Heuristics

Lab Section 5

For students with Programming background

- Go through codes: differencebetweenmeans_sig.py and differencebetweenmeans_conf.py
  - Ask anything code related if you don’t understand it
  - Run the codes with Diff2Mean.vals input file
  - Send me code/output screenshots of the above run
- Take input from the knap file which contains:
  - Continue on the next slide ON THE NEXT SLIDE

For students with Programming background

- Take input from the knap file which contains
  - First line as the total weight of all items
  - Rest of the lines as Items with weight as the first attribute and value as the second attribute
  - Read these attributes, take the weight integer from the user, and find the best items that approximately make that weight meaning having almost that weight those items have the maximum value of all the possible items
  - For example if we have items with weight and values = { 5: 10, 4: 20, 20: 100, 10:25} , now if the user gives a weight of 30, the program will select the best items with maximum value: {5: 10, 4:20, 20:100}

Today's Lab

- We will explore these exercises and concepts:
- quidditch.py
- differencebetweenmeans_sig.py
- differencebetweenmeans_conf.py
- Numpy library installation
- Numpy overview with indexing exercises
- Numpyexample.py
- Introduction to Knapsack problem

What is Numpy?

- Numpy, Scipy, and Matplotlib provide MATLAB-like functionality in python.
- Numpy Features:
  - Typed multidimentional arrays (matrices)
  - Fast numerical computations (matrix math)
  - High-level math functions

Why do we need NumPy

- Python does numerical computations slowly.
- 1000 x 1000 matrix multiply
  - Python triple loop takes > 10 min.
  - Numpy takes ~0.03 seconds

NumPy Overview

1. Arrays
2. Shaping and transposition
3. Mathematical Operations
4. Indexing and slicing
5. Broadcasting

Arrays

Structured lists of numbers.

- Vectors
- Matrices
- Images
- Tensors
- ConvNets

Arrays

Structured lists of numbers.

- **Vectors**
- **Matrices**
- Images
- Tensors
- ConvNets

$$\begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}$$

Arrays, Basic Properties

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]],dtype=np.float32)
print a.ndim, a.shape, a.dtype
```

1. Arrays can have any number of dimensions, including zero (a scalar).
2. Arrays are typed: np.uint8, np.int64, np.float32, np.float64
3. Arrays are dense. Each element of the array exists and has the same type.

Arrays, creation

- np.ones, np.zeros
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

Arrays, creation

- **np.ones, np.zeros**
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> np.ones((3,5),dtype=np.float32)
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]], dtype=float32)
```

```
>>> np.zeros((6,2),dtype=np.int8)
array([[0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0]], dtype=int8)
```

Arrays, creation

- np.ones, np.zeros
- **np.arange**
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> np.arange(1334,1338)
array([1334, 1335, 1336, 1337])
```

Arrays, creation

- np.ones, np.zeros
- np.arange
- **np.concatenate**
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> A = np.ones((2,3))
>>> B = np.zeros((4,3))
>>> np.concatenate([A,B])
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>>
```

Arrays, creation

- np.ones, np.zeros
- np.arange
- **np.concatenate**
- np.astype
- np.zeros_like, np.ones_lik
- np.random.random

```
>>> A = np.ones((4,1))
>>> B = np.zeros((4,2))
>>> np.concatenate([A,B], axis=1)
array([[ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.]])
```

Arrays, creation

- np.ones, np.zeros
- np.arange
- np.concatenate
- **np.astype**
- np.zeros_like, np.ones_l
- np.random.random

```
>>> A
array([[ 4670.5,  4670.5,  4670.5],
       [ 4670.5,  4670.5,  4670.5],
       [ 4670.5,  4670.5,  4670.5],
       [ 4670.5,  4670.5,  4670.5],
       [ 4670.5,  4670.5,  4670.5]], dtype=float32)
>>> print(A.astype(np.uint16))
[[4670 4670 4670]
 [4670 4670 4670]
 [4670 4670 4670]
 [4670 4670 4670]
 [4670 4670 4670]]
```

Arrays, creation

- np.ones, np.zeros
- np.arange
- np.concatenate
- np.astype
- **np.zeros_like, np.ones_like**
- np.random.random

```
>>> a = np.ones((2,2,3))
>>> b = np.zeros_like(a)
>>> print(b.shape)
```

