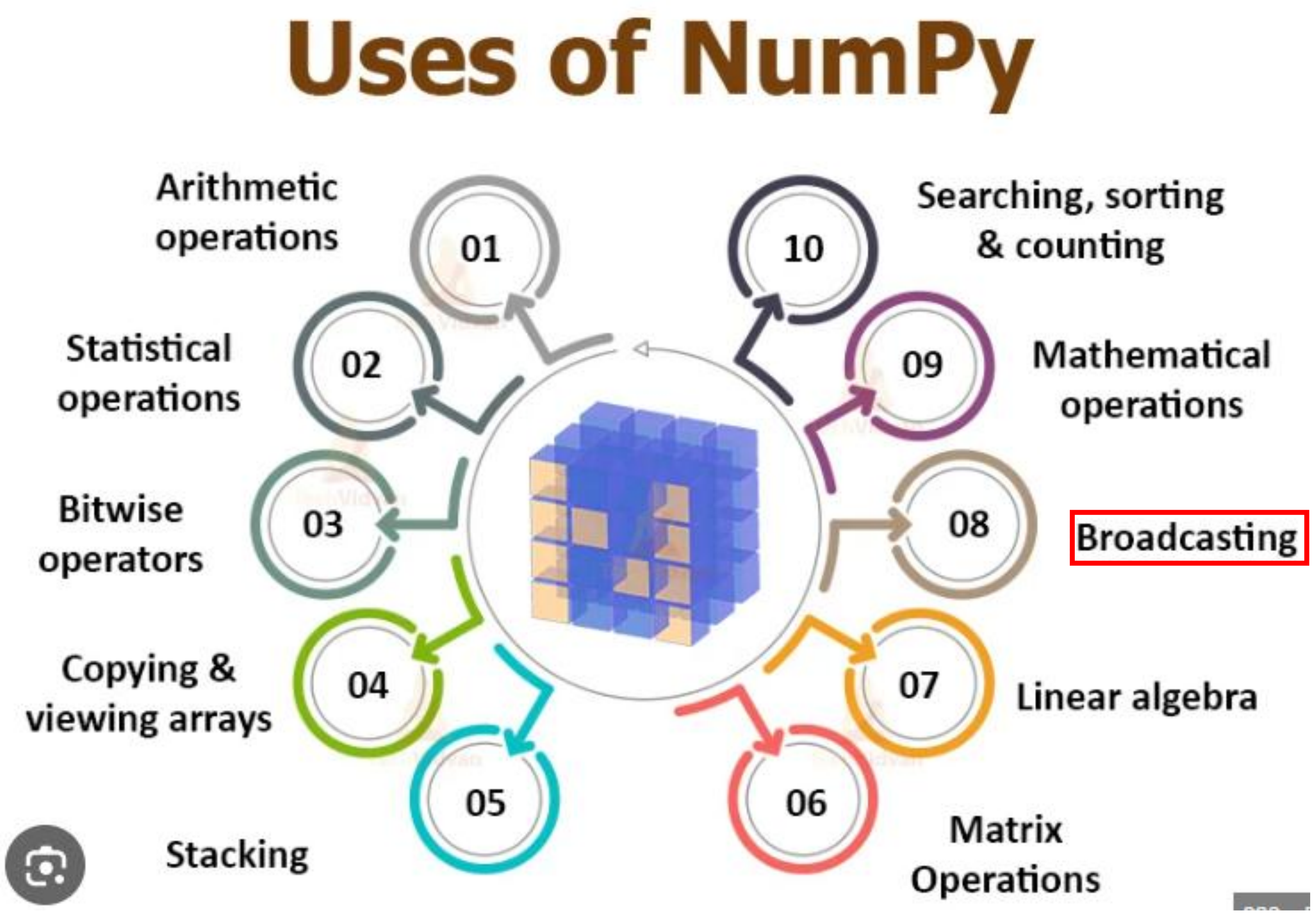
(U4284) Python程式設計
Package Intro - Numpy

Speaker: 吳淳硯

Why NumPy Matters

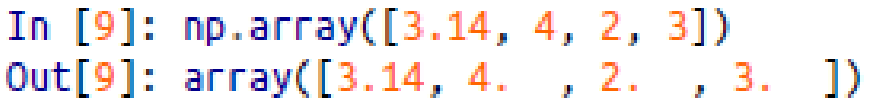


Intro

- NumPy (Short for *Numerical Python* with alias *np*).

- The core of NumPy is implemented in C and provides efficient functions for manipulating and processing arrays.

- NumPy arrays bear some resemblance to built-in *list* type. Python lists are generic containers of objects, NumPy arrays are homogeneous and typed arrays of fixed size. If the types do not match, NumPy will upcast them according to its type promotion rules.



```
In [9]: np.array([3.14, 4, 2, 3])
Out[9]: array([3.14, 4.  , 2.  , 3.  ])
```

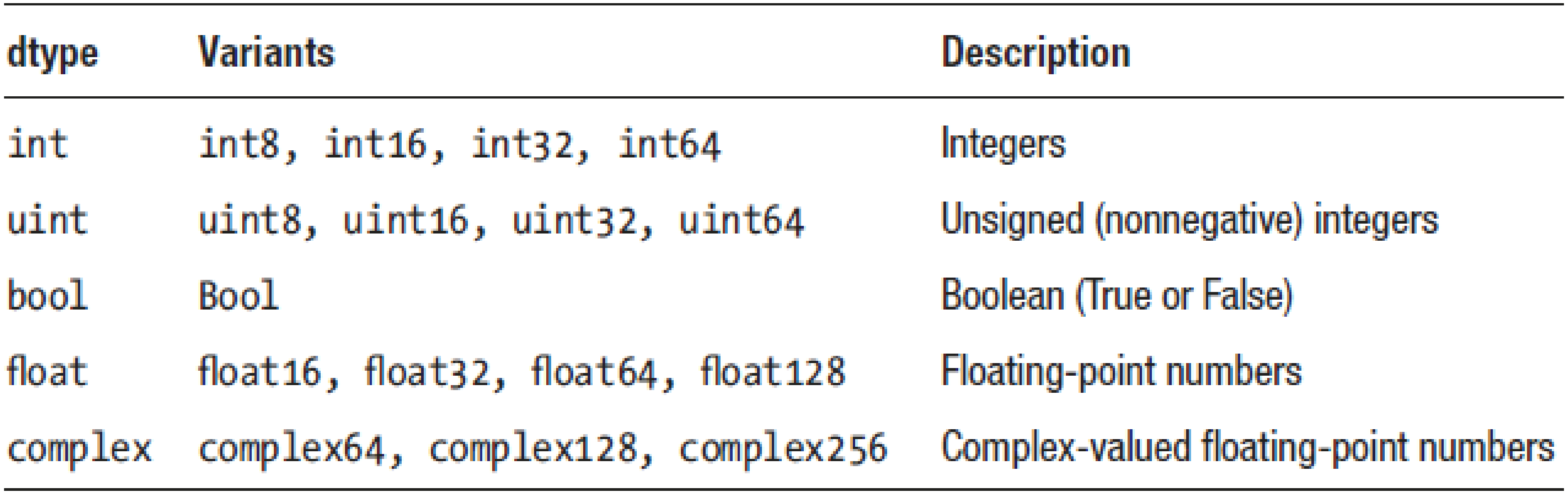
- For these and other reasons, operations and functions acting on NumPy arrays can be much more efficient than those using Python lists.

Standard Data Types

- Unlike Python lists, which are always 1-dim sequences, NumPy arrays can be multi-dim. Especially for large arrays, it is more efficient to create scratch using routines built into NumPy.
- Because NumPy is built in C, the types will be familiar to users of C. Note that when constructing an array, they can specified using a string or associated NumPy object.

```
np.zeros(10, dtype='int16')

np.zeros(10, dtype=np.int16)
```



dtype	Variants	Description
`int`	`int8, int16, int32, int64`	Integers
`uint`	`uint8, uint16, uint32, uint64`	Unsigned (nonnegative) integers
`bool`	`Bool`	Boolean (True or False)
`float`	`float16, float32, float64, float128`	Floating-point numbers
`complex`	`complex64, complex128, complex256`	Complex-valued floating-point numbers

Basic of Array - 0

- Each array has attributes including

Attribute	Description
Shape	A tuple that contains the number of elements (i.e., the length) for each dimension (axis) of the array.
Size	The total number elements in the array.
Ndim	Number of dimensions (axes).
nbytes	Number of bytes used to store the data.
dtype	The data type of the elements in the array.

- In 1-dim array, the ith value (counting from zero) can be accessed by specifying the desired index in square brackets. To index from the end of array, you can use negative indices.

```
In [3]: x1
Out[3]: array([9, 4, 0, 3, 8, 6])

In [4]: x1[0]
Out[4]: 9

In [5]: x1[4]
Out[5]: 8
```

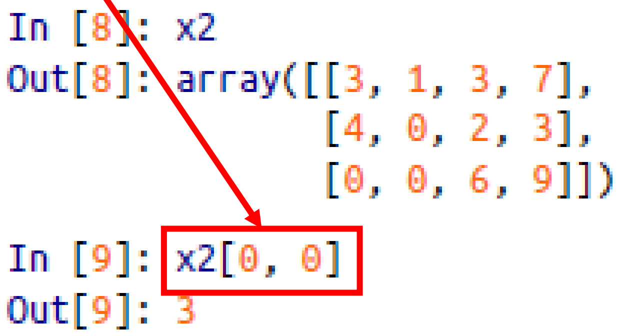
```
In [6]: x1[-1]
Out[6]: 6

In [7]: x1[-2]
Out[7]: 8
```

National Taipei University

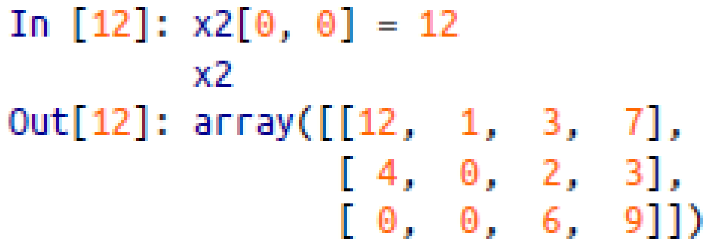
Basic of Array - 1

- In a multi-dim array, items can be accessed using a comma-separate *(row , column)* tuple. Values can also be modified using any of the preceding index notation.



