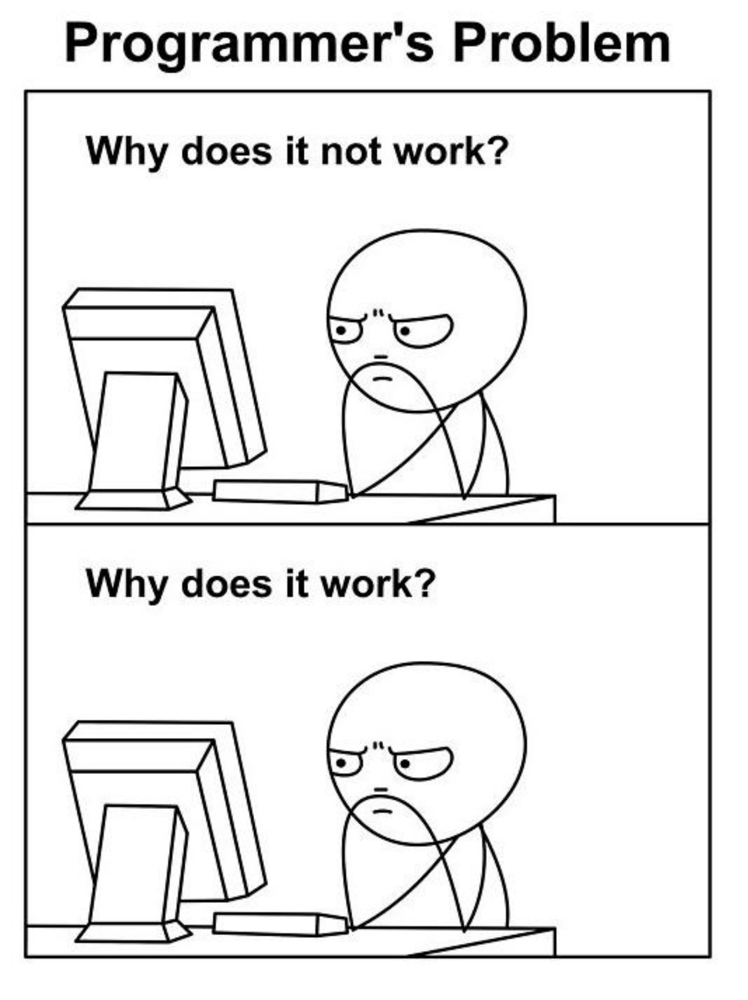

```
D    = Rep.transpose()
```




```
D    = (Rep - Rep.transpose())**2
```

	`loop`	`efficient loop`	`np.tile`
N = 500	**0.11 sec**	**0.05 sec**	**0.0003 sec**
N = 10,000	**35.0 sec**	**17.8 sec**	**1.25 sec**
N = 50,000			**180 sec**


loops
np.tile
computational time [s]
vector length
10^{3}
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{1}
10^{2}
10^{3}
10^{4}
$t \sim N^2$
no difference
memory issues
for np.tile



Outline

- introduction to numpy
- linear algebra with numpy
- avoiding loops
- **random numbers**

Why random numbers: **simulations** (Monte Carlo, Gillespie, Metropolis)
initialization (ANNs, HMMs, ML algorithms like k-means)
modelling (testing theoretical distribution vs data)

```
dir(np.random)
```

```
'beta',
'binomial',
'bit_generator',
'bytes',
'chisquare',
'choice',
'default_rng',
'dirichlet',
'exponential',
'f',
'gamma',
'geometric',
'get_bit_generator',
'get_state',
'gumbel',
'hypergeometric',
'laplace',
'logistic',
'lognormal',
'logseries',
'mtrand',
'multinomial',
'multivariate_normal',
'negative_binomial',
'noncentral_chisquare',
'noncentral_f',
'normal',
'pareto',
'permutation',
'poisson',
'poisson',
'power',
'rand',
'randint',
'randn',
'random',
'random_integers',
'random_sample',
'ranf',
'rayleigh',
'sample',
'seed',
'set_bit_generator',
'set_state',
'shuffle',
'standard_cauchy',
'standard_exponential',
'standard_gamma',
'standard_normal',
'standard_t',
'test',
'triangular',
'uniform',
'vonmises',
'wald',
'weibull',
'zipf']
```