Arrays, creation

- np.ones, np.zeros
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- **np.random.random**

```
>>> np.random.random((10,3))
array([[ 0.61481644,  0.55453657,  0.04320502],
       [ 0.08973085,  0.25959573,  0.27566721],
       [ 0.84375899,  0.2949532 ,  0.29712833],
       [ 0.44564992,  0.37728361,  0.29471536],
       [ 0.71256698,  0.53193976,  0.63061914],
       [ 0.03738061,  0.96497761,  0.01481647],
       [ 0.09924332,  0.73128868,  0.22521644],
       [ 0.94249399,  0.72355378,  0.94034095],
       [ 0.35742243,  0.91085299,  0.15669063],
       [ 0.54259617,  0.85891392,  0.77224443]])
```

Arrays, danger zone

- Must be dense, no holes.
- Must be one type
- Cannot combine arrays of different shape

```
>>> np.ones([7,8]) + np.ones([9,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together
 with shapes (7,8) (9,3)
```

Shaping

```
a = np.array([1,2,3,4,5,6])
a = a.reshape(3,2)
a = a.reshape(2,-1)
a = a.ravel()
```

1. Total number of elements cannot change.
2. Use -1 to infer axis shape
3. Row-major by default (MATLAB is column-major)

Return values

- Numpy functions return either **views** or **copies**.
- Views share data with the original array, like references in Java/C++. Altering entries of a view, changes the same entries in the original.
- The numpy documentation says which functions return views or copies
- Np.copy, np.view make explicit copies and views.

Saving and loading arrays

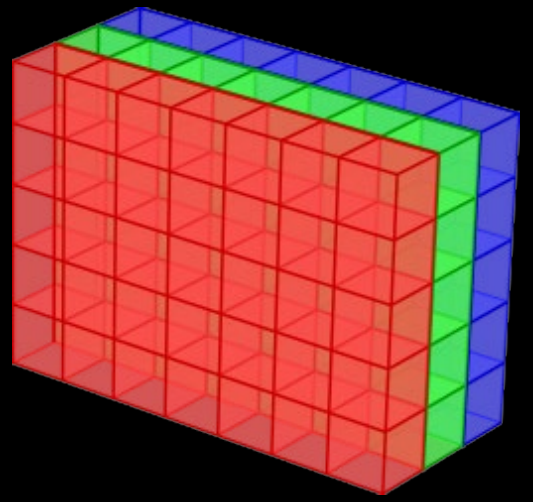
```
np.savez('data.npz', a=a)
data = np.load('data.npz')
a = data['a']
```

1. NPZ files can hold multiple arrays
2. np.savez_compressed similar.

A wish list

- we want to work with vectors and matrices

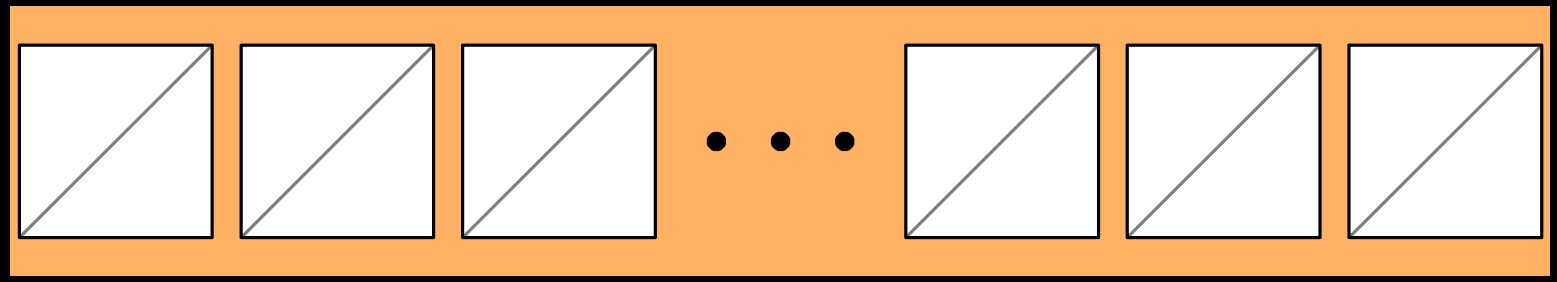
$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$



colour image as $N \times M \times 3$-array

- we want our code to run fast
- we want support for linear algebra
- ...

