Introduction to Computer Science

Linear Algebra – Arrays

鄒年棣 (Nien-Ti Tsou)

Introduction

- Linear algebra is one of the essential building blocks of computational mathematics. The objects of linear algebra are vectors and matrices. The package `NumPy` includes all the necessary tools to manipulate those objects.
  - The first task is to build matrices and vectors, or to alter existing ones by slicing.
  - The other main task is the `dot` operation, which embodies most of the linear algebra operations (scalar product, matrix-vector product, and matrix-matrix product).
  - Finally, various methods are available to solve linear problems.

Vectors

- Creating vectors is as simple as using the function `array` to convert a list to an array:

```
import numpy as np
v = np.array([1.,2.,3.])
```

- The object `v` is now a vector. Here are some illustrations of the basic linear algebra operations on vectors:

```
v1 = np.array([1., 2., 3.])
v2 = np.array([2, 0, 1])
print(v1/2)
print(3*v1 + 2*v2)

print(np.dot(v1,v2))
print(v1 @ v2) # alternative formulation
print(np.cross(v1,v2))
```

v1		(3,)	[1. 2. 3.]
v2		(3,)	[2 0 1]

```
[0.5 1.  1.5]
[ 7.  6. 11.]
5.0
5.0
[ 2.  5. -4.]
```

Universal functions: elementwise

- Note that all basic arithmetic operations are performed elementwise. Thus, they have an output array that has the same shape as the input array.

v1	float64	(3,)	[1. 2. 3.]
v2	int64	(3,)	[2 0 1]

```
print(v1 * v2)
print(v2 / v1)
print(v1 - v2)
print(v1 + v2)
```

```
[         ]
[                                       ]
[-1.  2.  2.]
[3. 2. 4.]
```

- Some operators act elementwise on arrays as well:

```
print(np.cos(np.array([0, (np.pi)/2, np.pi])))
print(np.array([1, 2])**2)
print(2**np.array([1, 2]))
print(np.array([1, 2])**np.array([1, 2]))
```

```
[ 1.000000e+00  6.123234e-17 -1.000000e+00]
[1 4]
[2 4]
[    ]
```

Matrix

- A matrix is created in a similar way to a vector, but from a list of lists instead:

```
M = np.array([[1.,2],[0.,1]])
print(M)
```

```
[[1. 2.]
 [0. 1.]]
```

- Python allows a line break for array creation, which makes it more pleasing to the human eye:

```
M = np.array([[1., 2],
              [0., 1]])

print(M)
```

More readable!

```
[[1. 2.]
 [0. 1.]]
```

- The *n* vector, an *n* × 1, and a 1 × n matrix are three different objects even if they contain the same data.

```
V = np.array([1., 2., 1.])
R = np.array([[1.,2.,1.]])
print(V, np.shape(V), np.ndim(V))
print(R, np.shape(R), np.ndim(R))
```

```
[1. 2. 1.] (3,) 1
```

Tuple!

Array properties

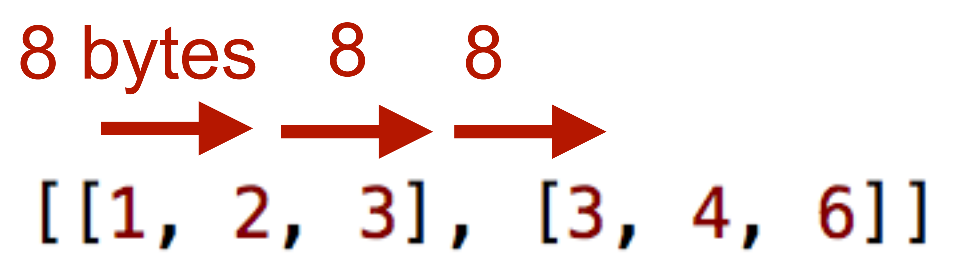
- Arrays are used to manipulate vectors, matrices, and more general tensors in NumPy.

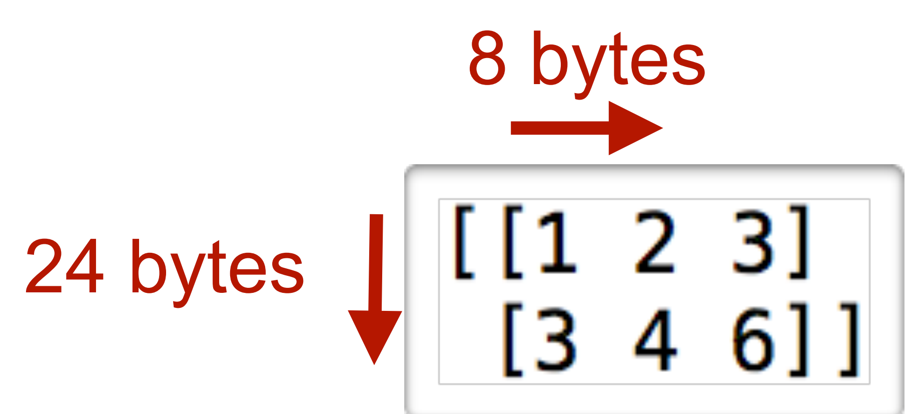
```
A = np.array([[1, 2, 3], [3, 4, 6]])
print(A.shape)
print(A.dtype)
print(A.strides)
```

```
int64
(24, 8)
```

- Type `'int64'` means they use 64 bits or 8 bytes in memory. The complete array is stored in memory row-wise.