a brief overview

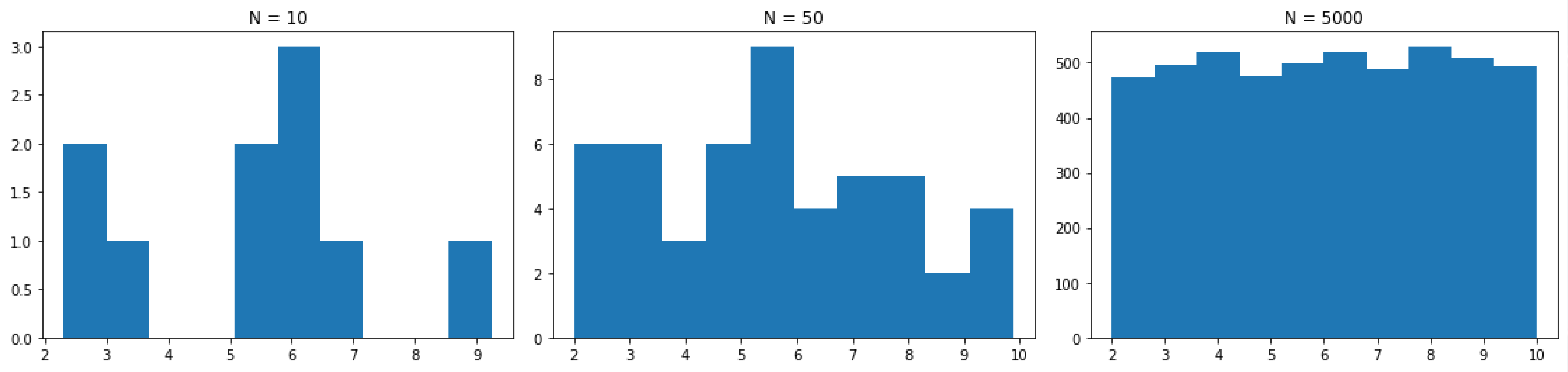
- uniform
- binomial
- poissonian
- normal (gaussian)

a brief overview

- **uniform**
- binomial
- poissonian
- normal (gaussian)

continuous support

```
U = np.random.uniform(low = 2, high = 10, shape = (N, 1))
plt.hist(U)
```

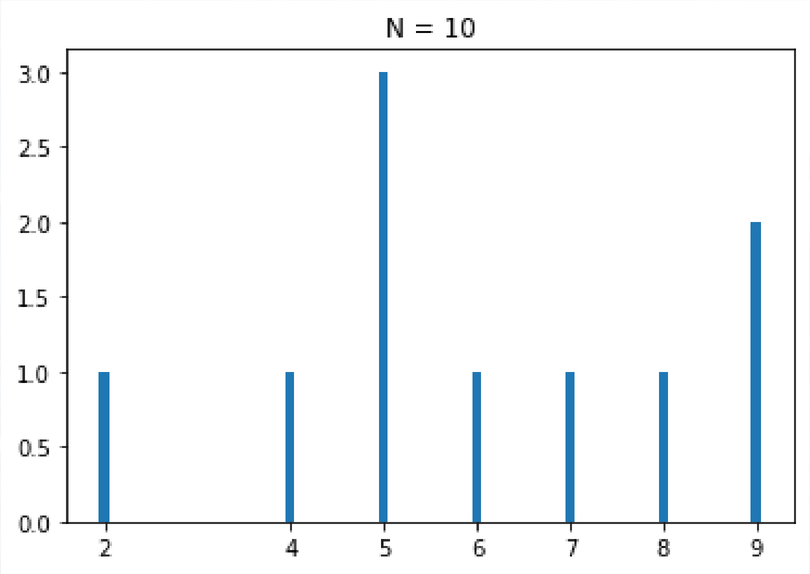


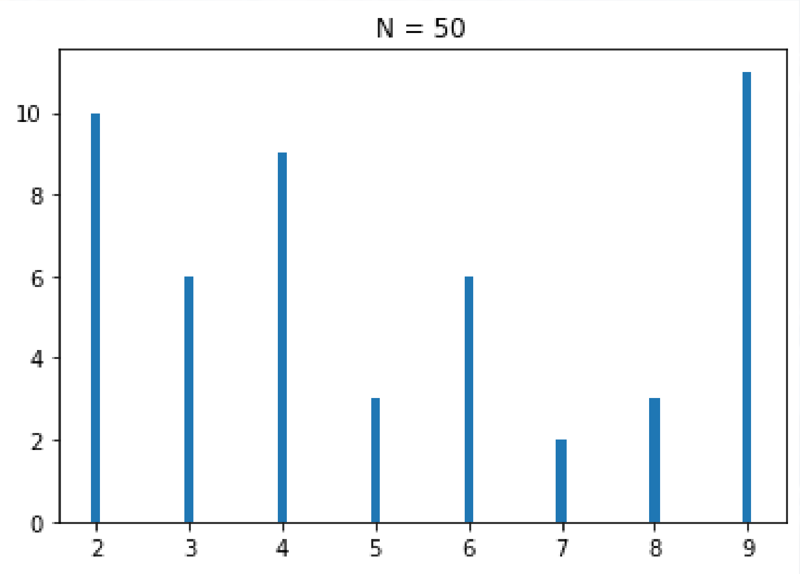
a brief overview

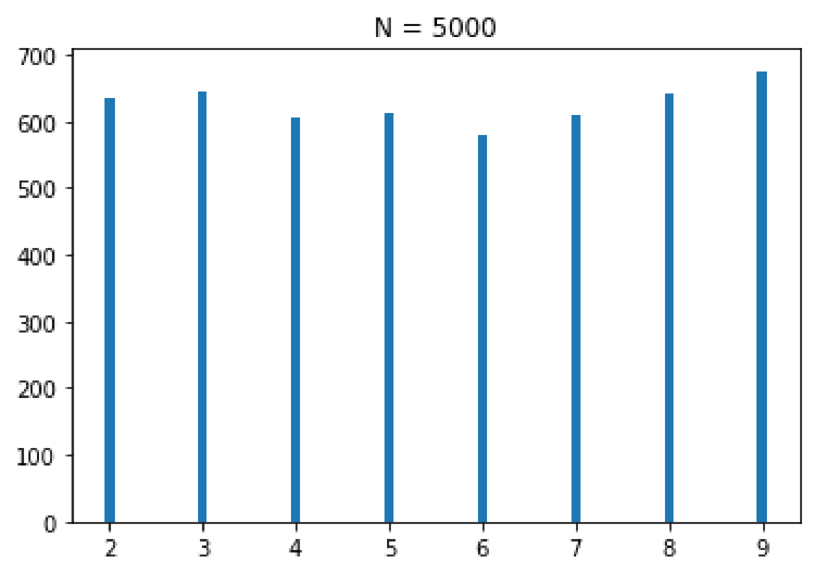
- **uniform**
- binomial
- poissonian
- normal (gaussian)

