The University of Manchester
Alliance Manchester Business School

BMAN73701

Programming in python for Business Analytics

Week 3: Lecture 2

Numerical Analysis

Dr. Xian Yang

xian.yang@manchester.ac.uk

Outline

- From basics to advanced analysis
- Labs: Less exercise-like, more case-study-like

- Week 3, Lecture **6**: Numerical Analysis (NumPy)
- Week 4, Lecture **7**: Data Exploration and Visualization (Pandas + Matplotlib)
- Week 4, Lecture **8**: Data preprocessing (Pandas + Scikit-learn)
- Week 5, Lecture **9&10**: Machine learning (Scikit-learn)
- Week 5, Revision Lecture

MANCHESTER 1824
The University of Manchester
Alliance Manchester Business School

BMAN73701

Programming in python™ for Business Analytics

Week 3: Lecture 2

Numerical Analysis

Part 1: Vectors and matrices

Part 2: Broadcasting

Part 3: Aggregations and other useful operations

Motivative example: Summing up 10M numbers

```
a = list(range(10**7))
```

For-loop

```
s = 0
for i in a:
    s += i
```

sum() over a list

```
s = sum(a)
```

np.sum()

```
x = np.array(a)
s = np.sum(x)
```

Which one is the fastest?

- Good integration with Pandas and Matplotlib
- Scikit-learn (ML) is built on top of NumPy

- **Fast** computations

- **Large** multi-dimensional arrays (vectors and matrices)

- Reference docs: https://docs.scipy.org/doc/numpy/reference/index.html

```
import numpy as np    # Just a shortcut.
```

NumPy arrays

```
import numpy as np
```

Vectors

$$\vec{a} = [0 \quad 1 \quad 2]$$

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

Out: array([0, 1, 2])


In: a.shape

Out: (3,) # 1D
```

Matrices

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix}$$

```
In: A = np.array([[0, 1, 2],
                  [3, 4, 5]])

Out: array([[0, 1, 2]
            [3, 4, 5] ])

In: A.shape

Out: (2,3) # 2D
```

Quiz 1

Which would give you the output includes 0, 2, 4, 6

A)
```
a = list(range(4))
a * 2
```

B)
```
a = list(range(4))
a + a
```

C)
```
a = np.arange(4)
a * 2
```

D)
```
a = np.arange(4)
a + a
```

NumPy arrays ≠ Lists

```
In: a = list(range(4))
Out: [0, 1, 2, 3]

In: a * 2
In: a + a
Out: [0, 1, 2, 3, 0, 1, 2, 3]

In:  a.append(10)
     a
Out: [0, 1, 2, 3, 10]
```

```
In: a = np.arange(4)
Out: array([0, 1, 2, 3])

In: a * 2
In: a + a
Out: array([0, 2, 4, 6])

In:  a = np.append(a, 10)
     a
Out: array([0, 1, 2, 3, 10])
```

Indexing: List vs NumPy

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix}$$

```
# A list of lists
A = [[0,1,2], [3,4,5]]
```

Return first row, first column ⇒ A[0][0]

Return first row ⇒ A[0]

Return first column ⇒ ??