- The distance from `A[0, 0]` to the first element in the next row `A[1,0]` is thus 24 bytes (three matrix elements) in memory. Correspondingly, the distance in memory between `A[0,0]` and `A[0,1]` is 8 bytes (one matrix element). These values are stored in the attribute `strides`.





Array types

- A proper array creation should be like:

```
Vf = np.array([1, 2, 1], dtype=float)
Vc = np.array([1, 2, 1], dtype=complex)
```

Vc	complex128	(3,)	[1.+0.j 2.+0.j 1.+0.j]
Vf	float64	(3,)	[1. 2. 1.]

- When no type is specified, the type is guessed. The `array` function chooses the type that allows storing of all the specified values:

```
V = np.array([1, 2])
print(V.dtype)
V = np.array([1., 2])
print(V.dtype)
V = np.array([1. + 0j, 2.])
print(V.dtype)
```

```
int64
float64
complex128
```

- Silent type conversion, which might give unexpected results:

```
a = np.array([1, 2, 3])
a[0] = 0.5
print(a)
```

Functions to construct arrays

- The usual way to set up an array is via a list.
- There are also a couple of convenient methods for generating special arrays.

```
r = np.random.rand(2,3)
print(r)
```

```
[[0.70173912 0.52768381 0.97475031]
 [0.48976793 0.87271617 0.29921848]]
[ 9.2 11.4 13.6 15.8 18.
[ 9.2 11.4 13.6 15.8 18.
```

```
a =   np.arange(9.2,20.2,2.2)
b = np.linspace(9.2,20.2,6)
print(a)
print(b)
```

Note!!! Tuple!

```
print(np.zeros((2,3)))
np.zeros(np.shape(r)) # Same as above
print(np.ones((2,3)))
print(np.identity(3))
```

```
[[0. 0. 0.]
 [0. 0. 0.]]
[[1. 1. 1.]
 [1. 1. 1.]]
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

```
x = np.diag([4,5,6]) # Generate
print(x)
print(np.diag(x)) # Get values
print(np.diag(x,1)) # Get values
print(np.diag(x,2)) # Get values
```

```
[[4 0 0]
 [0 5 0]
 [0 0 6]]
[       ]
[     ]
[  ]
```

Shaping of Array

Indexing and slices

- Array entries are accessed by indexes.
  - One pair of brackets:
  - Two pairs of brackets:

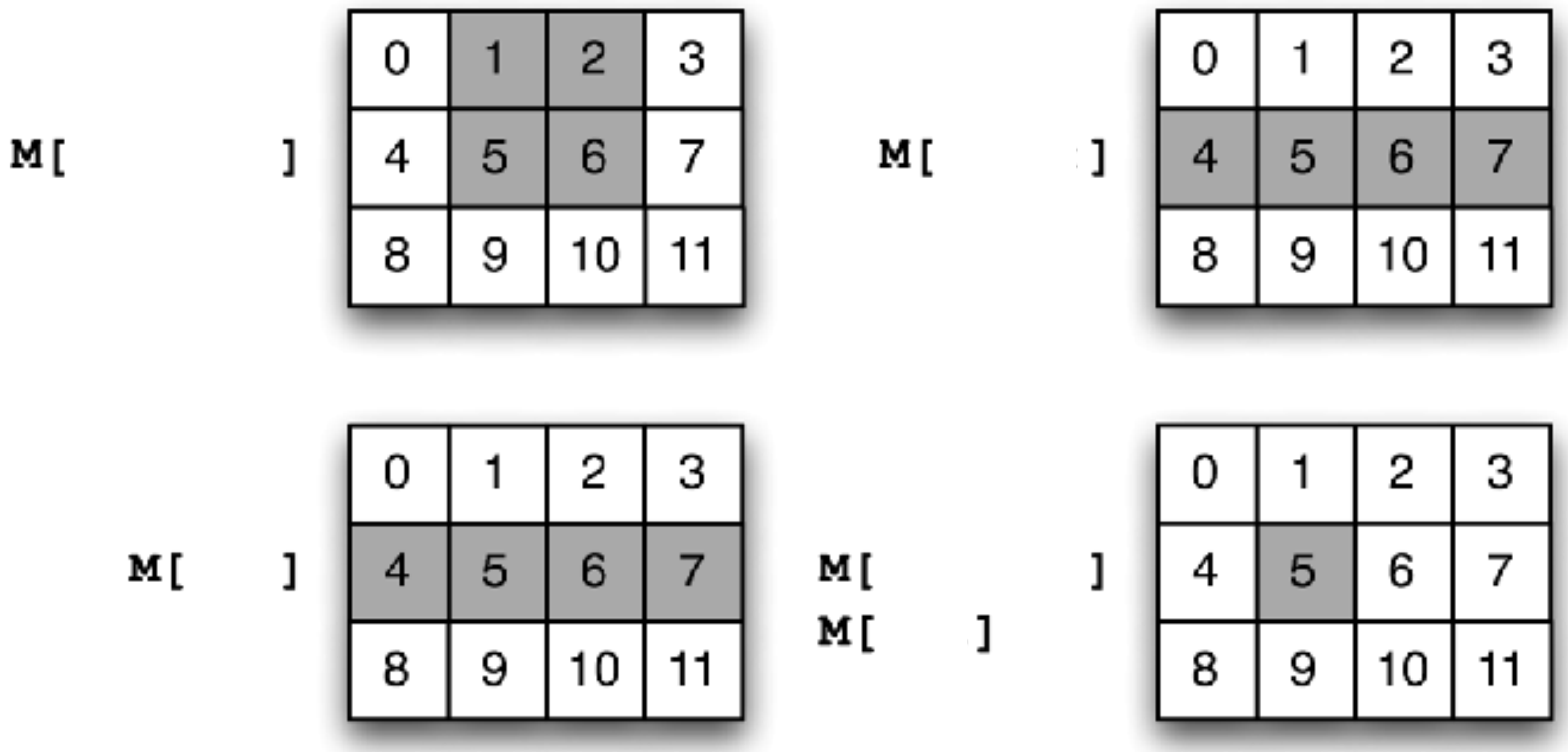
```
v = np.array([1., 2., 3])
M = np.array([[1., 2],[3., 4]])
print(v[0]) # works as for lists
print(v[1:])

print(M[0, 0])
print(M[1:]) # returns the matrix array
print(M[1]) # returns the vector array
```