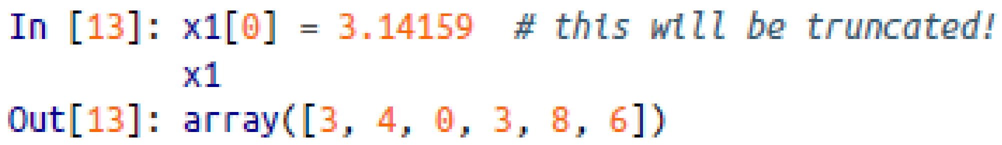
```
In [8]: x2
Out[8]: array([[3, 1, 3, 7],
               [4, 0, 2, 3],
               [0, 0, 6, 9]])

In [9]: x2[0, 0]
Out[9]: 3
```



```
In [12]: x2[0, 0] = 12
         x2
Out[12]: array([[12,  1,  3,  7],
                [ 4,  0,  2,  3],
                [ 0,  0,  6,  9]])
```

- NumPy array have fixed type. If you attempt to insert a floating point value into an integer array, the value will be silently truncated.



```
In [13]: x1[0] = 3.14159  # this will be truncated!
         x1
Out[13]: array([3, 4, 0, 3, 8, 6])
```

Basic of Array - 2

- Access subarrays with the *slice* notation, marked by the colon (:) character.

- The NumPy slicing syntax follows that of standard Python list
  - x[start : stop : step]
  - Default start = 0, stop = size of dim, step = 1

- A potential confusing case is when the step value is negative. In this case, the defaults for start and stop are swapped.

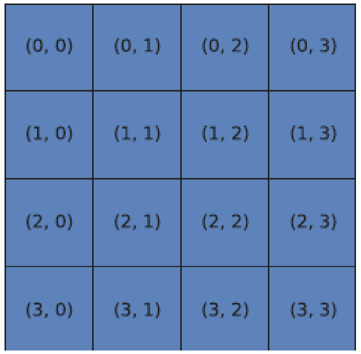
- Accessing single rows or columns of an array can be done by combing indexing and slicing, using an empty slice marked by a single colon (:).

Array Indexing and Slicing Expression

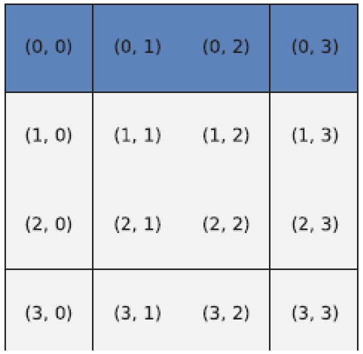
Expression	Description
`a[m]`	Select element at index *m*, where *m* is an integer (start counting form 0).
`a[-m]`	Select the *n*th element from the end of the list, where *n* is an integer. The last element in the list is addressed as –1, the second to last element as –2, and so on.
`a[m:n]`	Select elements with index starting at *m* and ending at *n* – 1 (*m* and *n* are integers).
`a[:]` or `a[0:-1]`	Select all elements in the given axis.
`a[:n]`	Select elements starting with index 0 and going up to index *n* – 1 (integer).
`a[m:]` or `a[m:-1]`	Select elements starting with index *m* (integer) and going up to the last element in the array.
`a[m:n:p]`	Select elements with index *m* through *n* (exclusive), with increment *p*.
`a[::-1]`	Select all the elements, in reverse order.

Visual indexing methods for NumPy arrays

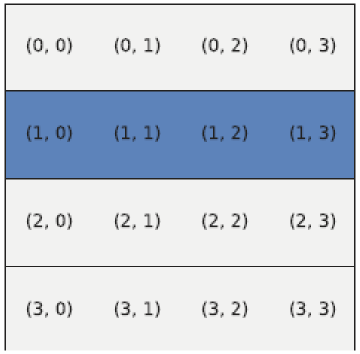
data



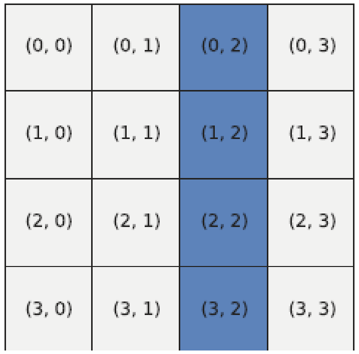
data[0]



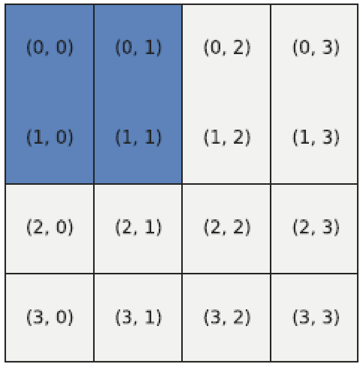
data[1, :]



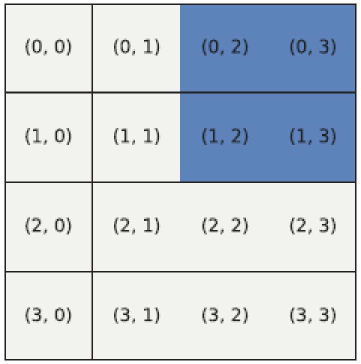
data[:, 2]



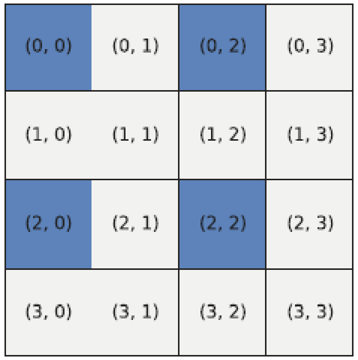
data[0:2, 0:2]



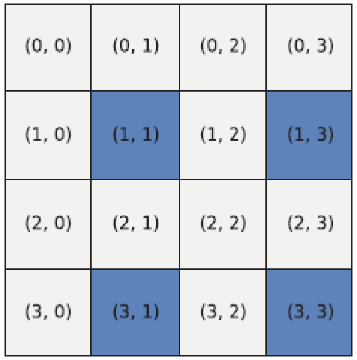
data[0:2, 2:4]



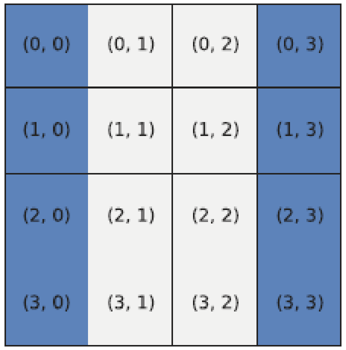
data[::2, ::2]



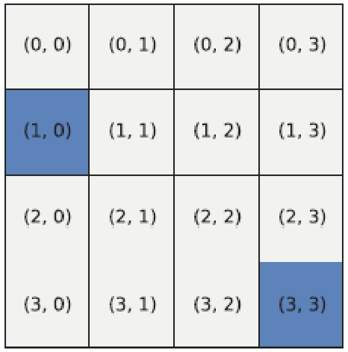
data[1::2, 1::2]



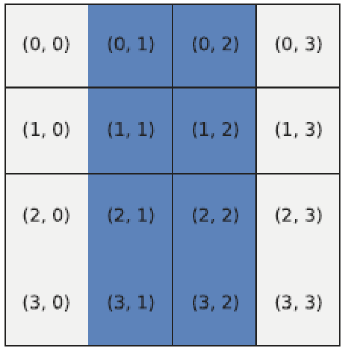
data[:,[0,3]]



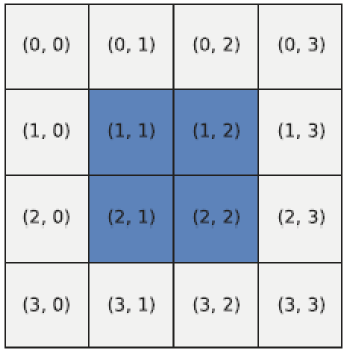
data[[1,3],[0,3]]



data[:,np.array([False, True, True, False])]



data[1:3,np.array([False, True, True, False]



Basic of Array - 3

- NumPy array slices are returned as *views* rather than *copies* of the array data, which means that subarray is mutable.