```
# A NumPy matrix
A = np.array([[0,1,2], [3,4,5]])
```

Return first row, first column ⇒ A[0, 0]

Return first row ⇒ A[0, :]

Return first column ⇒ A[:, 0]

Slice Notation

`x[:]` is the same as `x[0:len(x):1]`

`x[START:END:STEP]`

Start counting at START (default 0)
Stop counting **before** END (default **len** or num. rows/columns),
Increment by STEP (default 1)

`x[0:2]` same as `x[[0,1]]` or `x[range(2)]`

`x[1:]` same as `x[range(1,len(x))]`

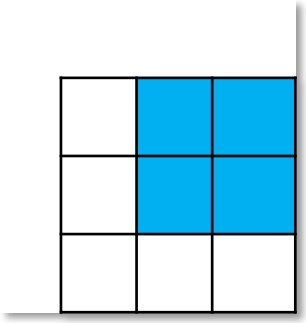
`x[:5]` same as `x[range(5)]`

`x[:]` same as `x[range(len(x))]`

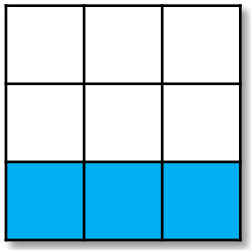
`x[:0:-1]` ⇒ `x[[len(x)-1,len(x)-2,…,1]]`

If STEP < 0,
default START is len(x) – 1
and default END is -1

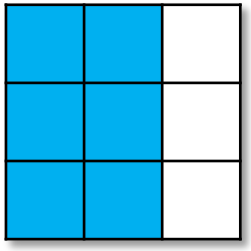
Slice Notation



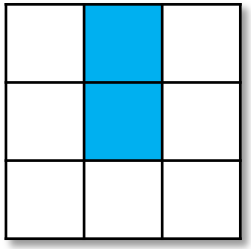
Expression	Shape
`A[0:2, 1:3]`	(2,2)
`A[:2, 1:]`	(2,2)



Expression	Shape
`A[2]`	(3,)
`A[2, :]`	(3,)
`A[2:, :]`	(1,3)



Expression	Shape
`A[:, :2]`	(3,2)
`A[:, [0,1]]`	(3,2)



Expression	Shape
`A[:2, 1]`	(2,)
`A[:2, 1:2]`	(2,1)

Slice Notation

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

```
In: x[:, 1]
Out: array([2, 5])
```

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

```
In: x[-1, :]
Out: array([4, 5, 6])
```

```
In: x[:2, 1:]
Out: array([[2, 1], [5, 6]])
```

```
In: x[::-1, [2,1,0]]
Out: array([[6, 5, 4], [1, 2, 3]])
```

Boolean Indexing

```
# A NumPy vector
x = np.array([0,3,1,4,2,5])
```

`x > 2` ⇒ `array([False, True, False, True, False, True])`

`x[x > 2]` ⇒ `array([3, 4, 5])`

```
# A list
x = [0,3,1,4,2,5]
```

`x[x > 2]` ⇒ `Error !!!`

```
# A NumPy Matrix
A = np.array([[0,3,1], [4,2,5]])
```

`A[A > 2]` ⇒ `array([3, 4, 5])`

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix}$$

Copies vs Views

`B = A` ⇒ Creates a **view**

`B[0, 0] = 10`

`A[0, 0]` ⇒ value ?

`B = A.copy()` ⇒ Creates a copy

`B[0, 0] = 20`

`A[0, 0]` ⇒ value ?

Slices are views not copies! Same as `B = A`

```
A[0, 0] = 1

row0 = A[0, :]          ⇒ returns first row

row0[0] = 5             ⇒ change first element of first row

A[0, 0]                 ⇒ value ?

slice_copy = A[0, :].copy()        # Creates a copy!
```

Example: Total gains and losses

Given a matrix X where X_{tj} is the net profit of department j in time period t, calculate:

$$gains = \sum_t^T \sum_j^D X_{tj} \quad \text{if } X_{tj} > 0$$

$$losses = \sum_t^T \sum_j^D X_{tj} \quad \text{if } X_{tj} < 0$$

for-loop

```
for t in range(X.shape[0]):
    for j in range(X.shape[1]):
        if X[t,j] > 0:
            gains += X[t,j]
        else:
            losses += X[t,j]
```

Time: ?? seconds

No loops!

```
gains = np.sum(X[X > 0])
losses= np.sum(X[X < 0])
```

Time: ?? seconds

Element-wise operators

(Most) operations are element-wise (+ - / * **)

```
A * 2  # multiply each element of A by 2

A ** 2        # square each element of A by 2

A * B  # Element-wise (not matrix product)