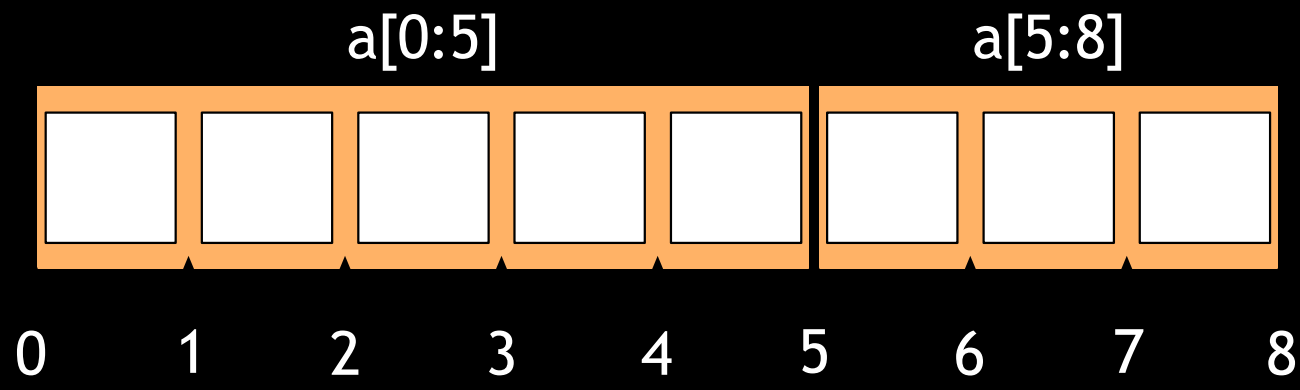
List indexing



- indexing starts at 0
- negative indices count from the end of the list to the beginning

List slicing

basic syntax: `[start:stop:step]`

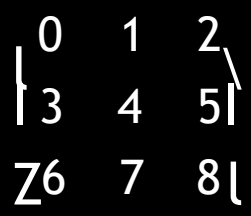


- if `step=1`
  - slice contains the elements `start` to `stop-1`
  - slice contains `stop-start` elements
- `start`, `stop`, and also `step` can be negative
- default values:
  - `start` 0, i.e. starting from the first element
  - `stop` N, i.e up to and including the last element
  - `step` 1

Let’s do some slicing

Matrices and lists of lists

Can we use lists of lists to work with matrices?

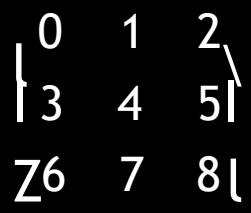



```
matrix = [[0, 1, 2],
          [3, 4, 5],
          [6, 7, 8]]
```

- How can we extract a row?
- How can we extract a column?

Matrices and lists of lists

Can we use lists of lists to work with matrices?



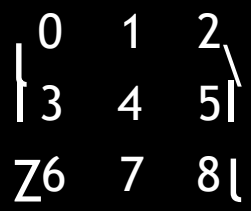

```
matrix = [[0, 1, 2],
          [3, 4, 5],
          [6, 7, 8]]
```

- How can we extract a row?
- How can we extract a column?

Let's do some experiments

Matrices and lists of lists

Can we use lists of lists to work with matrices?




```
matrix = [[0, 1, 2],
          [3, 4, 5],
          [6, 7, 8]]
```

- How can we extract a row? ☺
- How can we extract a column? ☹

Lists of lists do not work like matrices

Problems with lists as matrices

- different axes are not treated on equal footing
- lists can contain arbitrary objects
  matrices have a homogeneous structure
- list elements can be scattered in memory

Applied to matrices ...

...lists are conceptually inappropriate

...lists have less performance than possible

We need a new object

ndarray

multidimensional, homogeneous array of fixed-size items

Getting started

Install it with:

pip install numpy

pip3 install numpy

Import the NumPy package:

```
import numpy as np
```

Data types

Some important data types:

integer	`int8`, `int16`, `int32`, `int64`, `uint8`, ...
float	`float16`, `float32`, `float64`, ...
complex	`complex64`, `complex128`, ...
boolean	`bool8`
Unicode string	

default type: `float64`

Views

For the sake of efficiency, NumPy uses views if possible.