discrete support

```
U = np.random.randint(low, high, shape)

labels, counts = np.unique(U, return_counts = True)
plt.bar(labels, counts, align = 'center', width = 0.1)
plt.gca().set_xticks(labels)
plt.title('N = ' + str(N))
```



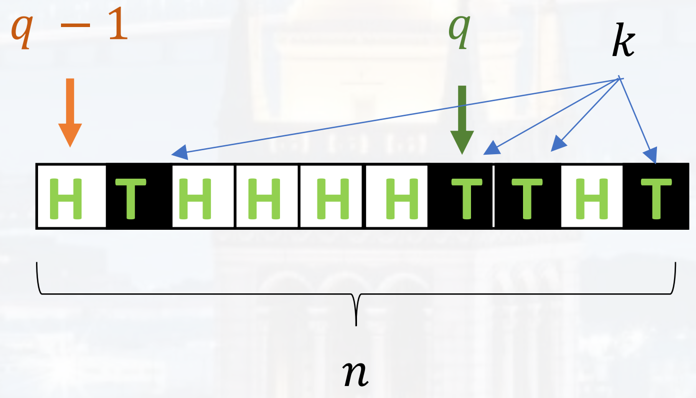




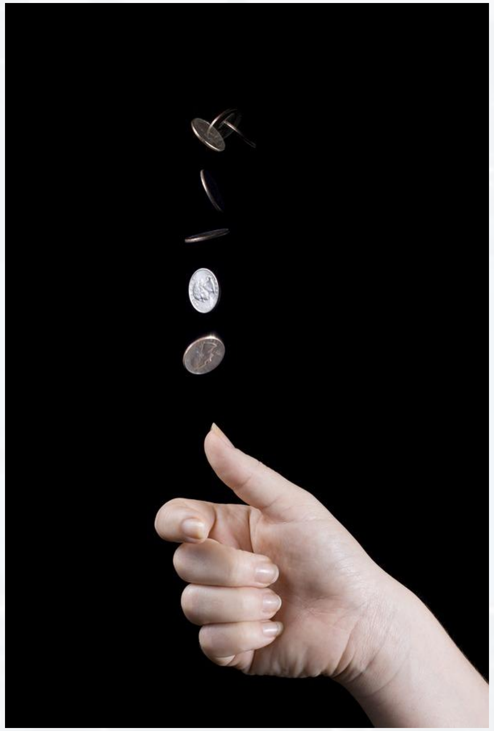
$$\boxed{P(k|q,n) = \binom{n}{k} q^k (1-q)^{n-k}}$$

binomial distribution

- flipping a coin ***n*** times
- biased coin: $\boldsymbol{q} \neq 50\%$ in general
- probability to have ***k*** heads/tails



now: running this experiment
With fixed ***n*** and ***q*** ***N*** times!