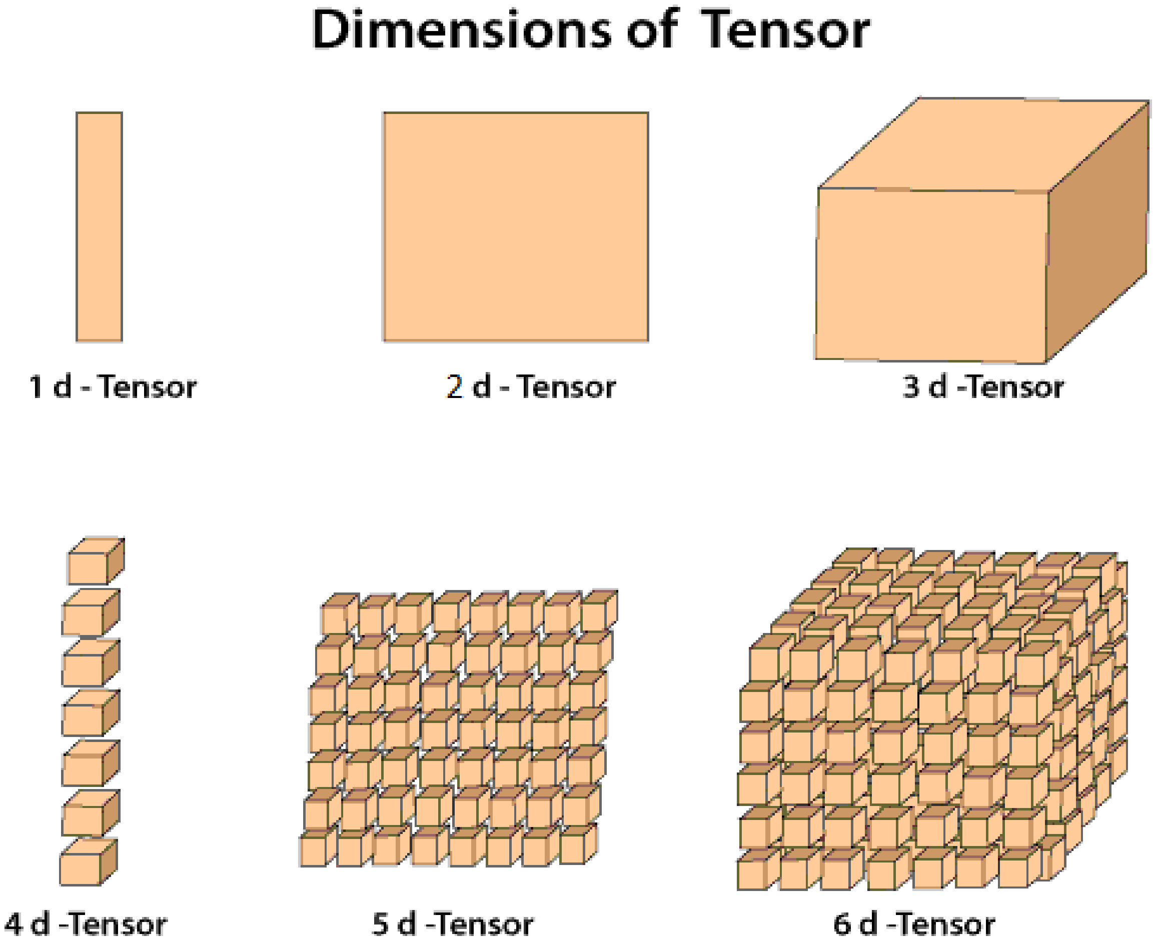
- Work with NumPy array, it is often to rearrange arrays and alter the way they are interpreted. For example, an $n \times n$ matrix array could be rearranged into a vector of length n^2, or a set of 1-dim arrays could be concatenated together or stacked next to each other to form a matrix.

- Concatenation of two arrays in NumPy, is primarily accomplished using the np.concatenate, np.vstack, and np.hstack.

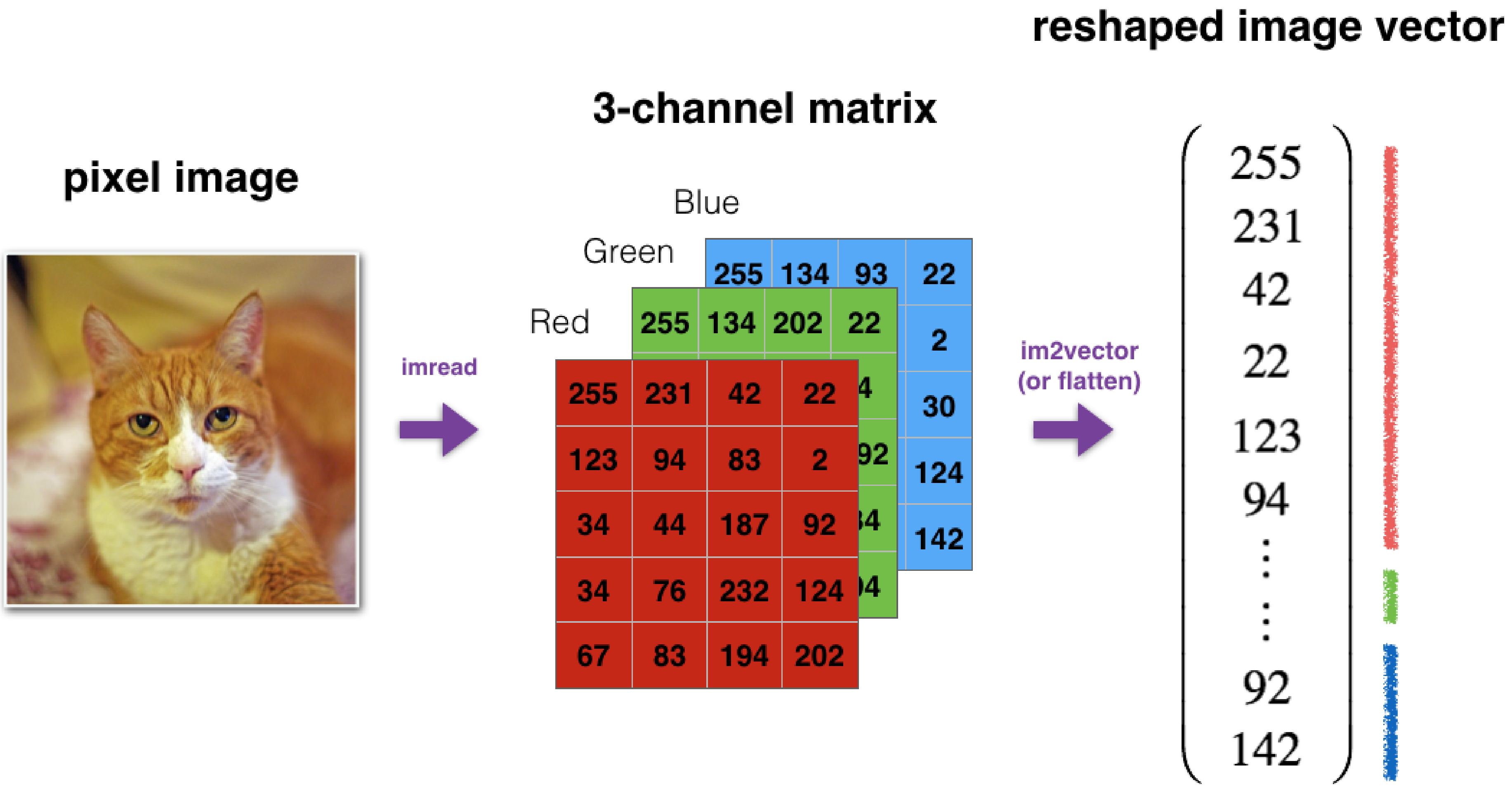
- The opposite of concatenation is splitting, which is implemented by the functions np.split, np.hsplit and np.vsplit. Similarly, for higher-dimensional arrays, np.dsplit will split arrays along the third axis.

Tensor to NumPy Array

- Tensor is a general name of multi-way array data. For example, 1d-tensor is vector, 2d-tensor is a matrix, and 3d-tensor is a cube. We can image 4d-tensor as a vector of cubes. In similar way, 5d-tensor is a matrix of cubes, and 6d-tensor is a cube of cubes.



Color Image = 3d-tensor (RGB)