np.dot(A, B) # Matrix product: A × B
```

Recap

- NumPy arrays represent vectors and matrices

  ≠ Lists !

- Indexing Numpy arrays:
  - Slicing similar to lists
  - More powerful than lists
  - Boolean indexing

- Element-wise mathematical operations

- Mathematical operations that apply to many elements of a Numpy array are much faster than indexing with for-loops

MANCHESTER 1824
The University of Manchester
Alliance Manchester Business School

BMAN73701

Programming in python for Business Analytics

Week 3: Lecture 2

Numerical Analysis

Part 1: Vectors and matrices

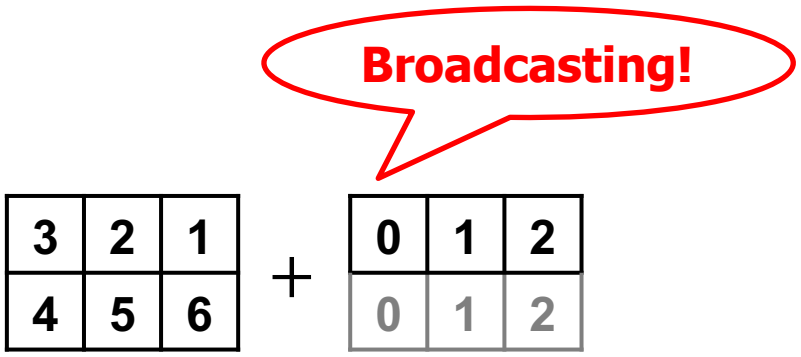
Part 2: Broadcasting

Part 3: Aggregations and other useful operations

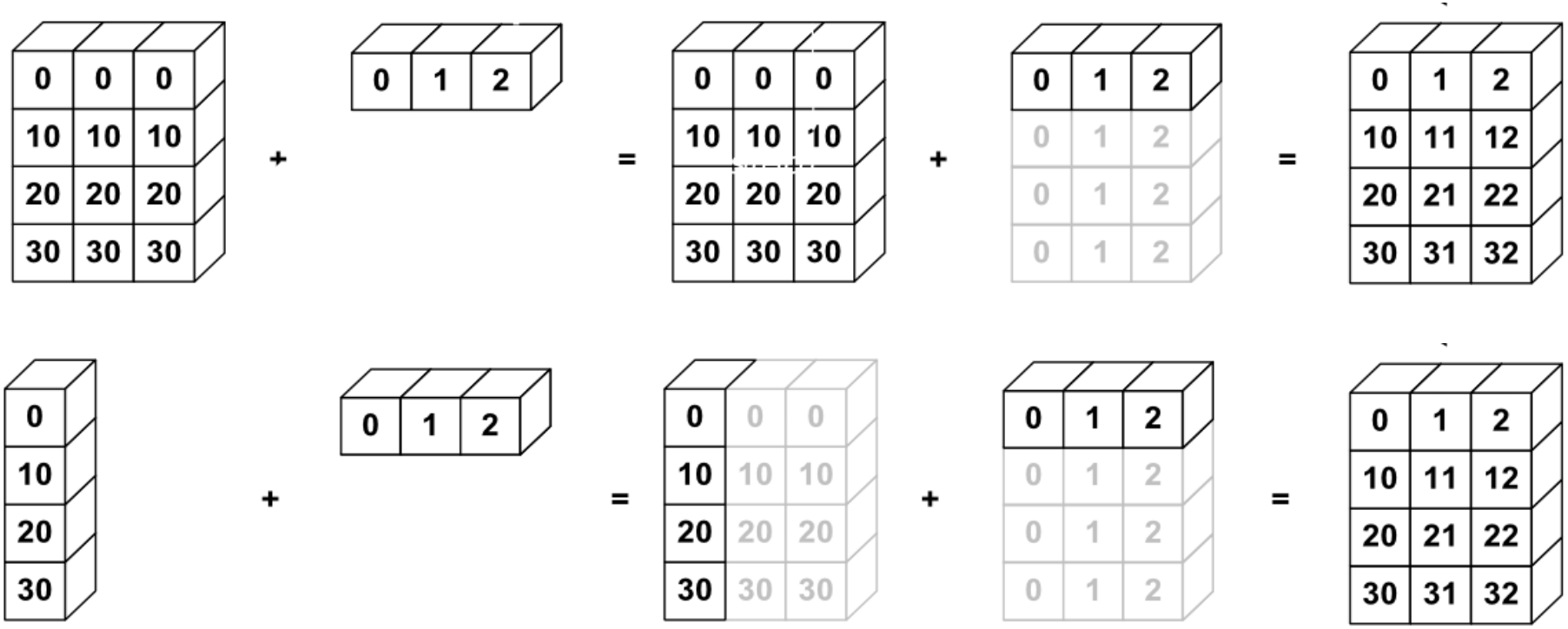
Broadcasting

$$\begin{bmatrix} 3 & 2 & 1 \\ 4 & 5 & 6 \end{bmatrix} + [0 \quad 1 \quad 2] = ?$$