```
1.0
[2. 3.]
1.0
```

```
v[0] = 10
print(v)
v[:2] = [0, 1]
print(v)
v[:2] = [1, 2, 3]
```

```
[10.  2.  3.]
```

Indexing and slices



- Omitting a dimension: If you omit an index or a slice, NumPy assumes you are taking rows only.

```
M = np.array([[0,1,2,3],[4,5,6,7],[8,9,10,11]])
print(M[2])
```

```
[ 8  9 10 11]
```

Altering an array using slices

```
M = np.array([[0,1.,2.],[3.,4.,5.],[6,7,8],[9,10,11]])
print(M)
M[1, 2] = 2.0
print(M)
M[2, :] = [1., 2., 3.]
print(M)
M[1:3, :] = np.array([[1., 2., 3.],[-1.,-2., -3.]])
print(M)
```

```
[[ 0.   1.   2.]
 [ 3.   4.   5.]
 [ 6.   7.   8.]
 [ 9.  10.  11.]]
[[             ]
 [             ]
 [             ]
 [             ]]
[[             ]
 [             ]
 [             ]
 [             ]]
[[             ]
 [             ]
 [             ]
 [             ]]
```

- There is a distinction between a column matrix and a vector.

```
M[1:4, 2:3] = np.array([[1.],[0.],[-1.0]])
print(M)
M[1:4, 2:3] = np.array([1., 0., -1.0])
```

```
[[             ]
 [             ]
 [             ]
 [             ]]
```

```
ValueError: could not broadcast input array
from shape (3) into shape (3,1)
```

Reshape

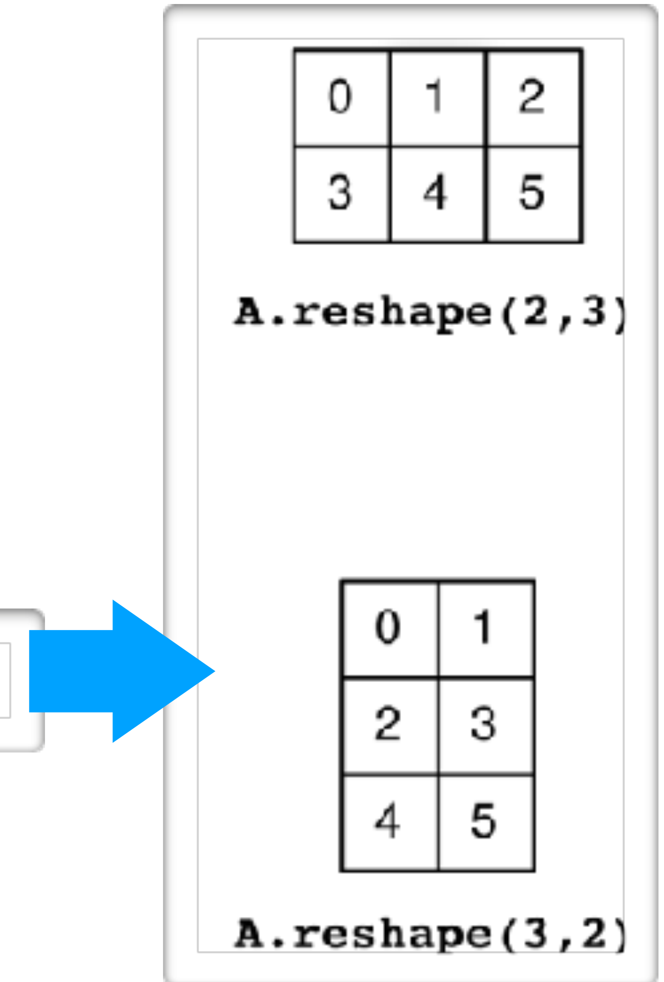
- Vector and row matrix can be viewed as a column matrix by the method `reshape`:

```
V = np.array([1., 2., 1.])
R = np.array([[1.,2.,1.]])
V2C = V.reshape(3, 1)
R2C = R.reshape(3, 1)
print(V2C,np.shape(V2C))
print(R2C,np.shape(R2C))
```

```
[[1.]
 [2.]
 [1.]] (3, 1)
[[1.]
 [2.]
 [1.]] (3, 1)
```

```
A = np.arange(6)
print(A)
```

```
[0 1 2 3 4 5]
```



- It is convenient to specify only one shape parameter by setting the free shape parameter `-1`:

```
v = np.array([1, 2, 3, 4, 5, 6, 7, 8])
M = v.reshape(2, -1)
print(M)
M = v.reshape(-1, 2)
print(M)
M = v.reshape(3,-1)
```

```
[[1 2 3 4]
 [5 6 7 8]]
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
```