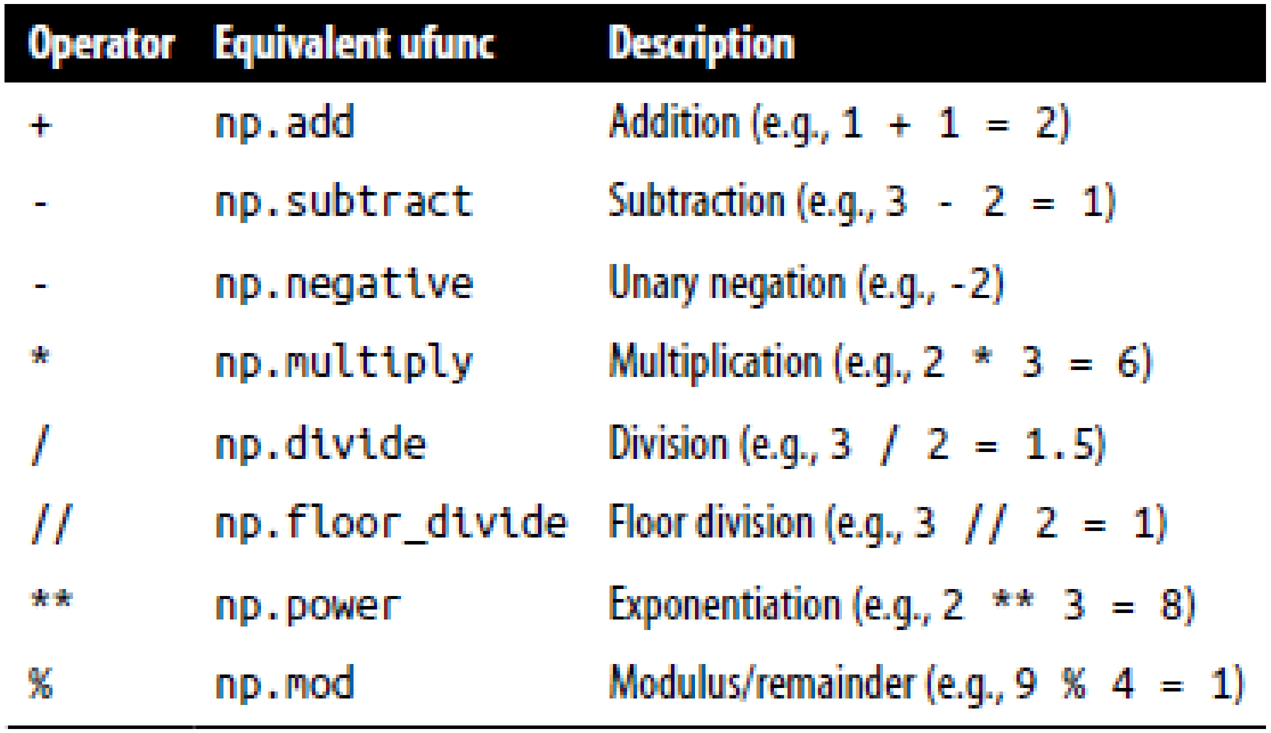
The Slowness of Loops

- Python’s default implementation (known as CPython) does some operations very slowly.
- This is partly due to the dynamic, interpreted nature of the language; types are flexible, so sequences of operations cannot be compiled down to efficient machine code as in languages like C and Fortran.
- The key to making it fast is to use vectorized operations, generally implemented through NumPy’s universal functions (ufuncs).
- This vectorized approach is designed to push the loop into the compiled layer that underlies NumPy, leading to much faster execution.

Ufuncs

- Ufuncs exist in two flavors:
  - *unary ufuncs* – operate on a single input
  - *binary ufuncs* – operate on two inputs

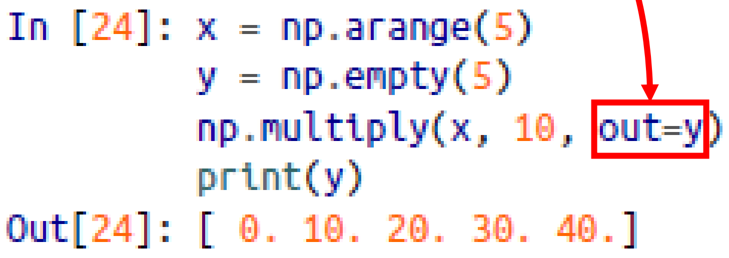
- Array Arithmetic



Operator	Equivalent ufunc	Description
`+`	`np.add`	Addition (e.g., `1 + 1 = 2`)
`-`	`np.subtract`	Subtraction (e.g., `3 - 2 = 1`)
`-`	`np.negative`	Unary negation (e.g., `-2`)
`*`	`np.multiply`	Multiplication (e.g., `2 * 3 = 6`)
`/`	`np.divide`	Division (e.g., `3 / 2 = 1.5`)
`//`	`np.floor_divide`	Floor division (e.g., `3 // 2 = 1`)
`**`	`np.power`	Exponentiation (e.g., `2 ** 3 = 8`)
`%`	`np.mod`	Modulus/remainder (e.g., `9 % 4 = 1`)

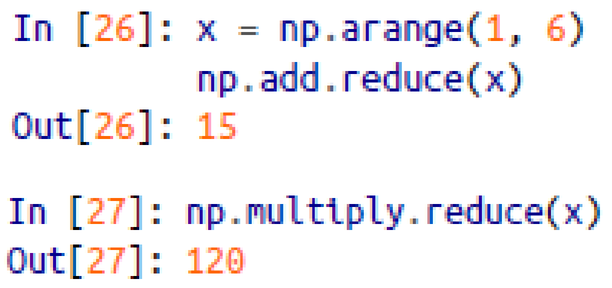
Advanced Ufuncs Feature

- Specify Output
  - Specify the array where the result of the calculation will be stored. This can be done using the out argument of function.



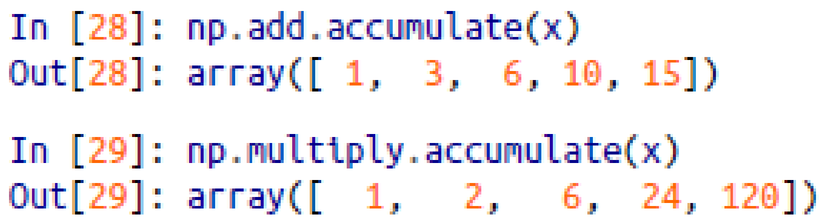
```
In [24]: x = np.arange(5)
         y = np.empty(5)
         np.multiply(x, 10, out=y)
         print(y)
Out[24]: [  0.  10.  20.  30.  40.]
```

- Aggregation
  - *Reduce* an array with a particular operation, we can use the *reduce* method of any ufunc.



```
In [26]: x = np.arange(1, 6)
         np.add.reduce(x)
Out[26]: 15

In [27]: np.multiply.reduce(x)
Out[27]: 120
```

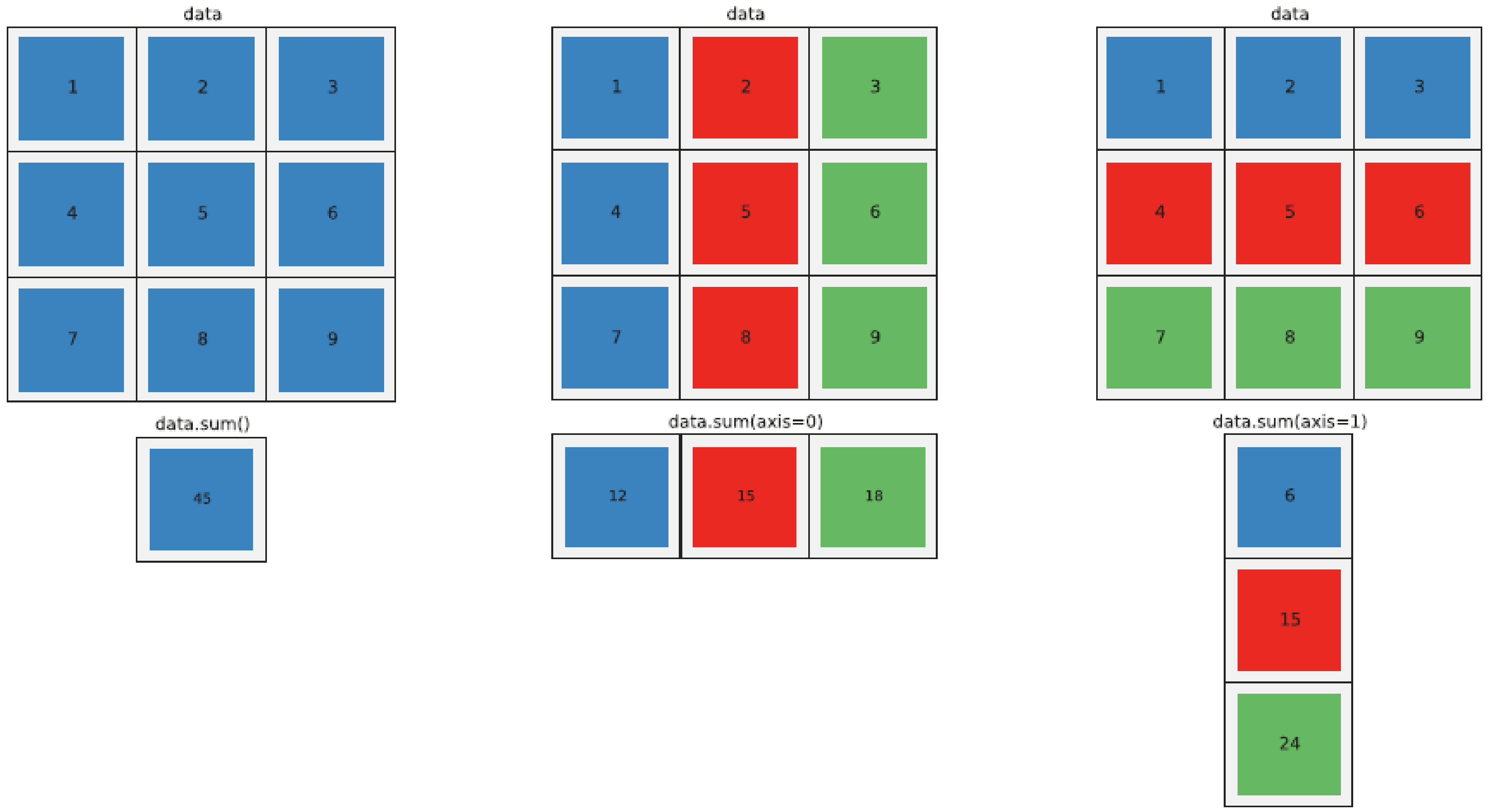


```
In [28]: np.add.accumulate(x)
Out[28]: array([ 1,  3,  6, 10, 15])

In [29]: np.multiply.accumulate(x)
Out[29]: array([  1,   2,   6,  24, 120])
```