```
A + x # Matrix (2,3) + Vector (3,)
```



Broadcasting

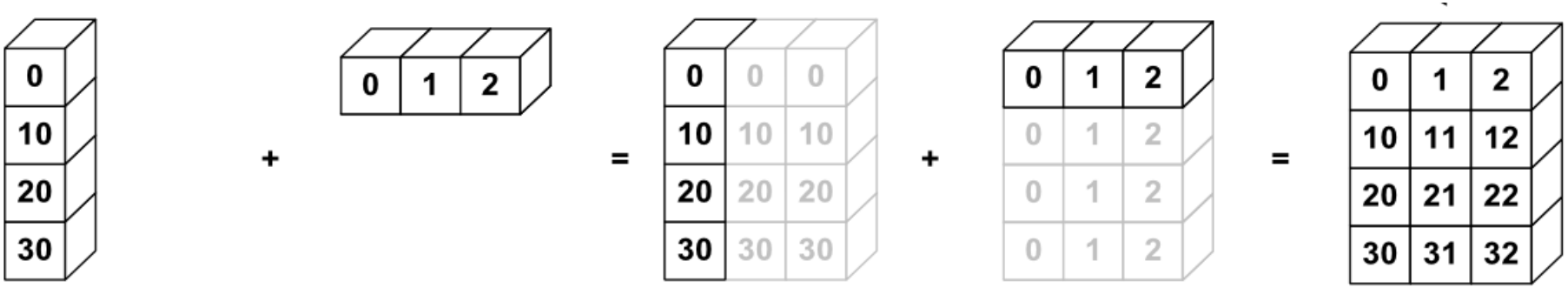


Dimensions Compatibility: For broadcasting to work, the dimensions of the arrays involved in the operation must be compatible. Compatible dimensions are either equal or one of them is 1. For example, a 3x3 array can be broadcasted with a 1x3 array because the dimensions are compatible.

Broadcasting Along Missing Dimensions: If an array has fewer dimensions than the other array, NumPy automatically adds new dimensions with size 1 to the smaller array to make their dimensions compatible.

http://www.scipy-lectures.org/intro/numpy/operations.html#broadcasting

Broadcasting



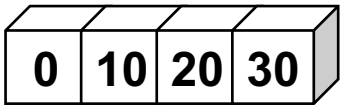

```
x = np.array([0,10,20,30])

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

In: x + y

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

In: x.reshape((4,1)) + y

Out: array([[ 0,  1,  2], [10, 11, 12],
            [20, 21, 22], [30, 31, 32]])
```

Reshaping

```
x = np.array([0,10,20,30])
```

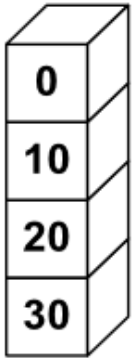
In: x

Out: array([0, 10, 20, 30])



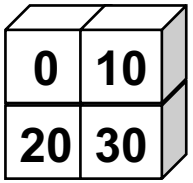
In: x.reshape((4,1))

Out: array([[0],
[10],
[20],
[30]])

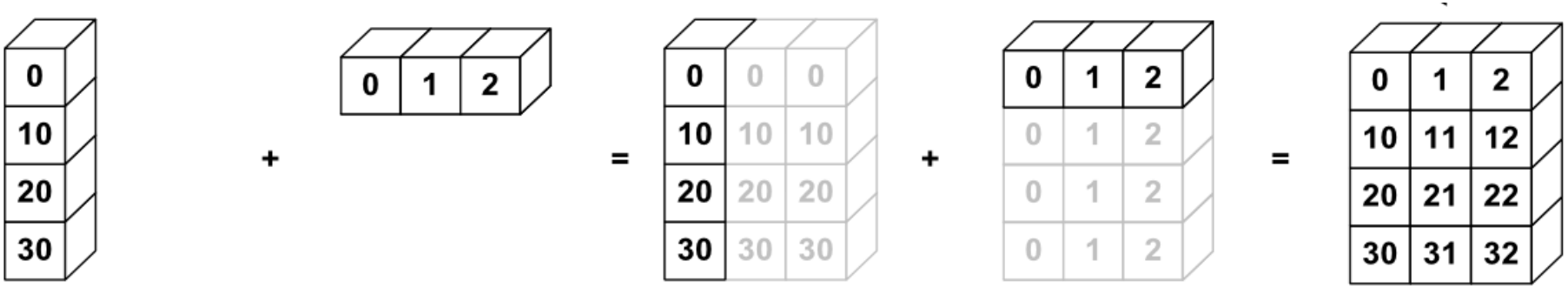


In: x.reshape((2,2))

Out: array([[0, 10],
[20, 30]])



Broadcasting: np.newaxis