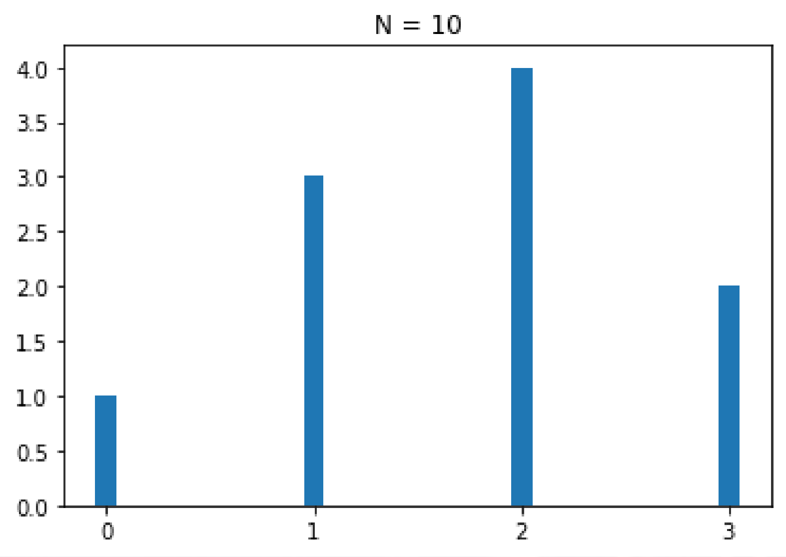


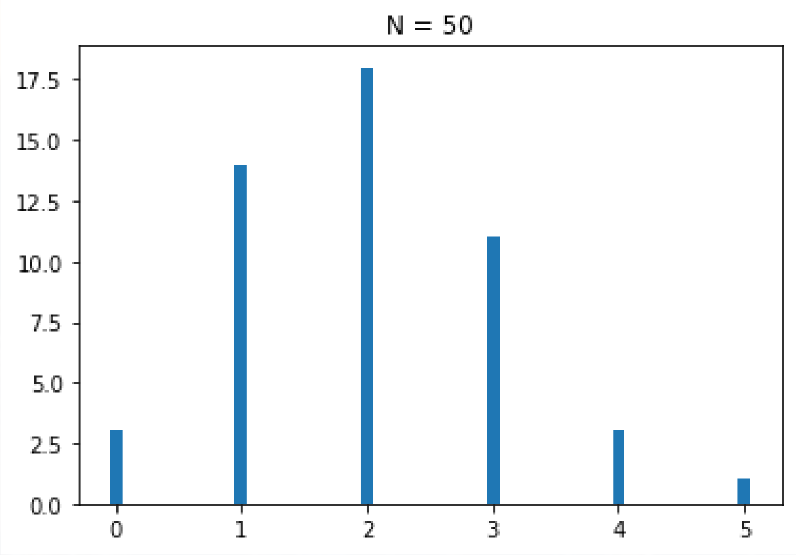
$$\boxed{P(k|q,n) = \binom{n}{k}\, q^k\, (1-q)^{n-k}}$$

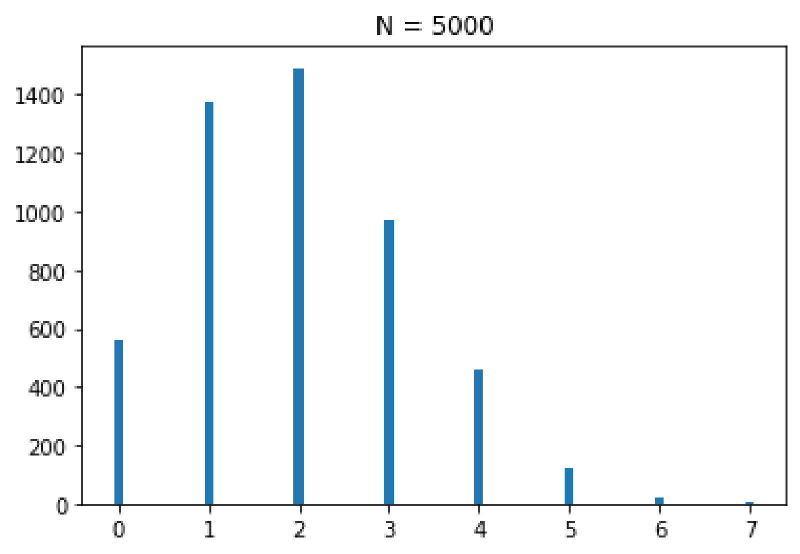
binomial distribution

- uniform
- **binomial**
- poissonian
- normal (gaussian)

```
q = 0.2
n = 10
K = np.random.binomial(n, q, N)

labels, counts = np.unique(K, return_counts = True)
plt.bar(labels, counts, align = 'center', width = 0.1)
plt.gca().set_xticks(labels)
plt.title('N = ' + str(N))
```