```
ValueError: cannot reshape array of
size 8 into shape (3,newaxis)
```

Transpose $B_{ij} = A_{ji}$

- A special form of reshaping is transposing. It just switches the two shape elements of the matrix.

In previous page

M
```
[[1, 2],
 [3, 4],
 [5, 6],
 [7, 8]]
```

```
print(M.T)
```

```
[[1 3 5 7]
 [2 4 6 8]]
```

- Transpose does not copy! Transposition is very similar to reshaping and just returns a view on the same array.

- Transposing a vector makes no sense in NumPy:

```
v = np.array([1., 2., 3.])
print(v)
print(v.T)
```

```
[1. 2. 3.]
[1. 2. 3.]
```

- Transpose a vector is probably to create a row or column matrix. This is done using `reshape`:

```
print(v.reshape(-1, 1))
print(v.reshape(1, -1))
```

```
[[1.]
 [2.]
 [3.]]
[[1. 2. 3.]]
```

Array view

- Basic slices (`:`, `T`, `reshape`) return views. It is possible to access the object that owns the data using the array attribute `base`. If an array owns its data, the attribute base is `none`.

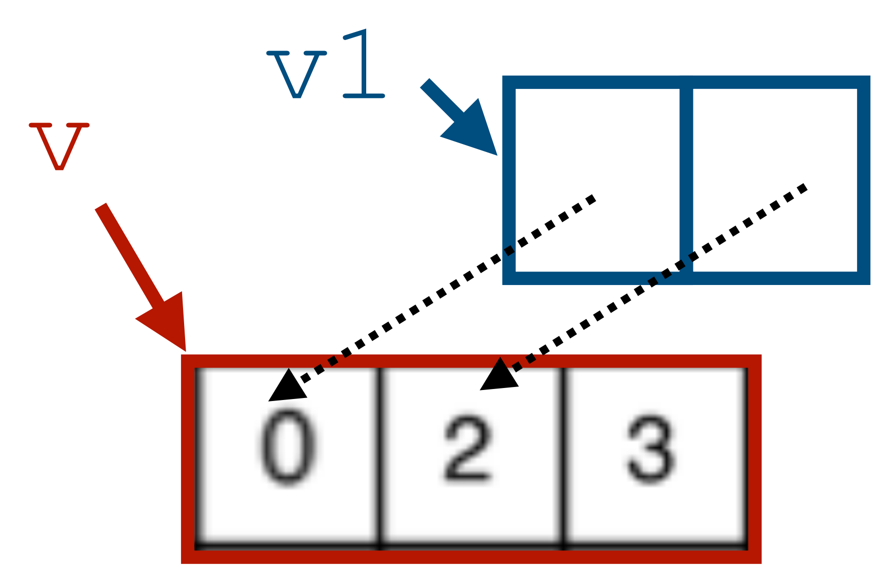
- Changing the elements of a slice of view affects the entire array

```python
v = np.array([1., 2., 3.])
print(v)
v1 = v[:2]
print(v1)
v1[0] = 0.
print(v)
print(v1.base is v)
print(v.base is None)
print(id(v),id(v1), v1 is v)
```

```
[1. 2. 3.]
[1. 2.]
[0. 2. 3.]


17692163088 26283434512
```

- We can say: the data pointed by `v1` is based on (or getting values from) the data pointed by `v`.



Array copy

- Sometimes it is necessary to explicitly request that the data be copied. This is simply achieved with the function `array`:

```
M = np.array([[1.,2.],[3.,4.]])
M2 = np.array(M) # copy of M
M3 = M
N = np.array(M.T) # copy of M.T
N2 = M.T
M[0,0]=10
print(M)
print(M2)
print(M3)
print(N)
print(N2)
```

Alternatively:

```
M2 = M.copy()
```

- We may verify that the data has indeed been copied:

```
print(M.base is None)
print(M2.base is None)
print(M3.base is M)
print(N.base is None)
print(N2.base is M)
```

```
True
True

True
True
```

Why???!!!