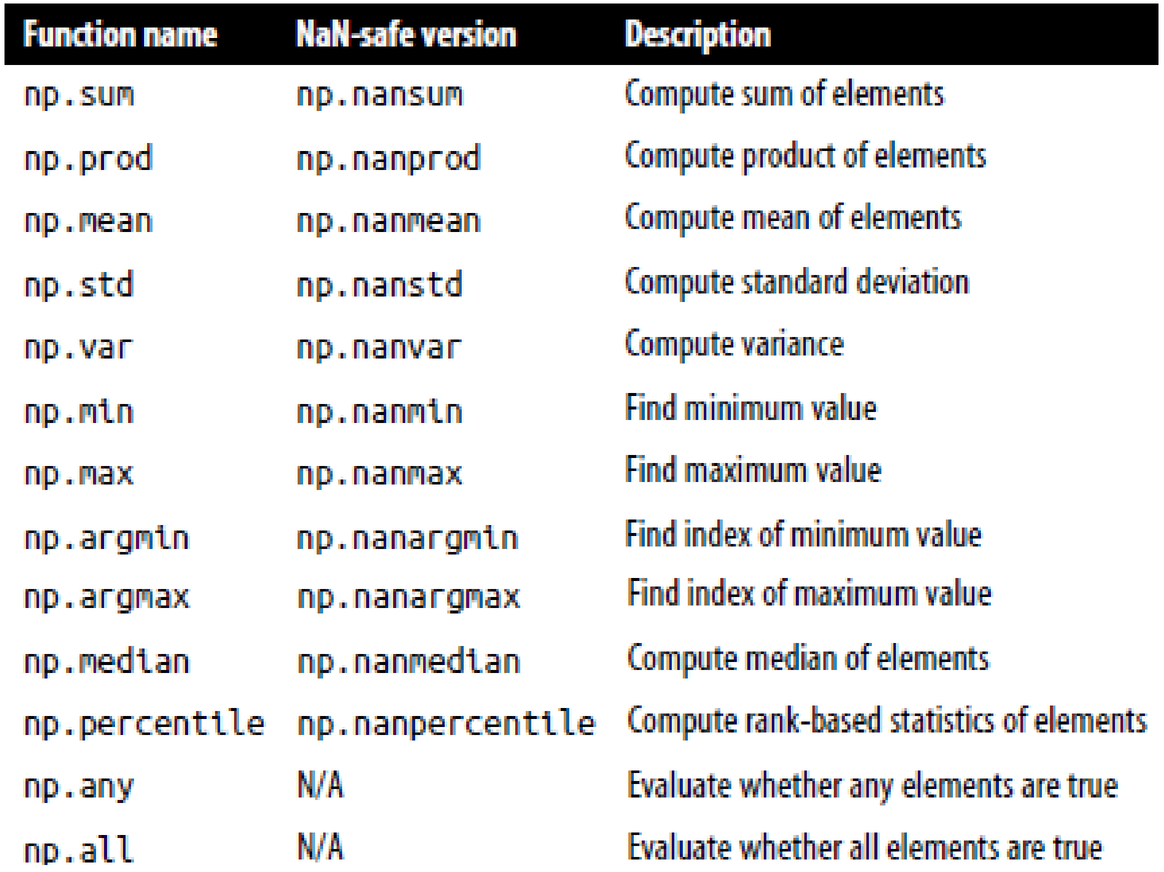
國立臺北大學 National Taipei University

A visual illustration of aggregation



NaN-safe Aggregation

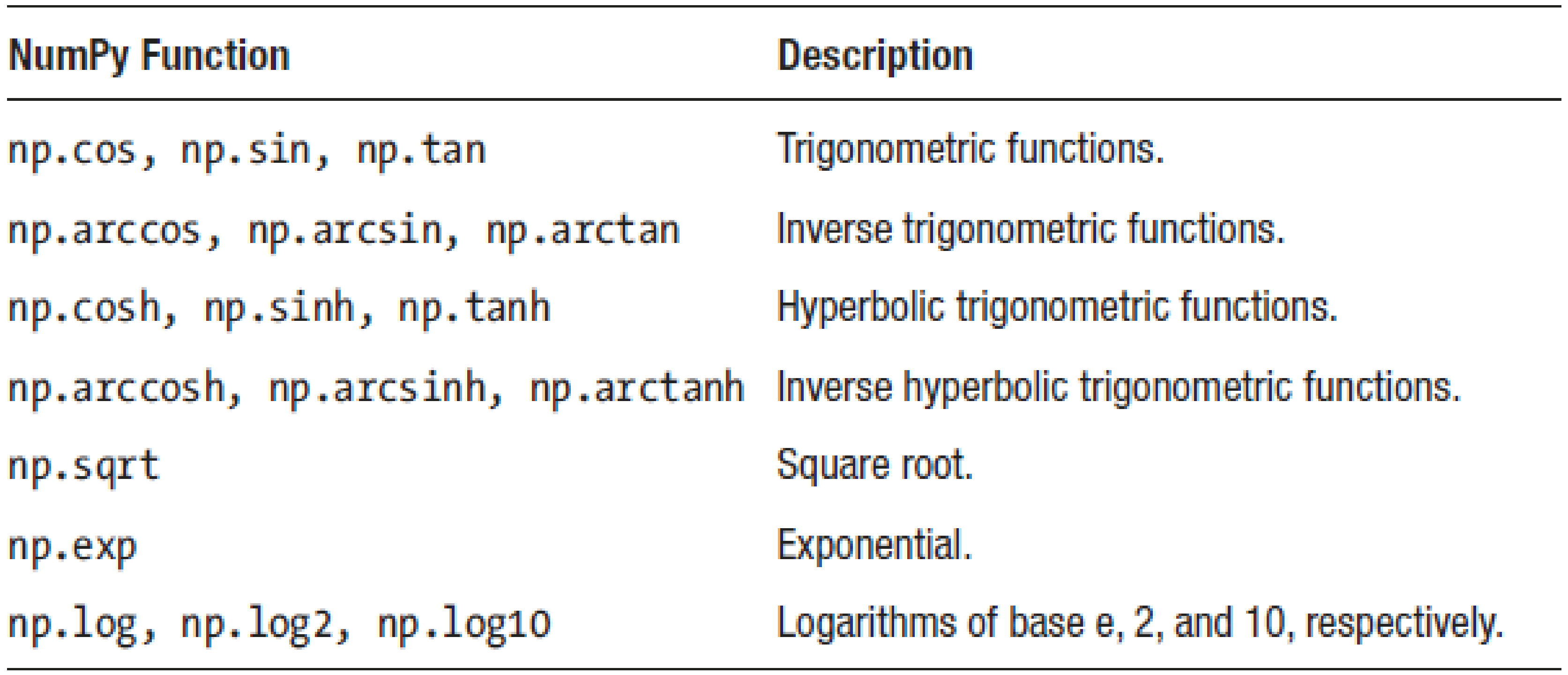
- NumPy has fast built-in aggregation functions for working on arrays.
- NumPy provides several other aggregation functions with a similar API, and additionally most have a NaN-safe counterpart that computes the result while ignoring missing values.



Function name	NaN-safe version	Description
`np.sum`	`np.nansum`	Compute sum of elements
`np.prod`	`np.nanprod`	Compute product of elements
`np.mean`	`np.nanmean`	Compute mean of elements
`np.std`	`np.nanstd`	Compute standard deviation
`np.var`	`np.nanvar`	Compute variance
`np.min`	`np.nanmin`	Find minimum value
`np.max`	`np.nanmax`	Find maximum value
`np.argmin`	`np.nanargmin`	Find index of minimum value
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements
`np.any`	N/A	Evaluate whether any elements are true
`np.all`	N/A	Evaluate whether all elements are true

Elementwise Function

- NumPy provides vectorized functions for elementwise evaluation of many elementary mathematical functions and operations.



NumPy Function	Description
`np.cos, np.sin, np.tan`	Trigonometric functions.
`np.arccos, np.arcsin, np.arctan`	Inverse trigonometric functions.
`np.cosh, np.sinh, np.tanh`	Hyperbolic trigonometric functions.
`np.arccosh, np.arcsinh, np.arctanh`	Inverse hyperbolic trigonometric functions.
`np.sqrt`	Square root.
`np.exp`	Exponential.
`np.log, np.log2, np.log10`	Logarithms of base e, 2, and 10, respectively.