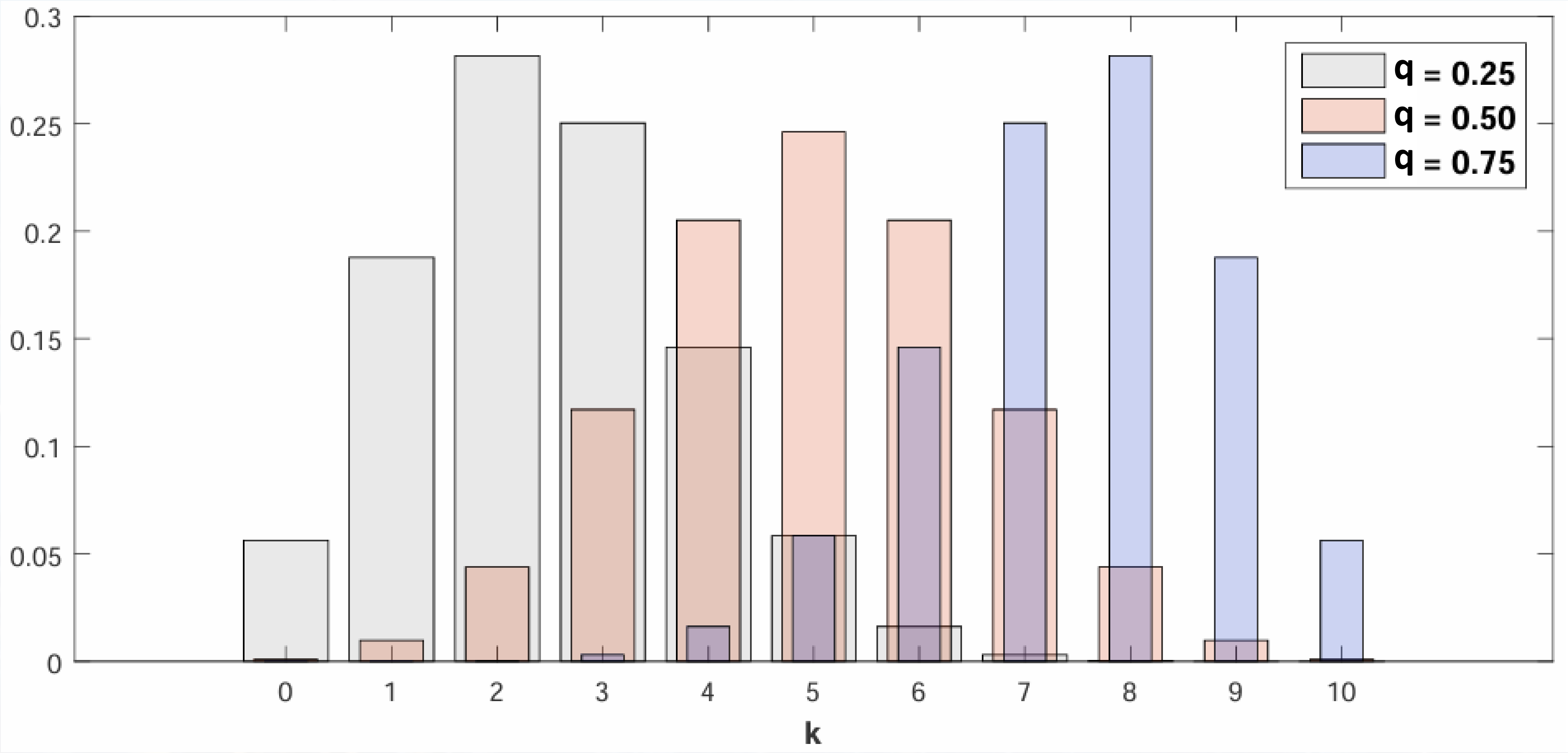






$$\boxed{P(k|q,n) = \binom{n}{k}\, q^k\,(1-q)^{n-k}}$$

binomial distribution

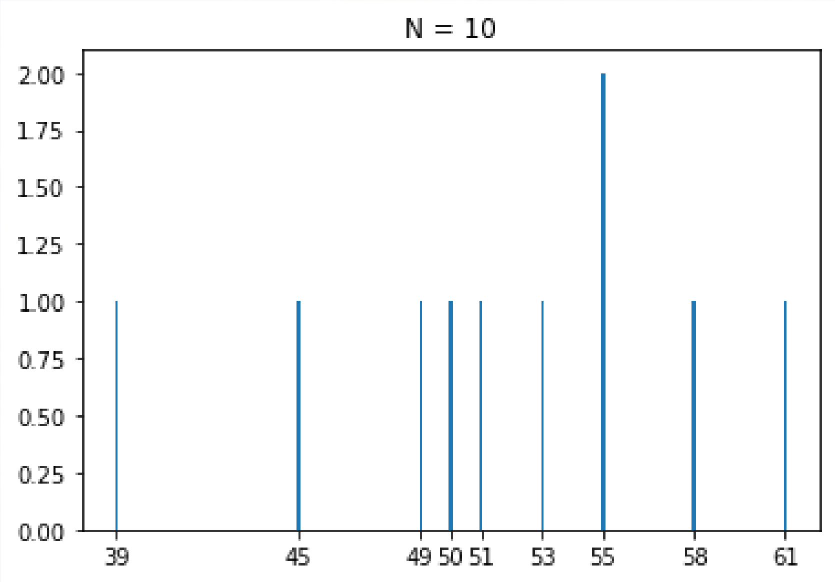


$$P(k|c) = \frac{(c\,\Delta t)^k e^{-c\,\Delta t}}{k!}$$

Poisson distribution

- uniform
- binomial
- **poissonian**
- normal (gaussian)

- rate of ***c*** WhatsUp messages/day
- observational time ***Δt***
- probability to get ***k*** messages within ***Δt***, given ***c***