- Changing one or more matrix elements will change it in all views.
- Example: transposition of a matrix `a.T`
  No need to copy the matrix and to create a new one

Some array creation routines –revision

- numerical ranges: `arange`, `linspace`, `logspace`
- homogeneous data: `zeros`, `ones`
- diagonal elements: `diag`, `eye`
- random numbers: `rand`, `randint`

Numpy has an `append()`-method. Avoid it if possible.

Indexing and slicing in one dimension

1d arrays: indexing and slicing as for lists

- first element has index 0
- negative indices count from the end
- slices: `[start:stop:step]`
  without the element indexed by `stop`
- if values are omitted:
  - `start`: starting from first element
  - `stop`: until (and including) the last element
  - `step`: all elements between `start` and `stop`-1

Indexing and slicing in higher dimensions

- usual slicing syntax
- difference to lists:
  slices for the various axes separated by comma

a[2, -3]

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39

Indexing and slicing in higher dimensions

- usual slicing syntax
- difference to lists:
  slices for the various axes separated by comma

a[:3, :5]

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39

Indexing and slicing in higher dimensions

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39

Indexing and slicing in higher dimensions

a[-3:, -3:]

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39

Indexing and slicing in higher dimensions

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39

Indexing and slicing in higher dimensions

a[:, 3]

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39

Indexing and slicing in higher dimensions

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39

Indexing and slicing in higher dimensions

a[1, 3:6]

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39

Indexing and slicing in higher dimensions

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39

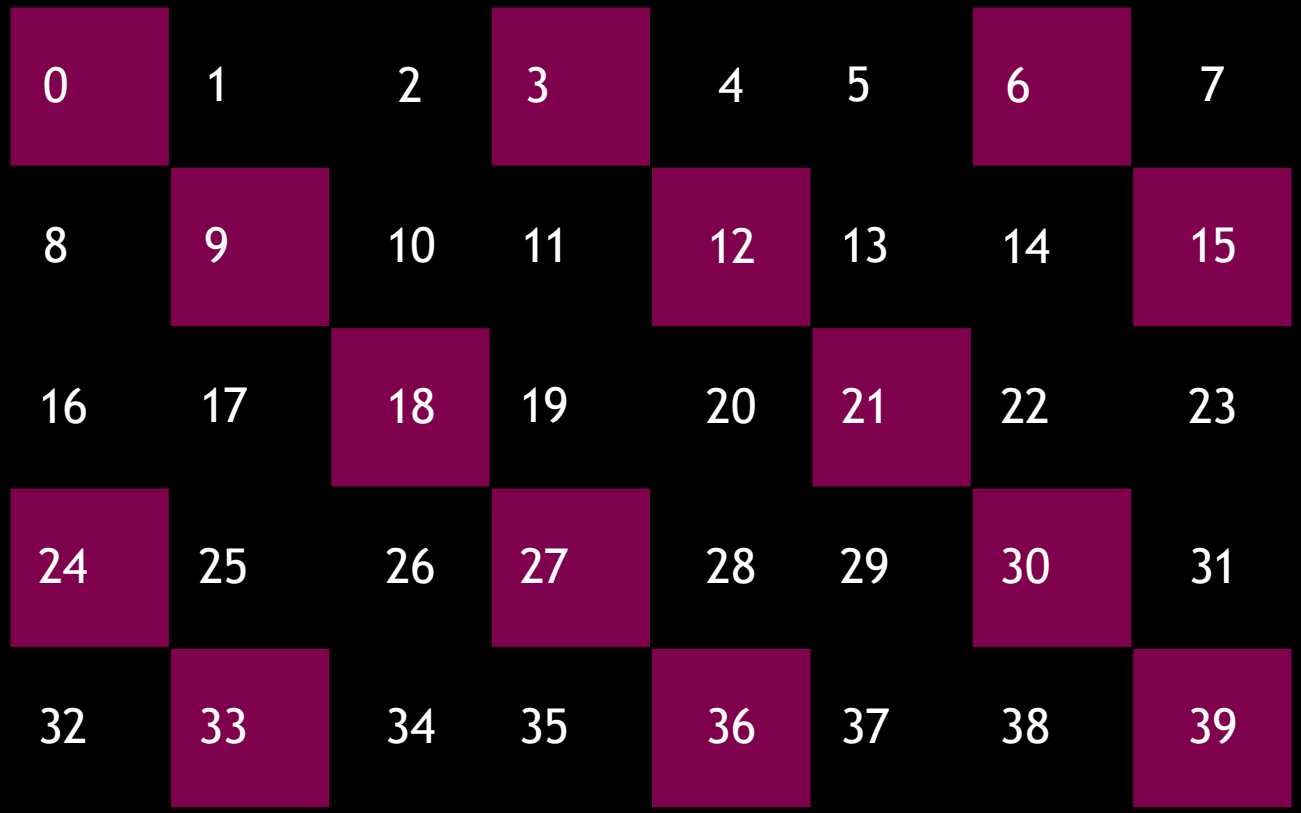
Indexing and slicing in higher dimensions