```
M4 = M[:]
print(M4.base is M)
```

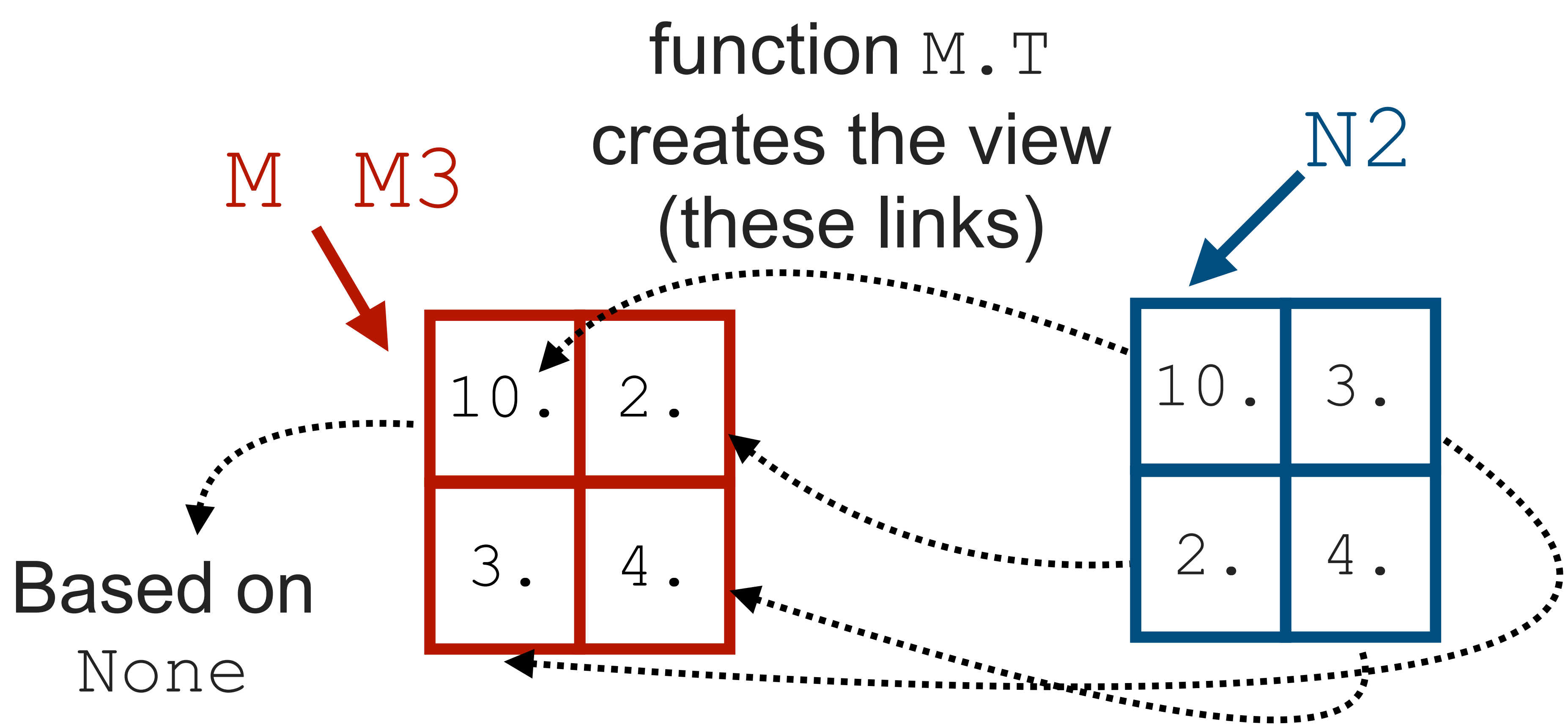
Why does `M3.base is M` return `False`?

```
print(id(M))
print(id(M.base))
print(id(M3))
print(id(M3.base))
print(M3 is M)
print(M3.base is M)
print(id(N2))
print(id(M.T))
print(id(M.T.base))
print(id(N2.base))
```

```
1949924536848
1564976272
1949924536848
1564976272
True
False
1949924565760
1949924534272
1949924536848
1949924536848
```

`M3 is M`

`M3` is not an object extended from `M`, so no base relationship between `M3` and `M`. On the other hand, `N2` gets its values from the memory pointed by `M`.



Thanks to
0611534 陳怡儒

Stacking

- The method to build matrices from a couple of (matching) submatrices is `concatenate`.
- This command stacks the submatrices vertically when `axis=0`. With the `axis=1`, they are stacked horizontally.
- `concatenate` creates a new matrix

```
Z = np.zeros((3,3))
I = np.identity(3)
print(Z)
print(I)
print(np.concatenate((Z,I)))
print(np.concatenate((Z,I),axis = 1))
X = np.concatenate((Z,I),axis = 1)
Z[0,0]=100
print(Z)
print(X)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[[0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 1.]]
[[100.   0.   0.]
 [  0.   0.   0.]
 [  0.   0.   0.]]
[[          1. 0. 0.]
 [          0. 1. 0.]
 [          0. 0. 1.]]
```

More stacking

`hstack`: Used to stack matrices horizontally

`vstack`: Used to stack matrices vertically

`column_stack`: Used to stack vectors in columns

- One may stack vectors row-wise or column-wise using `vstack` and `column_stack`.

```
a=np.array([[1],[2],[3]])
aaa = np.hstack((a,a,a))
a[0,0] = 10
print(a)
print(aaa)
```

They can be () or []

```
[[10]
 [ 2]
 [ 3]]
[[1 1 1]
 [2 2 2]
 [3 3 3]]
```

```
v1 = np.array([1,2])
v2 = np.array([3,4])
vv = np.vstack([v1,v2])
print(vv)
v1v2 = np.column_stack([v1,v2])
print(v1v2)
```

vstack([v1,v2])

column_stack([v1,v2])

Operation on Arrays

Array functions

- Some functions acting on the whole matrix, row-wise, or column-wise, such as `max`, `min`, and `sum`.

```
A = (np.arange(1,9)).reshape(2,-1)
print(A)
print(np.sum(A),np.min(A),np.max(A))
```

```
[[1 2 3 4]
 [5 6 7 8]]
36 1 8
```

- Command `sum` has an optional parameter, `axis` . It allows us to choose along which axis to perform the operation.

```
print(np.sum(A, axis=0))
print(np.sum(A, axis=1))
```

```
[ 6  8 10 12]
[10 26]
```

Comparing arrays

- Comparing two arrays is not as simple as it may seem. Let's check whether two matrices are close to each other:

```
A = np.array([0.,0.])
B = np.array([0.,0.])
if abs(B-A) < 1e-10:
    print("The two arrays are close enough")
```

```
ValueError: The truth value of an array
with more than one element is ambiguous.
Use a.any() or a.all()
```

- The reason why the if statement lead error message is the output of that comparison return a boolean array. It is simply an array for which the entries have the type `bool`. For example:

```
TEST = np.array([True,False])
print(TEST.dtype)
```

```
bool
```

Boolean array

- Any comparison operator acting on arrays will create a Boolean array instead of a simple Boolean

```
M = np.array([[2, 3],[1, 4]])
print(M > 2)
print(M == 0)
N = np.array([[2, 3],[0, 0]])
print(M == N)
```

```
[[False  True]
 [False  True]]
[
                ]
[
                ]
```

- So what is the outcome of the follows?