```
x = np.array([0,10,20,30])

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

In: x + y

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

In: x[:, np.newaxis] + y

Out: array([[ 0,  1,  2], [10, 11, 12],
            [20, 21, 22], [30, 31, 32]])
```

Example: Total of outer product

Given two vectors x, y calculate the sum of every pairwise product of their elements:

$x =$ | 1 | 10 | 20 | 30 |

$y =$ | 2 | 3 | 4 | 5 |

$$total = \sum_{i=1}^{n} \sum_{j=1}^{n} x_i \cdot y_j$$

for-loops

```
n = x.shape[0]
total = 0
for i in range(n):
    for j in range(n):
        total += x[i] * y[j]
```

No loops!

```
total = np.sum(
        x.reshape((n,1)) * y)
```

1	1	1	1
10	10	10	10
20	20	20	20
30	30	30	30

*

2	3	4	5
2	3	4	5
2	3	4	5
2	3	4	5

Quiz 2: Broadcasting and reshape

1. Consider two arrays: A with shape (5, 1) and B with shape (1, 6). If you multiply A and B using NumPy, what will be the shape of the resulting array?

a) (1, 1)

b) (5, 6)

c) (5, 5)

d) (6, 6)

2. If you have an array Z of shape (5, 4) and you want to add a scalar value s to all elements of Z, how would broadcasting work?

a) It will reshape s to (5, 4) and then add it to Z.

b) It will generate an error since the shapes are different.

c) It will reshape s to (1, 1) and then add it to Z.

d) It doesn't need to reshape s; it will directly add s to every element of Z.

3. In numpy, if x = np.array([1, 2]) and y = np.array([[3], [4]]), what will be the output of the expression x + y?

a) array([[4, 5], [5, 6]])

b) array([4, 6])

c) Throws a ValueError

d) array([[4, 5], [7, 8]])

Recap

- Broadcasting allows mathematical operations between NumPy arrays of different shapes
- If shapes cannot be broadcast ⇒ Error!
- Mathematical operations using broadcasting are:
  - Faster to execute
  - Shorter to write

  } than for-loops

MANCHESTER 1824
The University of Manchester
Alliance Manchester Business School

BMAN73701

Programming in python for Business Analytics

Week 3: Lecture 2

Numerical Analysis

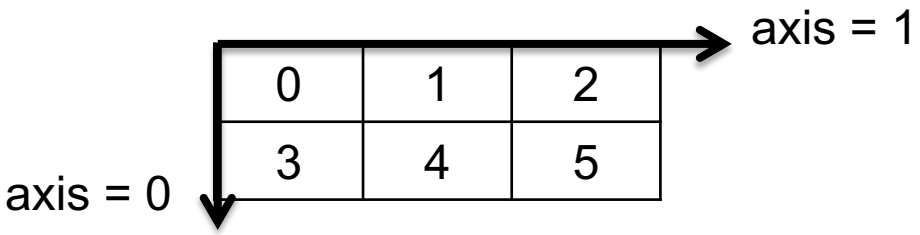
Part 1: Vectors and matrices

Part 2: Broadcasting

Part 3: Aggregations and other useful operations

Aggregations (Reductions)

Some operations aggregate: sum, min, max, mean, all, any, ...



`np.min(A)` ⇒ returns 1 number

`np.min(A, axis = 0)` ⇒ min along rows/vertical dimension, returns 1 number per column

`np.min(A, axis = 1)` ⇒ min along columns/horizontal dimension, returns 1 number per row

Quiz3: Maximum Mean Squared Error

$$\mathrm{P} = \begin{bmatrix} P_{11} & \cdots & P_{1n} \\ \vdots & \ddots & \vdots \\ P_{k1} & \cdots & P_{kn} \end{bmatrix}$$ each row gives n predictions by *k* ML methods