a[1::2, ::3]

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39

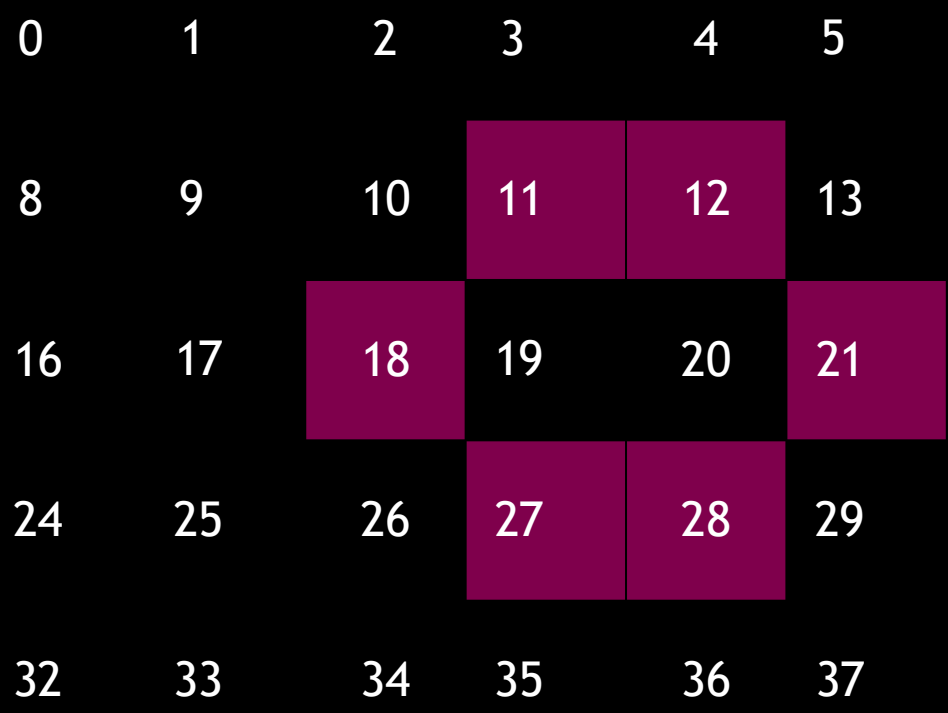
Fancy indexing – Boolean mask

a[a % 3 == 0]