```
print(abs(B-A) < 1e-10)
```

Functions `all` and `any`

- Therefore, one cannot use array comparison directly in conditional statements, for example, `if` statements. the solution is to use the methods `all` and `any`

```
A = np.array([[1,2],[3,4]])
B = np.array([[1,2],[3,3]])
print(A == B)
print((A == B).all())
print((A != B).any())
if (abs(B-A) < 1e-10).all():
    print("The two arrays are close enough")
```

```
[[ True  True]
 [ True False]]
```

Checking for equality

- Note that two floats may be very close without being equal. In NumPy, it is possible to check for equality by function `allclose` with a given precision.

```
data = np.random.rand(2)*1e-3
small_error = np.random.rand(2)*1e-8
data_err = data + small_error
print(data)
print(data_err)
print(data == data_err)
print(np.allclose(data, data_err, rtol=1.e-5, atol=1.e-8))
```

```
[0.00076381 0.00017465]
[0.00076382 0.00017465]
[False False]
True
```

- The tolerance is given in terms of a relative tolerance bound, `rtol`, and an absolute error bound, `atol`. The command `allclose` is a short form of:

```
(abs(A-B) < atol+rtol*abs(B)).all()
```

- Note that `allclose` can be also applied to scalars.

Boolean operations on arrays

- You cannot use `and`, `or`, and `not` on Boolean arrays. Instead, we can use these operators for componentwise logical operations on Boolean arrays:

Logic operator	**Replacement for Boolean arrays**
`A and B`	`A & B`
`A or B`	`A | B`
`not A`	`~ A`

```
A = np.array([True, True, False, False])
B = np.array([True, False, True, False])
print(A & B)
print(A | B)
print(~A)
print(A and B)
```

```
[                                    ]
[                                    ]
[                                    ]
```

```
ValueError: The truth value of an
array with more than one element is
ambiguous. Use a.any() or a.all()
```

Indexing with Boolean arrays

- It is often useful to access and modify only parts of an array, depending on its value. For instance, one might want to access all the positive elements of an array. This turns out to be possible using Boolean arrays, which act like masks to select only some elements of an array.

- The result of such an indexing is always a vector.

```
B = np.array([[True, False],[False, True]])
M = np.array([[2, 3],[1, 4]])
print(M[B])
M[B] = 0
print(M)
M[B] = 10, 20
print(M)
M[M>2] = 0
print(M)
```

When will we use Boolean array Indexing?

- Now we have a sequence of data with some measurement error. Suppose further that we run a regression and it gives us a deviation for each value. We wish to obtain

1. all the exceptional values with absolute error > 0.5
2. all the good values which are lower than 5.0

```
data = np.linspace(1,10,10) # data
error = np.random.rand(10) # the errors
exceptional = data[abs(error) > 0.5]
small = data[(abs(error)<= 0.5) & (data<5.)]
print(data)
print(error)
print(exceptional)
print(small)
```

```
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
[0.03212591 0.87259363 0.21471738 0.79291333 0.33642996 0.2386707
 0.13483616 0.73206135 0.3556771  0.90487261]
[ 2.  4.  8. 10.]
[1. 3.]
```

Using `where`

- The command `where(condition, a, b)` can take a Boolean array as a condition. This will return values from `a` when the condition is `True` and values from `b` when it is `False`.

- For instances consider, a Heaviside function:

```
def H(x):
    return np.where(x < 0, 0, 1)

x = np.linspace(-1,1,11)
print(x)
print(H(x))
```

$$H(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$$

```
[-1.  -0.8 -0.6 -0.4 -0.2  0.   0.2  0.4  0.6  0.8  1. ]
[0 0 0 0 0 1 1 1 1 1 1]
```

Using `where`

- More example to demonstrated how to manipulate elements from an array or a scalar depending on a condition:

```
x = np.linspace(-4,4,5)
print(x)
print(np.where(x > 0, x**2+3, 0))
print(np.where(x > 0, 1, -1))
```

```
[-4. -2.  0.  2.  4.]
[ 0.  0.  0.  7. 19.]
[-1 -1 -1  1  1]
```

- If the second and third arguments are omitted, then a tuple containing the indexes of the elements satisfying the condition is returned.

```
a = np.linspace(-4,4,9)
b = a.reshape((3,3))
print(a)
print(np.where(a > 0 ))
print(b)
print(np.where(b > 0))
```

```
[-4. -3. -2. -1.  0.  1.  2.  3.  4.]