c = 5 messages per day
Δt = 10 days observational time span

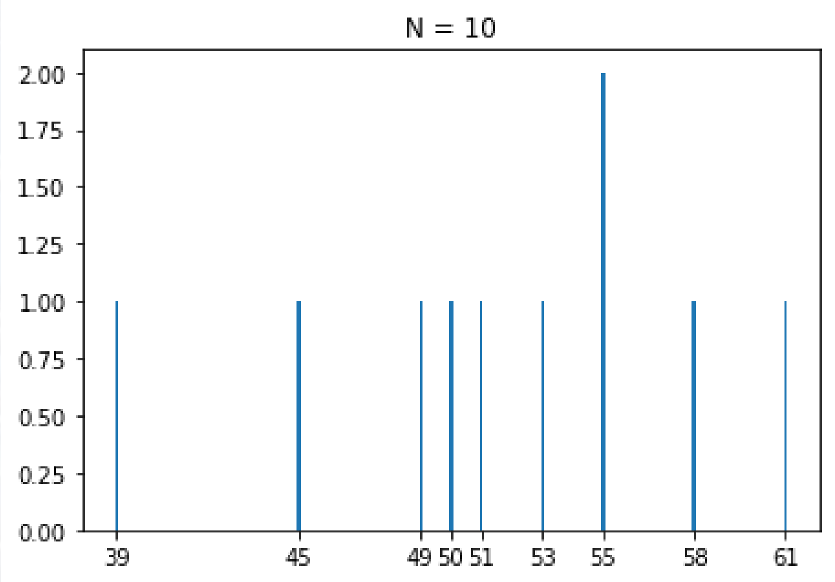
experiment was repeated N = 10 times

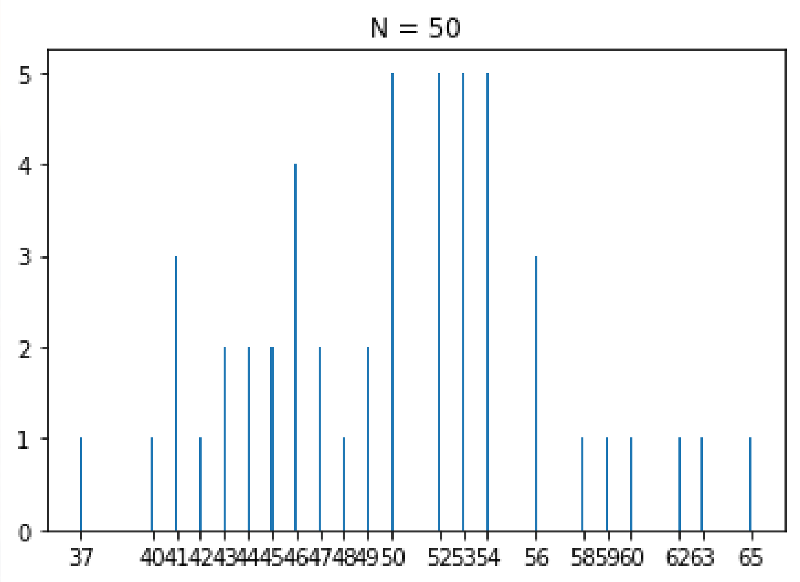
- uniform
- binomial
- **poissonian**
- normal (gaussian)

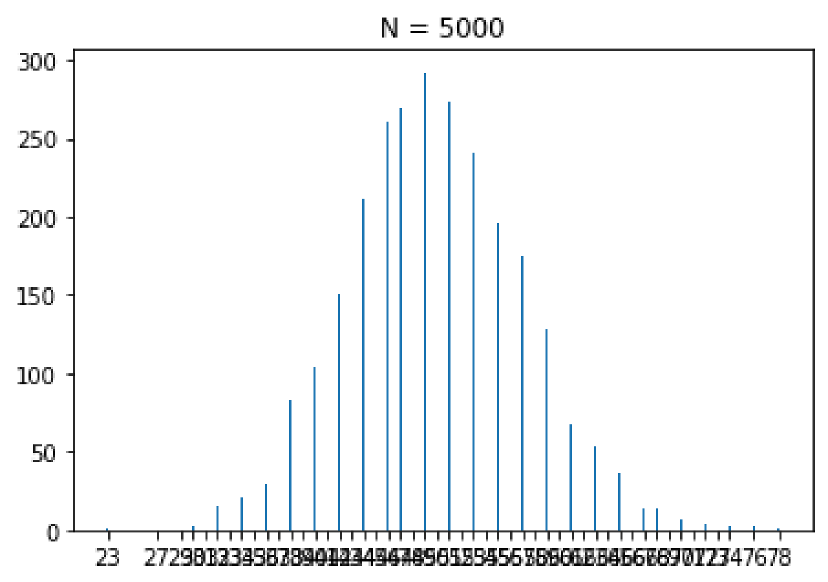
$$\boxed{P(k|c) = \frac{(c\,\Delta t)^k e^{-c\,\Delta t}}{k!}}$$

Poisson distribution

```
c    = 5
delt = 10
lam  = c * delt
K    = np.random.poisson(lam, N)
labels, counts = np.unique(K, return_counts = True)
plt.bar(labels, counts, align = 'center', width = 0.1)
plt.gca().set_xticks(labels)
plt.title('N = ' + str(N))
```