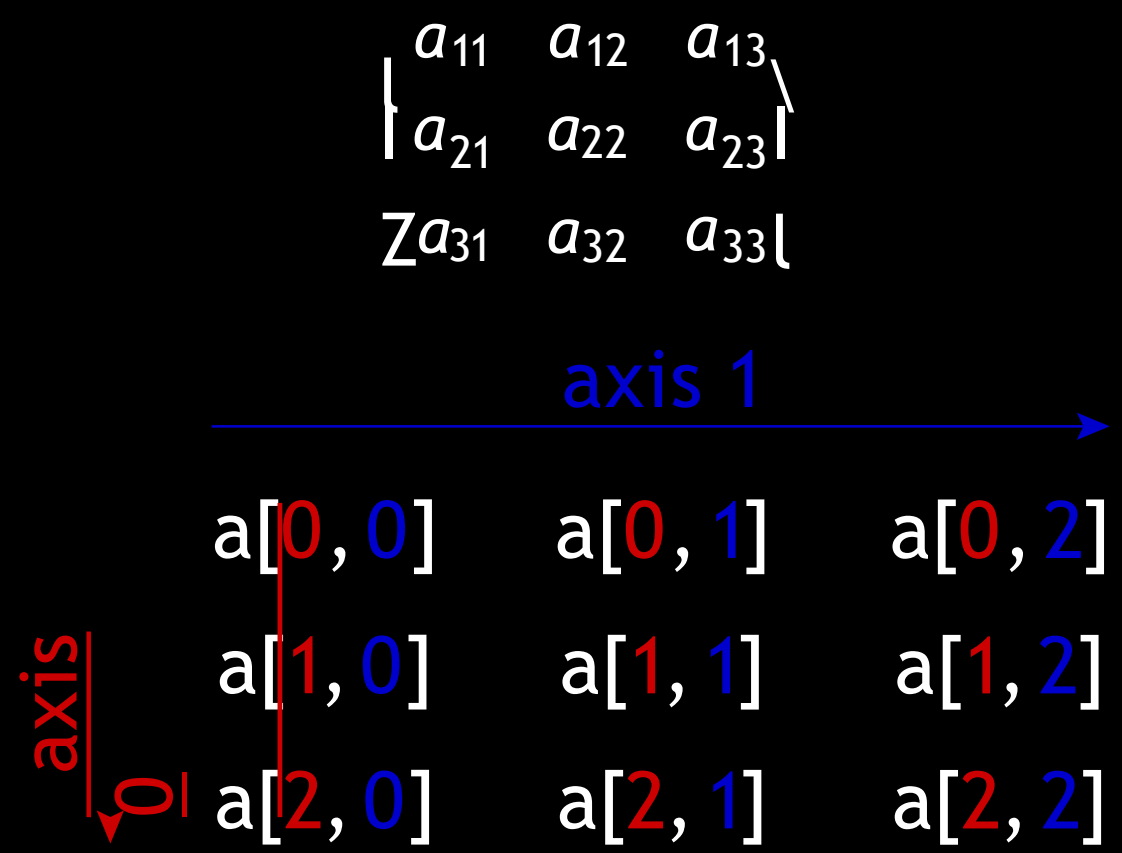


Fancy indexing – array of integers

a[(1, 1, 2, 2, 3, 3), (3, 4, 2, 5, 3, 4)]