$\mathrm{obs} = [obs_1 \cdots obs_n]$ the actual observed values

Calculate the maximum Mean Squared Error (MSE) given as:

$$maxMSE = \max_{j=1,\ldots k} \left(\frac{1}{n} \sum_{i=1}^{n} \left(P_{ji} - obs_i \right)^2 \right)$$

A) maxMSE = np.max(np.mean((P-obs)**2, axis = 1))

B) maxMSE = np.max(np.mean((P-obs)**2, axis = 0))

Maximum Mean Squared Error

$$\mathrm{P} = \begin{bmatrix} P_{11} & \cdots & P_{1n} \\ \vdots & \ddots & \vdots \\ P_{k1} & \cdots & P_{kn} \end{bmatrix}$$ each row gives n predictions by *k* ML methods

$\mathrm{obs} = [obs_1 \cdots obs_n]$ the actual observed values

Calculate the maximum Mean Squared Error (MSE) given as:

$$maxMSE = \max_{j=1,\ldots k} \left(\frac{1}{n} \sum_{i=1}^{n} \left(P_{ji} - obs_i \right)^2 \right)$$

A) maxMSE = np.max(np.mean((P-obs)**2, axis = 1))

$$\begin{bmatrix} P_{11} & \cdots & P_{1n} \\ \vdots & \ddots & \vdots \\ P_{k1} & \cdots & P_{kn} \end{bmatrix} - \begin{bmatrix} obs_1 & \cdots & obs_n \\ \vdots & \ddots & \vdots \\ obs_1 & \cdots & obs_n \end{bmatrix}$$

we want to get the mean for each resulting row => 1 value per row

Functions vs. Methods

Most functions in NumPy have an equivalent method

np.min(A) ⇒ A.min()

np.min(A, axis = 0) ⇒ A.min(axis = 0)

- Methods can only be applied to NumPy arrays!

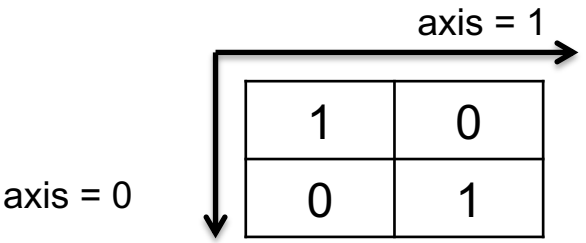
np.min([1,2,3]) ⇒ OK

[1,2,3].min() ⇒ Error

- Some methods modify the array in-place!
  - e.g., A.sort()

Boolean arrays: Any vs. All

```
b = np.array([1, 1, 0, 0]) # 1 is True, 0 is False
np.logical_not(b)
np.logical_and(b, b)
np.logical_or(b, b)
np.all(b)      # all True?
np.any(b)      # any True?
```



B = np.array([[1,0],[0,1]])

np.all(B, axis = 0) ⇒ all True along rows ?
returns 1 value per column

np.any(B, axis = 1) ⇒ any True along columns ?
returns 1 value per row

Quiz 4

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

1. np.any(A < 0)

a) Are there negative values?

b) Are all negative values?

2. np.all(A < 0, axis = 0)

a) Which columns have only negative values?

b) Which columns have at least one negative values?

c) Which rows have only negative value?

d) Which rows have at least one negative value?

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

1. a) Are there negative values? ⇒ np.any(A < 0)

Are all negative values? ⇒ np.all(A < 0)

2. a) Which columns have only negative values?
⇒ np.all(A < 0, axis = 0)

Which rows have at least one negative value?
⇒ np.any(A < 0, axis = 1)

Sorting

- Direct sorting np.sort(array, axis = 0 or 1)

  np.sort(a) ⇒ sort ascending

  -np.sort(-a) ⇒ sort descending

  np.sort(A, axis=0) ⇒ sort each column (*along rows*)

  The default is axis = -1, which sorts along the last axis.