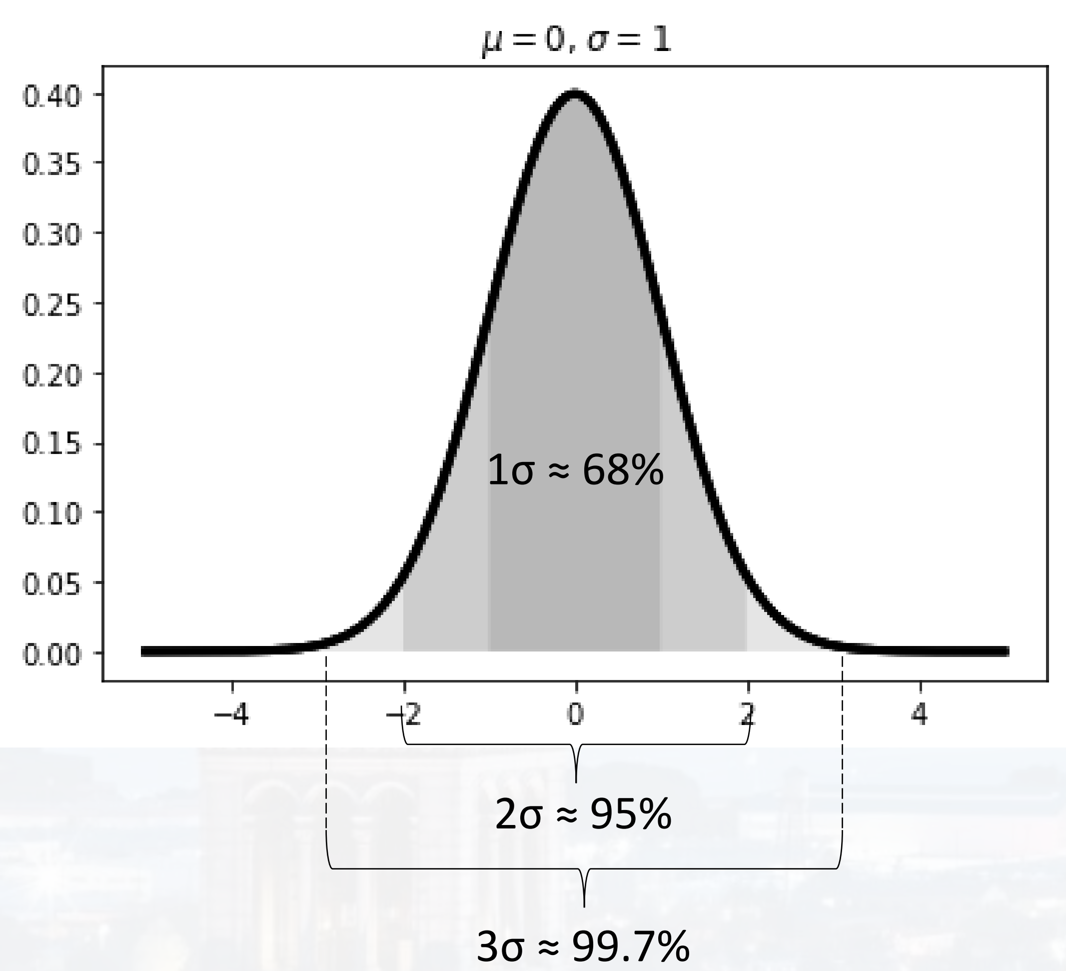


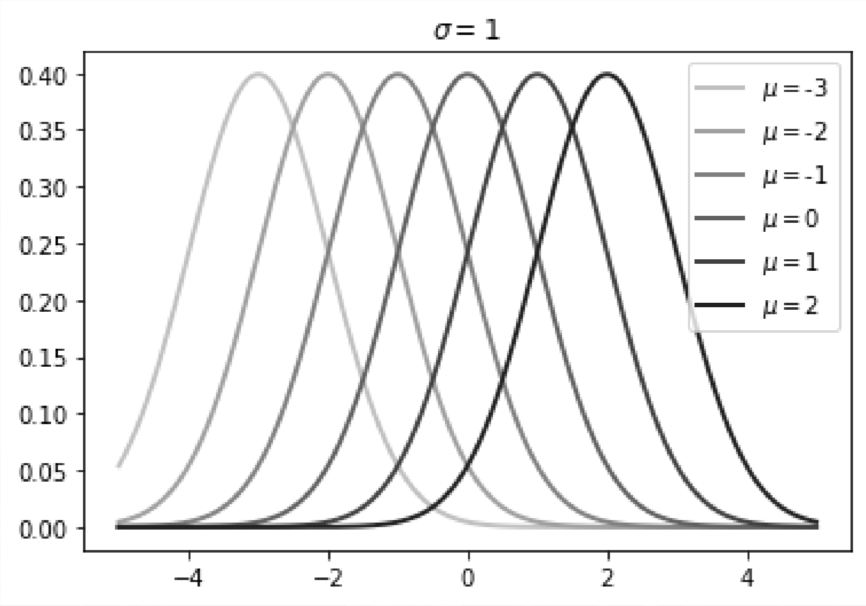


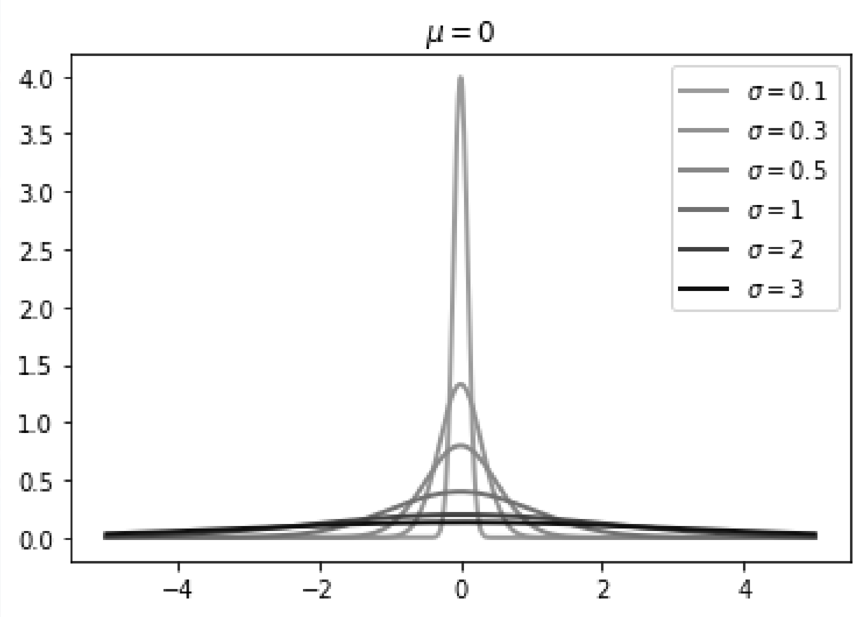
$$P(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi\,\sigma^2}}\,exp\left[-\frac{(x-\mu)^2}{2\,\sigma^2}\right]$$