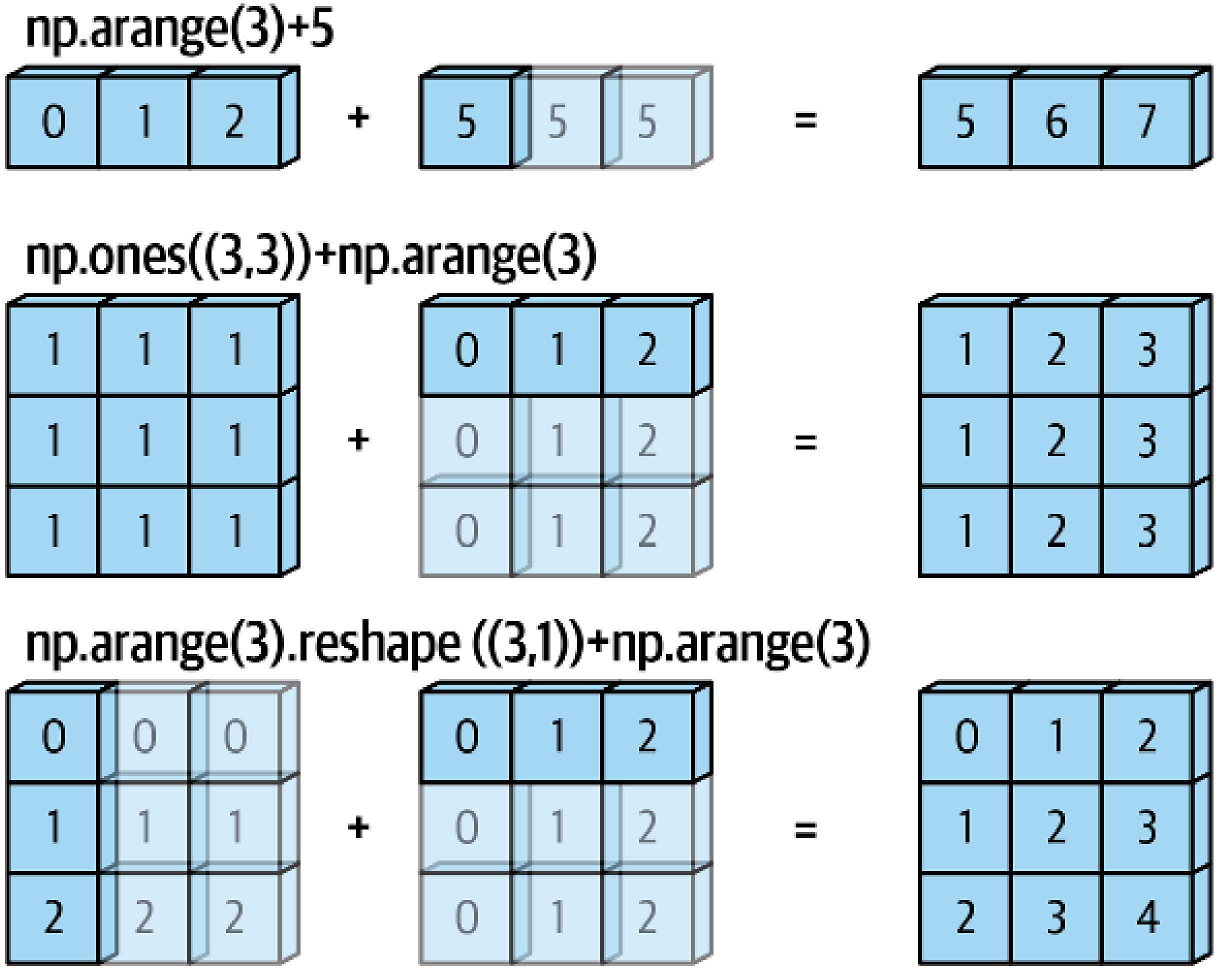
Elementwise Operation

NumPy Function	Description
`np.add, np.subtract, np.multiply, np.divide`	Addition, subtraction, multiplication, and division of two NumPy arrays.
`np.power`	Raises first input argument to the power of the second input argument (applied elementwise).
`np.remainder`	The remainder of division.
`np.reciprocal`	The reciprocal (inverse) of each element.
`np.real, np.imag, np.conj`	The real part, imaginary part, and the complex conjugate of the elements in the input arrays.
`np.sign, np.abs`	The sign and the absolute value.
`np.floor, np.ceil, np.rint`	Convert to integer values.
`np.round`	Rounds to a given number of decimals.

National Taipei University

Broadcasting - 0

- A set of rules by which NumPy lets you apply binary operation between arrays of different sizes and shapes.



國立臺北大學
National Taipei University

Broadcasting - 1

- Rules of Broadcasting
  - **Rule 1** – If the two arrays differ in their number of dim, the shape of the one with fewer dim is padded with ones on its leading (left) side.
  - **Rule 2** – If the shape of the two arrays does not match in any dim, the array with shape equal to 1 in that dim is stretched to match the other shape.
  - **Rule 3** – If in any dim the sizes disagree and neither is equal to 1, an error is raised.
- An example of not compatible. Consider shape of $M_{(3,2)}$ is (3,2) and shape of $A_{(3,)}$ is (3,).
  - By **Rule 1**, we must pad the shape of A with 1, $A_{(1,3)}$.
  - By **Rule 2**, the first dim of $A_{(1,3)}$ is then stretched to match that of $M_{(3,2)}$, $A_{(3,3)}$.
  - By **Rule 3**, the final shapes do not match that of $M_{(3,2)}$, so they are incompatible.

Mask

- Masking comes up when you want to extract, modify, count, or otherwise manipulate values in an array based on some criterion.
- NumPy also implements comparison operators as element-wise ufuncs.

Operator	Equivalent ufunc	Operator	Equivalent ufunc
`==`	`np.equal`	`!=`	`np.not_equal`
`<`	`np.less`	`<=`	`np.less_equal`
`>`	`np.greater`	`>=`	`np.greater_equal`

- The result of these comparison operators is always an array with a Boolean data type.

```
x
Out[12]: array([[9, 4, 0, 3],
                [8, 6, 3, 1],
                [3, 7, 4, 0]])

In [13]: x < 6
Out[13]: array([[False,  True,  True,  True],
                [False, False,  True,  True],
                [ True, False,  True,  True]])
```

Sorting

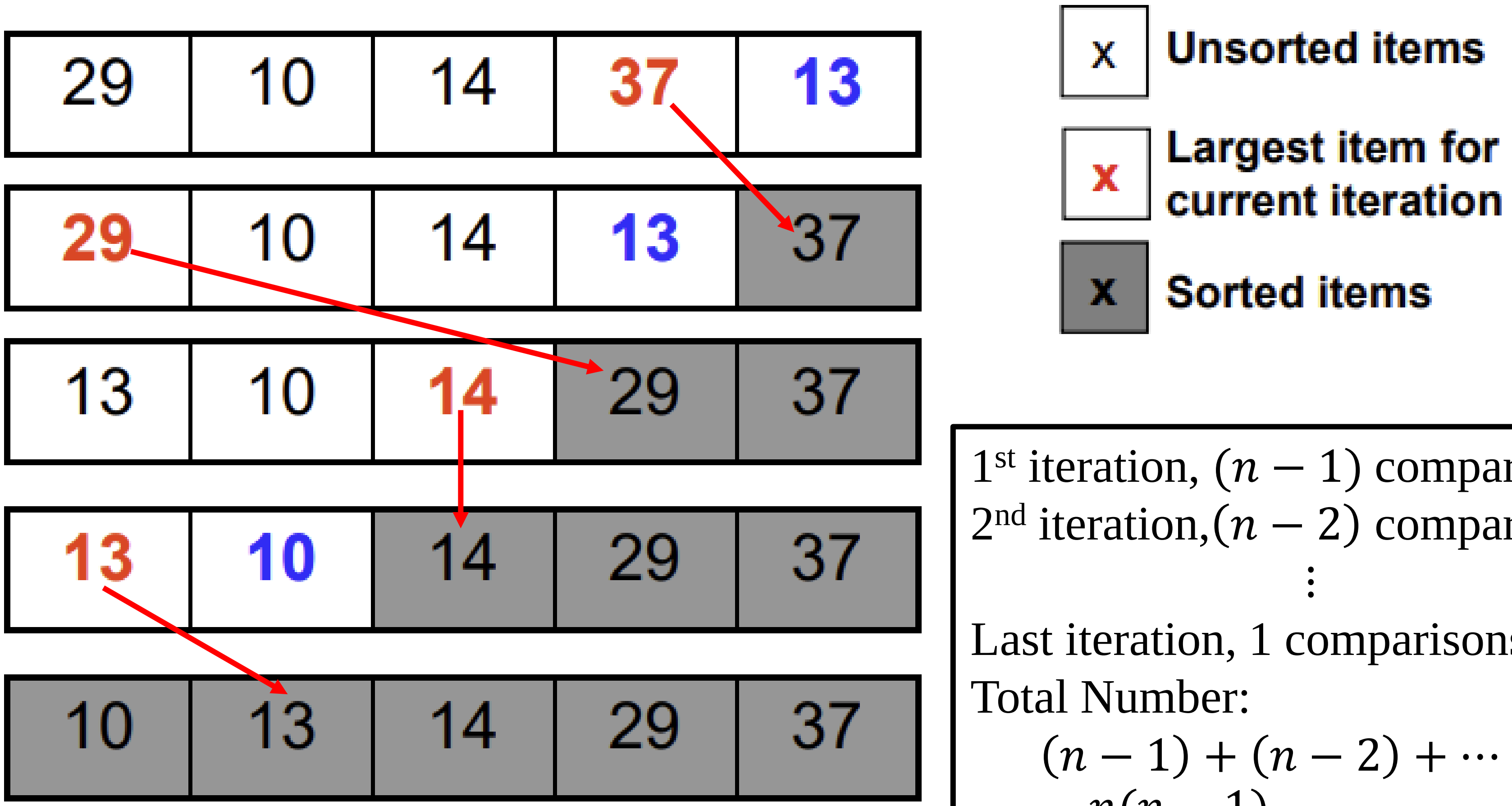
- In CS, sorting algorithm is an algorithm that puts elements of a list into an order. The most frequently used orders are numerical order and lexicographical order, and either ascending or descending.