[[-4. -3. -2.]
 [-1.  0.  1.]
 [ 2.  3.  4.]]
(array([1, 2, 2, 2]), array([                 ]))
```

Broadcasting

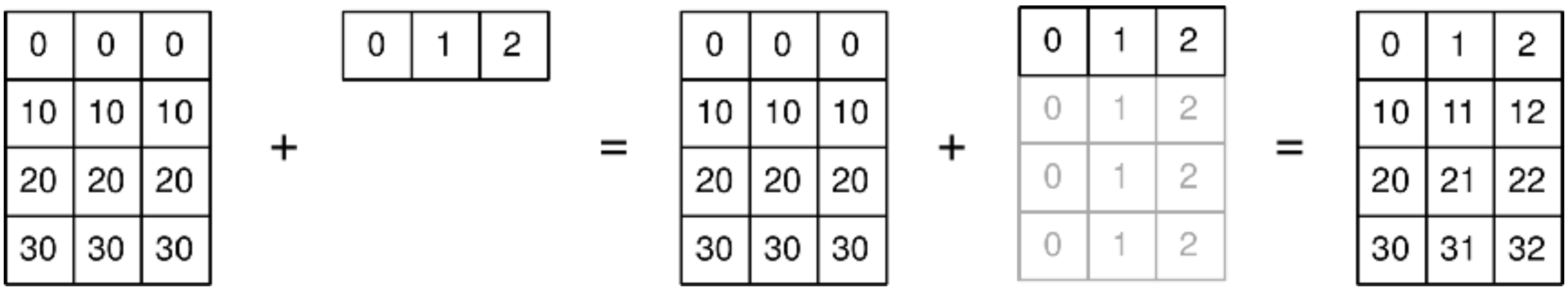
- Broadcasting in NumPy denotes the ability to guess a common, compatible shape between two arrays.

```
vector = np.arange(4)
vector2 = vector + 1.
vector[0]=3.3
print(vector)
print(vector2)
```

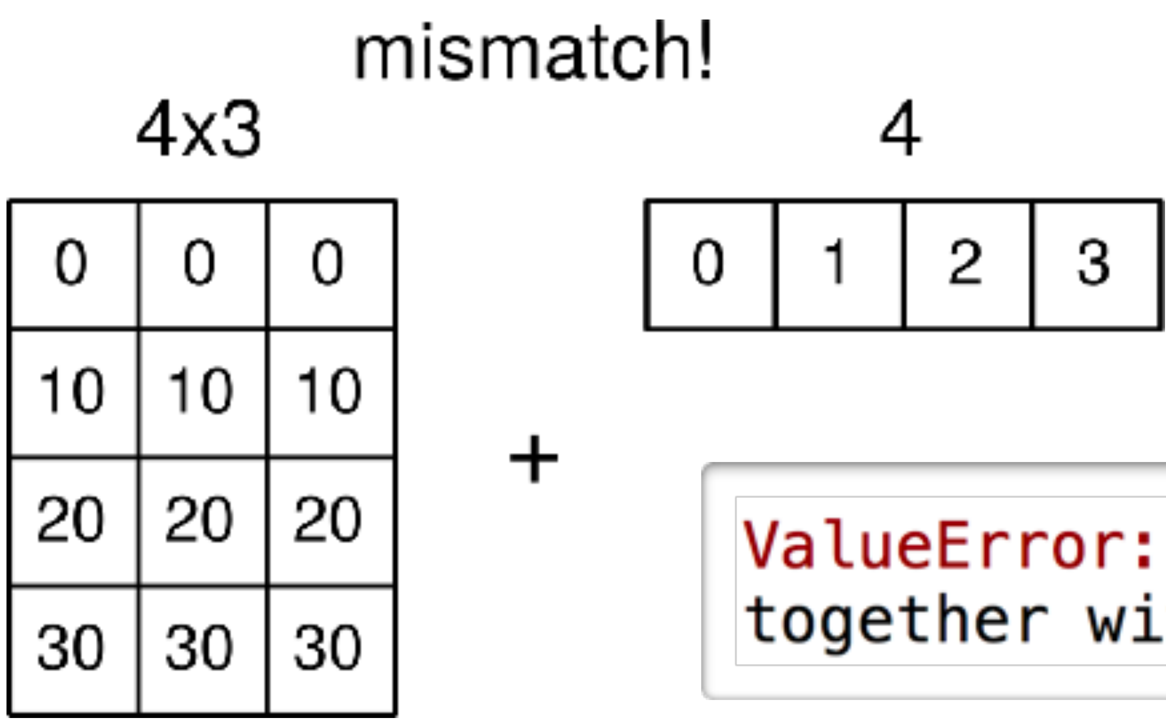
- In this example, everything happens as if the scalar `1.` had been converted to an array of the same length as `vector`, that is, `array([1.,1.,1.,1.])`, and then added to `vector`.

- ```
  C = np.arange(1,3).reshape(-1,1) # column
  R = np.arange(2).reshape(1,-1) # row
  print(C)
  print(R)
  print(C + R )
  ```
  ```

Quiz



Shape mismatch




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

Solve a Linear Algebra by Python

Linear algebra operations

- The essential operator that performs most of the usual operations of linear algebra is the Python function `dot`. It is used for computing matrix-vector multiplications, a scalar product between two vectors, matrix-matrix products.

```python
A = np.array([[1.,2,3],[1,1,3],[1,2,4]])
b = np.array([10.,8,13])
M = np.array([[1.,2,3],[4,5,6],[7,8,9]])
print(A)
print(b)
print(np.dot(A, b)) # Matrix dot vector
print(np.dot(A, M)) # Matrix dot Matrix
```

```
[[1. 2. 3.]
 [1. 1. 3.]
 [1. 2. 4.]]
[10.  8. 13.]
[65. 57. 78.]
[[30. 36. 42.]
 [26. 31. 36.]
 [37. 44. 51.]]
```

Solving a linear system

- If *A* is a matrix and *b* is a vector, you can solve the linear equation: $$Ax = b$$
- Using the `solve` method in scipy linear algebra

In previous page, A and b

```
[[1. 2. 3.]
 [1. 1. 3.]
 [1. 2. 4.]]
[10.  8. 13.]
```

```python
import scipy.linalg as sl

x = sl.solve(A, b)
print(x)

print(np.dot(A, x)-b, sl.norm(np.dot(A, x)-b))
print(sl.norm(np.dot(A, x)-b)<1e-6) # old school
print(np.allclose(np.dot(A, x), b))
```

```
[-3.  2.  3.]
[0. 0. 0.] 0.0
True
True
```

- The command `norm` is the positive length to each vector.

$$\|\boldsymbol{x}\|_2 := \sqrt{x_1^2 + \cdots + x_n^2}.$$

- The command `allclose` is used here to compare two vectors. If they are close enough to each other, this command returns `True`. The default tolerance value is 10^{-8}.