Normal/Gauss distribution

- uniform
- binomial
- poissonian
- **normal (gaussian)**







- uniform
- binomial
- poissonian
- **normal (gaussian)**

$$\boxed{P(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi\,\sigma^2}} exp\left[-\frac{(x-\mu)^2}{2\,\sigma^2}\right]}$$

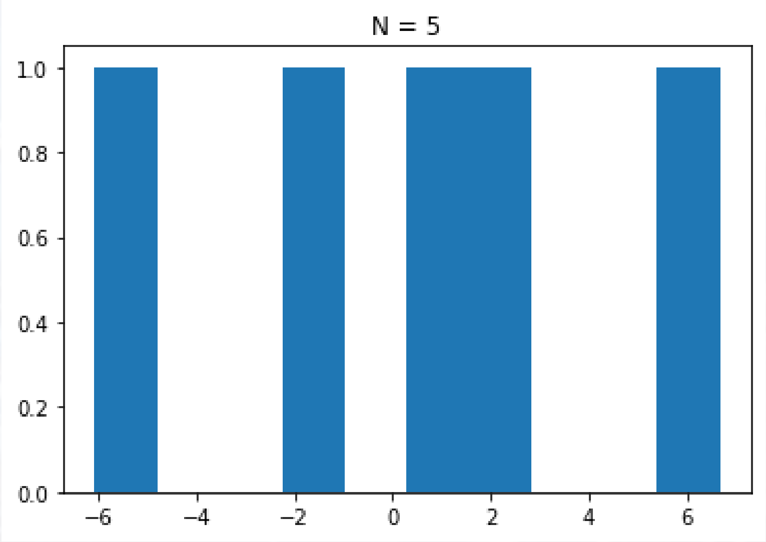
Normal/Gauss distribution

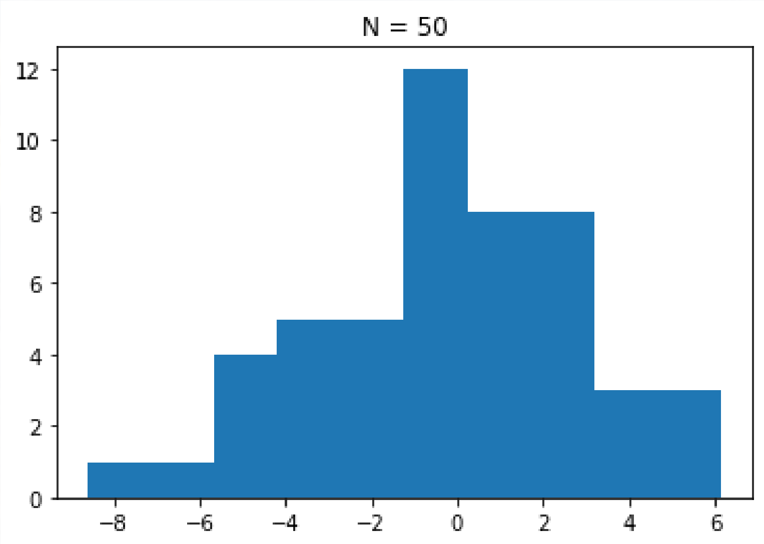
$$P(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi\,\sigma^2}} exp\left[-\frac{(x-\mu)^2}{2\,\sigma^2}\right]$$

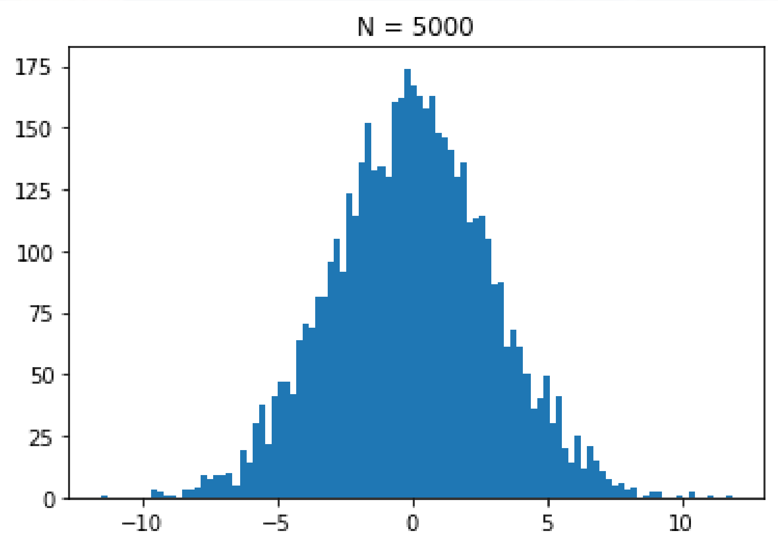
Normal/Gauss distribution

- uniform
- binomial
- poissonian
- **normal (gaussian)**

```
mu = 0
s  = 1
P  = np.random.normal(mu, s, N)
plt.hist(P)
plt.title('N = ' + str(N))
```







Thank you very much for your attention!