```
x = np.array([11,12,10,9])
In:     np.sort(x)
Out: [9,10,11,12]

In: -np.sort(-x)
Out: [12,11,10,9]
```

Indirect Sorting

- Direct sorting (ascending) `np.sort(array, axis=)`
- Indirect sorting
  - `np.argsort(array, axis=)` returns the indices that would sort an array along a specified axis in ascending order.
    `np.argmin(array, axis=)`
    `np.argmax(array, axis=)`

```
x = np.array([12,11,10,9])

In:    np.sort(x)
Out: [9,10,11,12]

In: np.argsort(x)
Out: [3,2,1,0]

In: x[np.argsort(x)]
Out: [9,10,11,12]
```

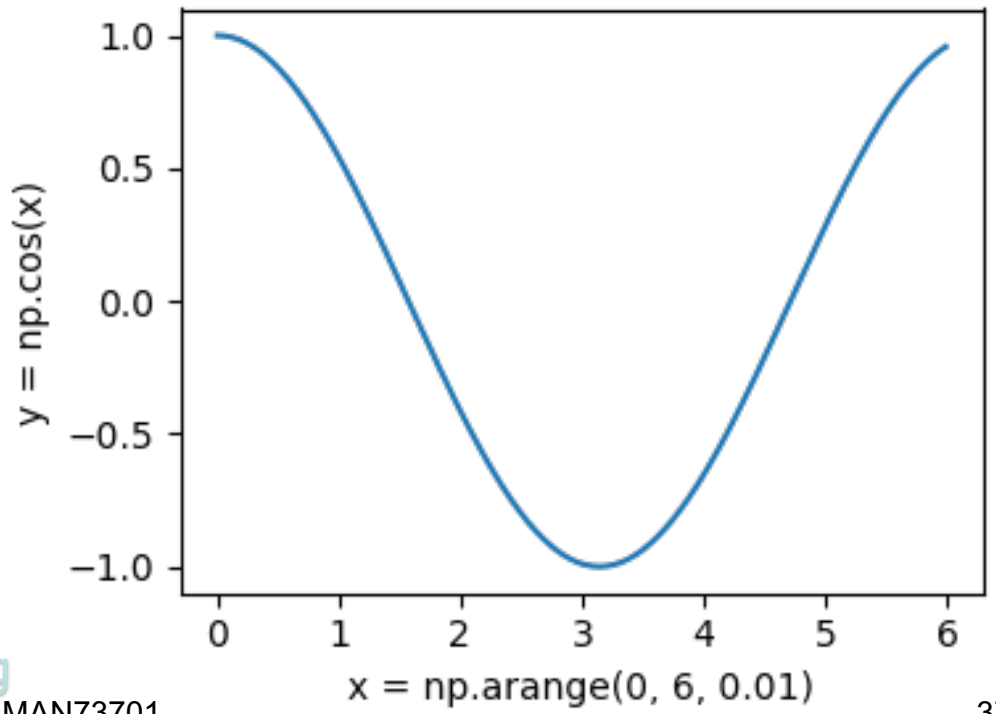
```
In: np.max(x)

Out: 12

In: np.argmax(x)

Out: 0
```

Minimisation by indirect sorting

- Find the x value that produces the minimum of cos(x) between [0, 6] with a precision of 0.0000001

```
x = np.arange(0, 6, 0.0000001)
y = np.cos(x)
i = np.argmin(y)
print(x[i])
```

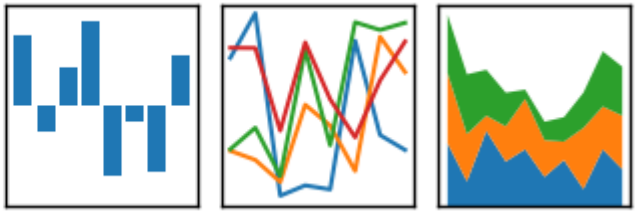
```
Out: 3.1415926999999
```



Next week

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



- Python library for data manipulation and analysis

matplotlib

- Advanced customisation requires using Matplotlib functions
- Complex plots require Matplotlib concepts