More methods in `scipy.linalg`

Methods	**Description (matrix methods)**
`sl.det`	Determinant of a matrix
`sl.eig`	Eigenvalues and eigenvectors of a matrix
`sl.inv`	Matrix inverse
`sl.pinv`	Matrix pseudoinverse
`sl.norm`	Matrix or vector norm
`sl.svd`	Singular value decomposition
`sl.lu`	LU decomposition
`sl.qr`	QR decomposition
`sl.cholesky`	Cholesky decomposition
`sl.solve`	Solution of a general or symmetric linear system: $Ax = b$
`sl.solve.banded`	The same for banded matrices
`sl.lstsq`	Least squares solution

Methods: Examples

```
A = np.array([[1.,2,3],[1,1,3],[1,2,4]])
b = np.array([10.,8,13])

InvA = sl.inv(A)
print(InvA)
print(np.dot(InvA,b))
print(sl.solve(A, b))

print(sl.det(A))
(eigVal, eigVec) = sl.eig(A)
print(eigVal)
print(eigVec)
```

$$\begin{cases} x+2y+3z=10 \cdots (1) \\ x+\ \ y+3z=8 \ \cdots (2) \\ x+2y+4z=13 \cdots (3) \end{cases}$$

```
[[ 2.   2. -3.]
 [ 1. -1.  0.]
 [-1.  0.  1.]]
[-3.  2.  3.]
[-3.  2.  3.]
```

```
-0.9999999999999998
[ 6.29257994+0.j  0.2783488 +0.j -0.57092874+0.j]
[[-0.56270964 -0.91432584  0.58695504]
 [-0.4855477  -0.19908612 -0.78101129]
 [-0.66903012  0.35266553  0.21331935]]
```

Methods: Examples

- `inv()` and `solve()` can only be applied to a square matrix. `pinv()`, however, can determined the inverse matrix numerically.

```
A = np.array([[1.,2,-1],[0,-5,3]])
b = np.array([4.,1])
InvA = sl.pinv(A)
print(InvA)
x_pinv = np.dot(InvA,b)
print(x_pinv)
print(sl.norm(np.dot(A,x_pinv)-b)<1e-6)
```

$$\begin{bmatrix} 1 & 2 & -1 \\ 0 & -5 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 1 \end{bmatrix}$$

```
[[ 0.97142857  0.37142857]
 [ 0.08571429 -0.11428571]
 [ 0.14285714  0.14285714]]
[4.25714286 0.22857143 0.71428571]
True
```

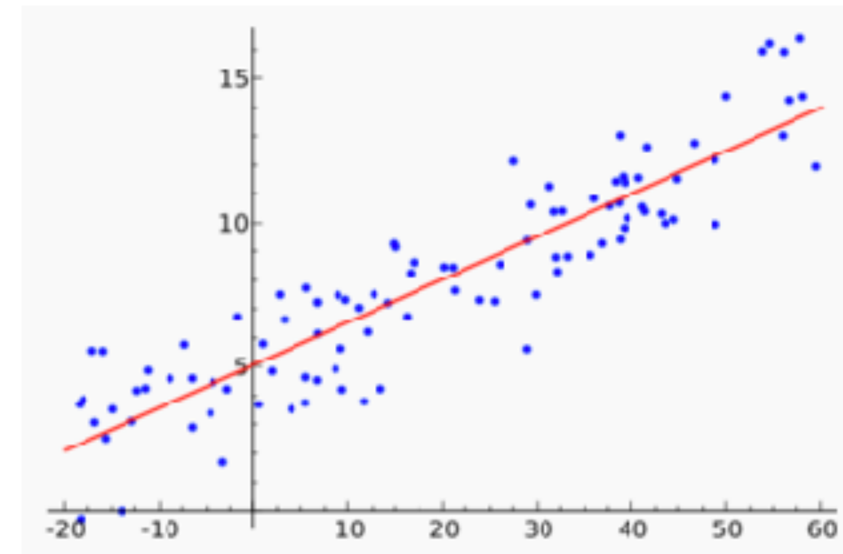
```
InvA = sl.inv(A)
x = sl.solve(A, b)
```

```
ValueError: expected square matrix
```

```
ValueError: Input a needs to be a square matrix.
```

Methods: Examples

- The method of least squares is a standard approach in regression analysis to approximate the solution.
- “Least squares" means that the overall solution minimizes the sum of the squares of the residuals made in the results of every single equation. i.e. that minimizes the Euclidean 2-norm || *b* - *a x* ||^2




```
(x_lstsq, r, rank, s) = sl.lstsq(A, b)
print(x_lstsq)
print(sl.norm(np.dot(A,x_pinv)-b)<1e-6)
print(r, rank, s)
```

$$\begin{bmatrix} 1 & 2 & -1 \\ 0 & -5 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 1 \end{bmatrix}$$

The solution `x`

```
[4.25714286 0.22857143 0.71428571]
True
[] 2 [6.25339693 0.94605857]
```

The residues of `b`

The rank of `A`

The singular value of `A`