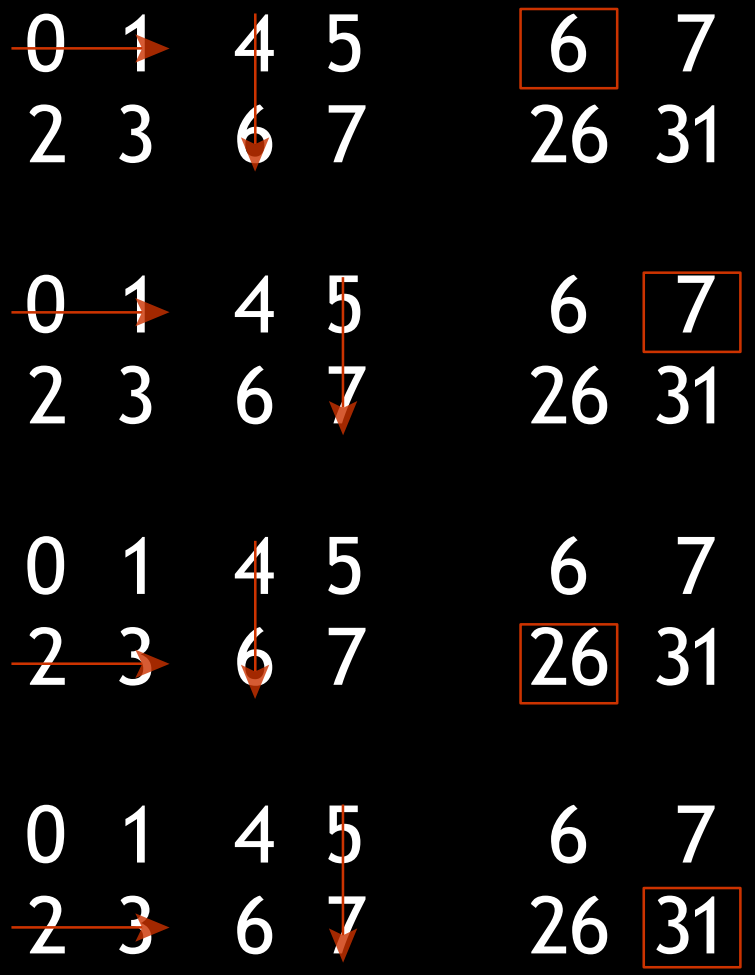


Axes




```
np.sum(a)
np.sum(a, axis=...)
```

Matrix multiplication



try `np.dot(•, •)`

`•.dot(•)`

`• @ •` *)

*) Python ≥3.5, NumPy ≥1.10

Mathematical functions in NumPy

Universal functions (ufuncs) take ndarrays as argument

Trigonometric functions

sin, cos, tan, arcsin, arccos, arctan, hypot, arctan2, degrees, radians, unwrap, deg2rad, rad2deg

Hyperbolic functions

sinh, cosh, tanh, arcsinh, arccosh, arctanh

Rounding

around, round_, rint, fix, floor, ceil, trunc

Sums, products, differences

prod, sum, nansum, cumprod, cumsum, diff, ediff1d, gradient, cross, trapz

Exponents and logarithms

exp, expm1, exp2, log, log10, log2, log1p, logaddexp, logaddexp2

Other special functions

i0, sinc

Floating point routines

signbit, copysign, frexp, ldexp

Arithmetic operations

add, reciprocal, negative, multiply, divide, power, subtract, true_divide, floor_divide, fmod, mod, modf, remainder

Handling complex numbers

angle, real, imag, conj

Miscellaneous

convolve, clip, sqrt, square, absolute, fabs, sign, maximum, minimum, fmax, fmin, nan_to_num, real_if_close, interp

Many more special functions are provided as ufuncs by SciPy

Linear algebra in NumPy

```
import numpy.linalg as LA
```

Matrix and vector products

dot, vdot, inner, outer, matmul, tensordot, einsum, LA.matrix_power, kron

Decompositions

LA.cholesky, LA.qr, LA.svd

Matrix eigenvalues

LA.eig, LA.eigh, LA.eigvals, LA.eigvalsh

Norms and other numbers

LA.norm, LA.cond, LA.det, LA.matrix_rank, LA.slogdet, trace

Solving equations and inverting matrices

LA.solve, LA.tensorsolve, LA.lstsq, LA.inv, LA.pinv, LA.tensorinv

Statistics in NumPy

Order statistics

amin, amax, nanmin, nanmax, ptp, percentile, nanpercentile

Averages and variances

median, average, mean, std, var, nanmedian, nanmean, nanstd, nanvar

Correlating

corrcoef, correlate, cov

Histograms

histogram, histogram2d, histogramdd, bincount, digitize

Mathematical operators

- Arithmetic operations are element-wise
- Logical operator return a bool array
- In place operations modify the array

Mathematical operators

- **Arithmetic operations are element-wise**
- Logical operator return a bool array
- In place operations modify the array

```
>>> a
array([1, 2, 3])
>>> b
array([ 4,  4, 10])
>>> a * b
array([ 4,  8, 30])
```

Mathematical operators

- Arithmetic operations are element-wise
- **Logical operator return a bool array**
- In place operations modify the array

```
>>> a
array([[ 0.93445601,  0.42984044,  0.12228461],
       [ 0.06239738,  0.76019703,  0.11123116],
       [ 0.14617578,  0.90159137,  0.89746818]])
>>> a > 0.5
array([[ True, False, False],
       [False,  True, False],
       [False,  True,  True]], dtype=bool)
```

Mathematical operators

- Arithmetic operations are element-wise
- Logical operator return a bool array
- **In place operations modify the array**

```
>>> a
array([[ 4, 15],
       [20, 75]])
>>> b
array([[ 2,  5],
       [ 5, 15]])
>>> a /= b
>>> a
array([[2, 3],
       [4, 5]])
```

End