- Iterative sorting algorithms
  - Bubble Sort – A horrible $\mathcal{O}(n^2)$ algorithm
  - Selection Sort – A better $\mathcal{O}(n^2)$ algorithm
  - Insertion Sort – The best $\mathcal{O}(n^2)$ algorithm
  - Heap Sort – A simple $\mathcal{O}(n \log n)$ algorithm
  - Bucket Sort – Sorting in $\mathcal{O}(n)$ time

- Recursive sorting algorithm
  - Merge Sort – Another $\mathcal{O}(n \log n)$ algorithm
  - Quick Sort – A quick average $\mathcal{O}(n \log n)$ algorithm

國立臺北大學 National Taipei University

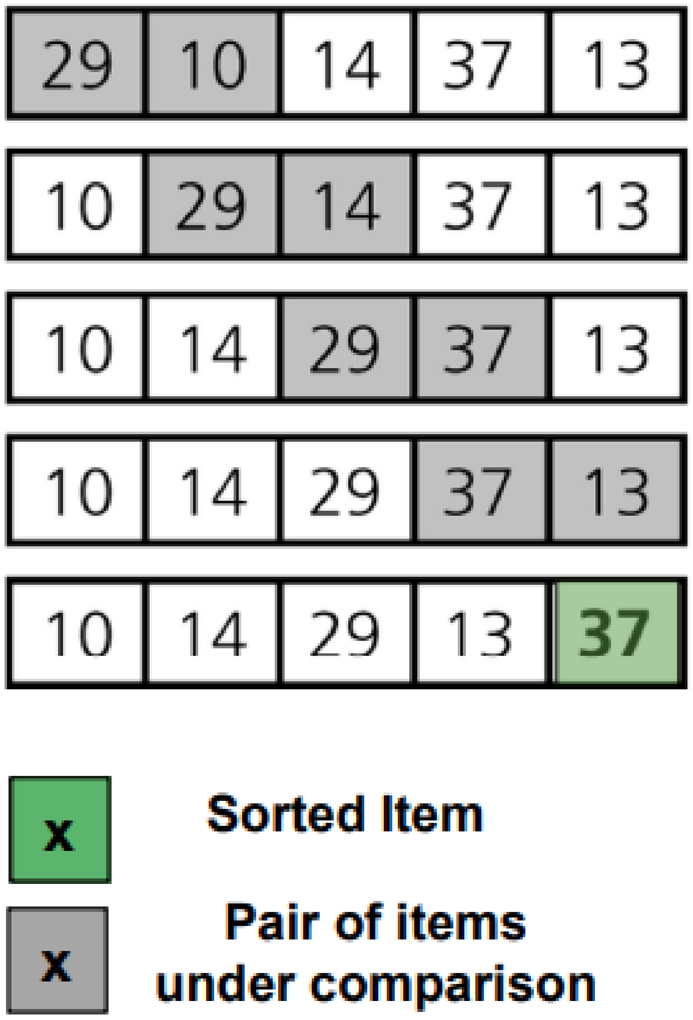
Selection Sort

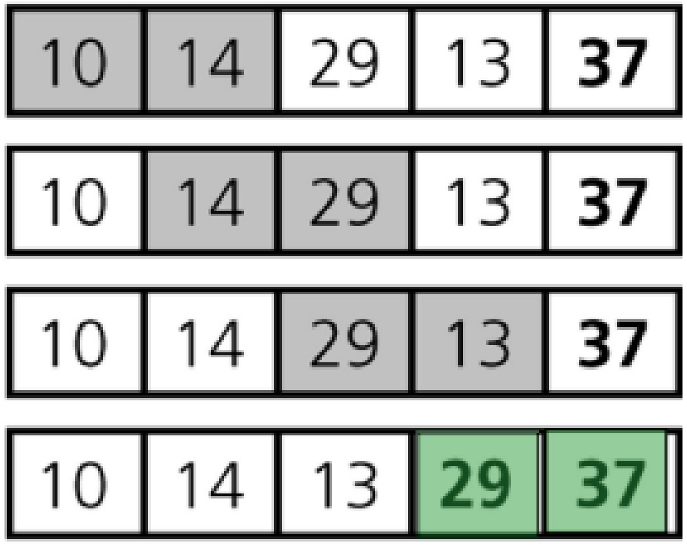


1st iteration, $(n-1)$ comparisons
2nd iteration, $(n-2)$ comparisons
$\vdots$
Last iteration, 1 comparisons
Total Number:

$$(n-1)+(n-2)+\cdots+1$$
$$=\frac{n(n-1)}{2}=\mathcal{O}(n^2)$$

Bubble Sort





Outer Loop : exactly n iteration

Inner Loop:

when $i = 0 \Rightarrow (n-1)$ iterations

when $i = 1 \Rightarrow (n-2)$ iterations

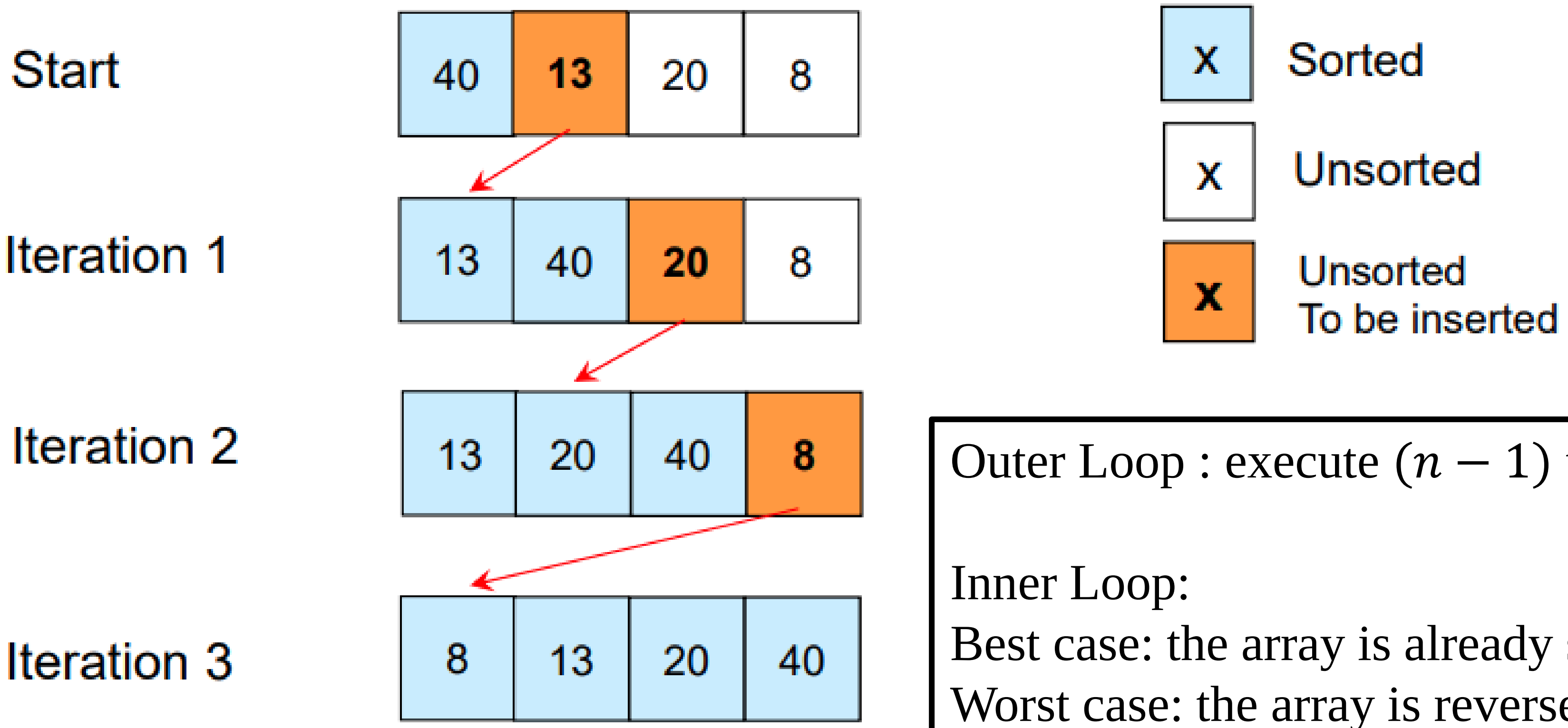
$\vdots$

when $i = (n-1) \Rightarrow 0$ iterations

Total number of iteration

$$\frac{n(n-1)}{2} = O(n^2)$$

Insertion Sort

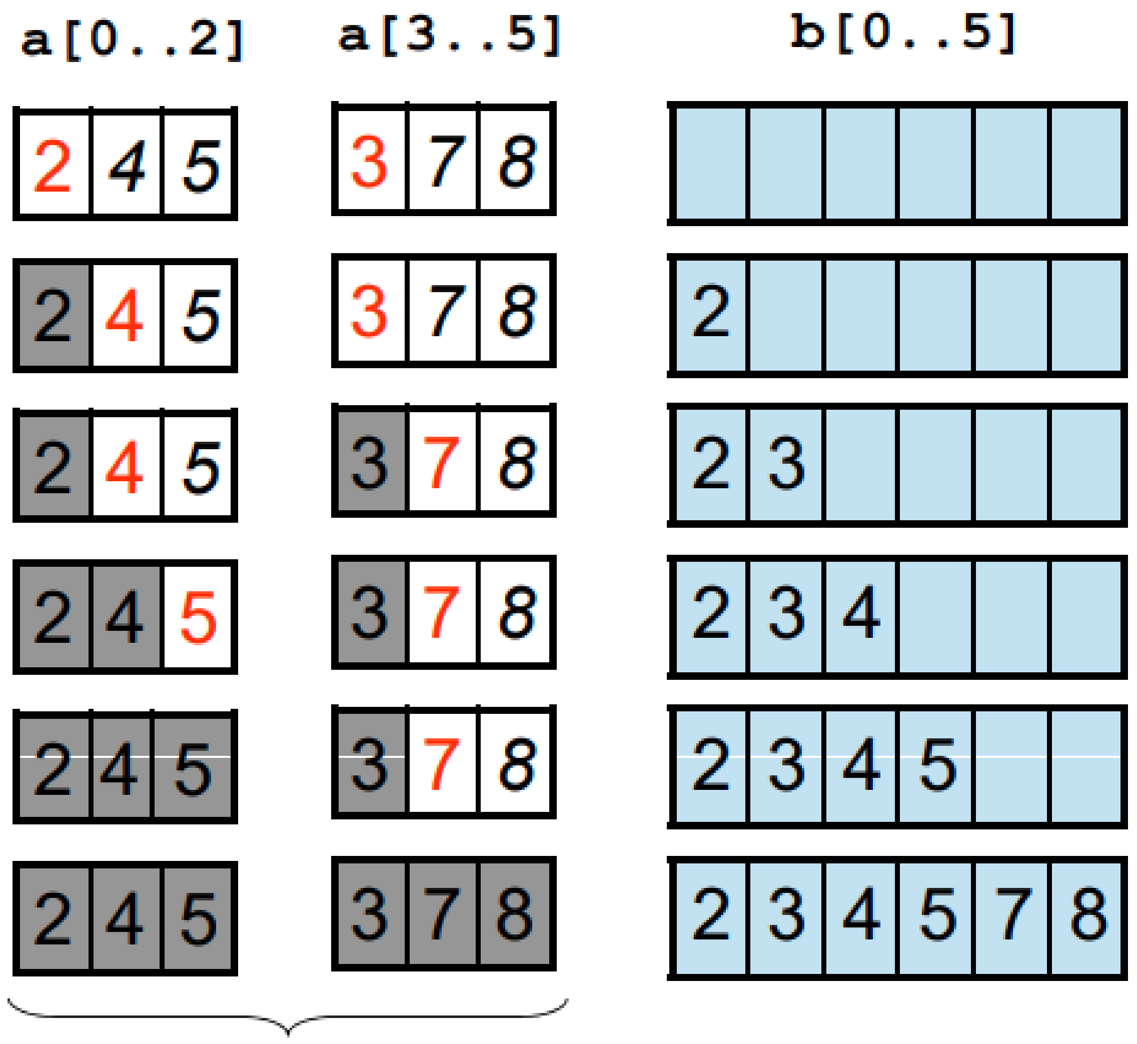


Outer Loop : execute $(n-1)$ times

Inner Loop:
Best case: the array is already sorted
Worst case: the array is reversely sorted

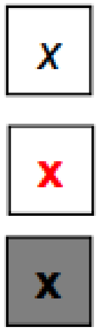
Therefore,
Best case time is $\mathcal{O}(n)$
Wort case time is $\mathcal{O}(n^2)$

Merge Sort - 0



Two sorted halves to be merged

Merged result in a temporary array



Unmerged items

Items used for comparison

Merged items

National Taipei University

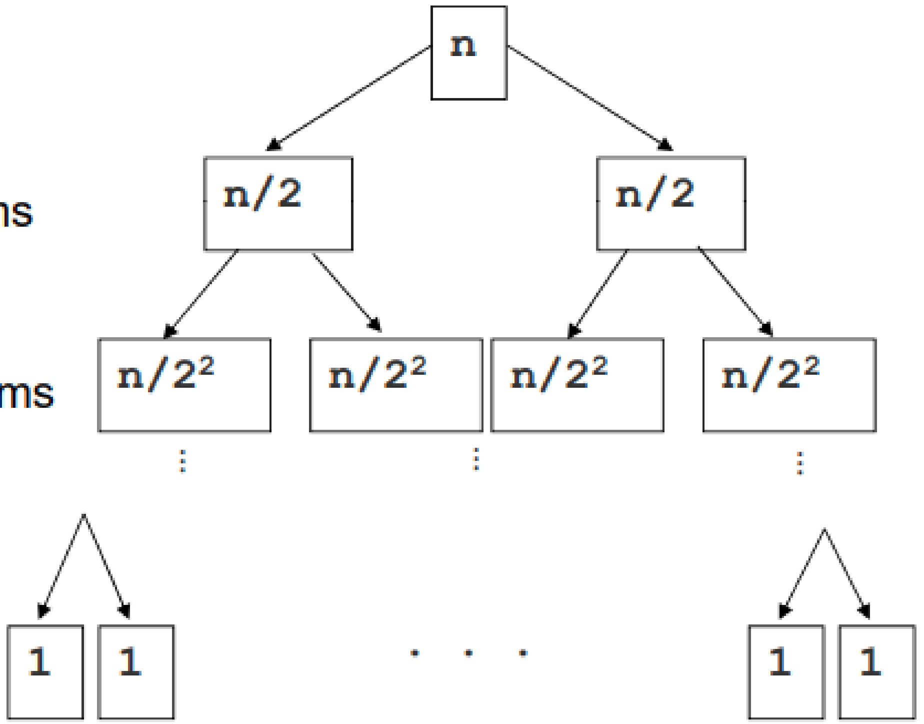
Merge Sort - 1

Level 0:
mergeSort n items

Level 1:
mergeSort n/2 items

Level 2:
mergeSort $n/2^2$ items

Level (**lg** ***n***):
mergeSort 1 item



Level 0:
1 call to mergeSort

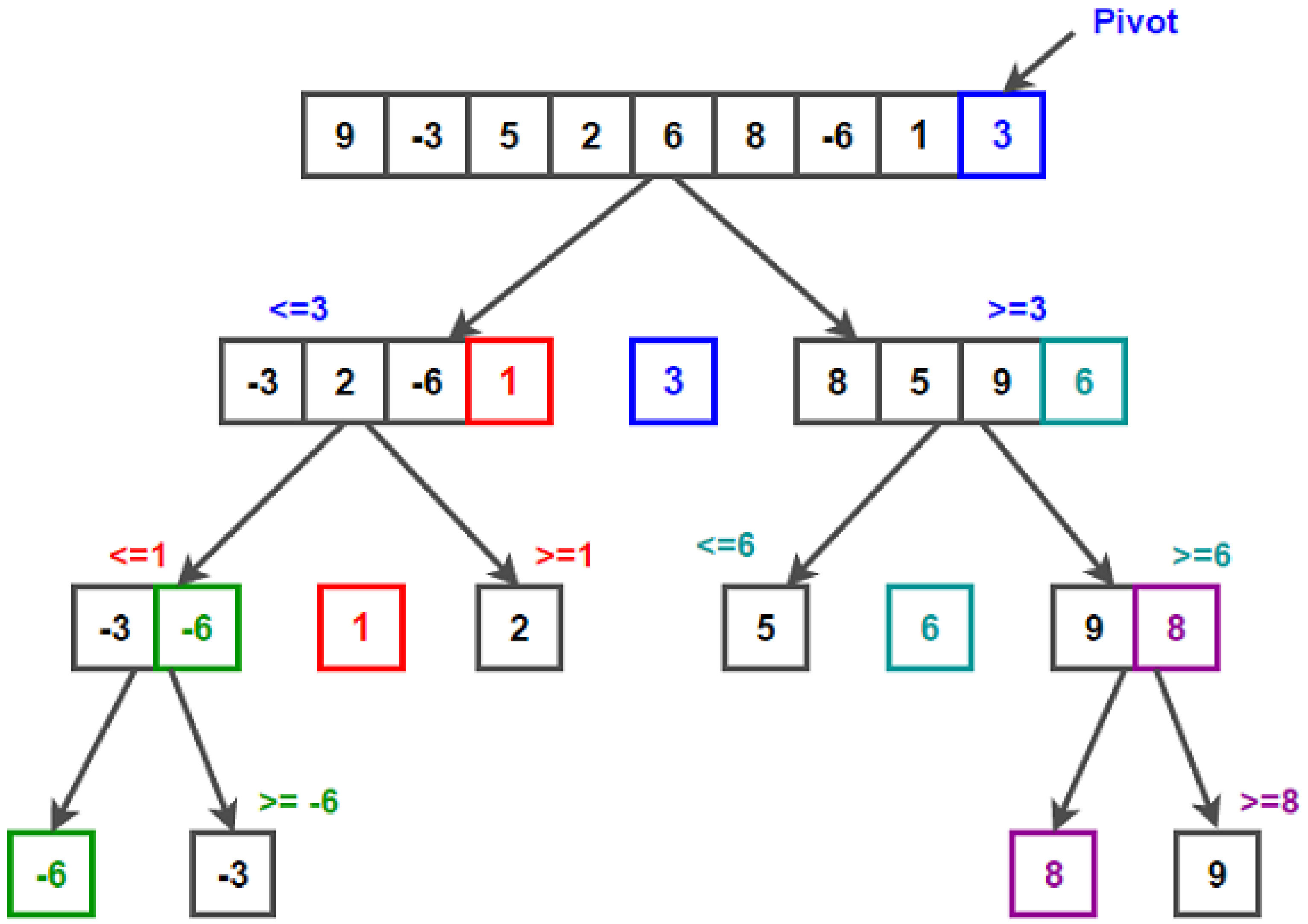
Level 1:
2 calls to mergeSort

Level 2:
$\mathbf{2^2}$ calls to mergeSort

Level (**lg** ***n***):
$\mathbf{2^{\lg n}(= n)}$ calls to mergeSort

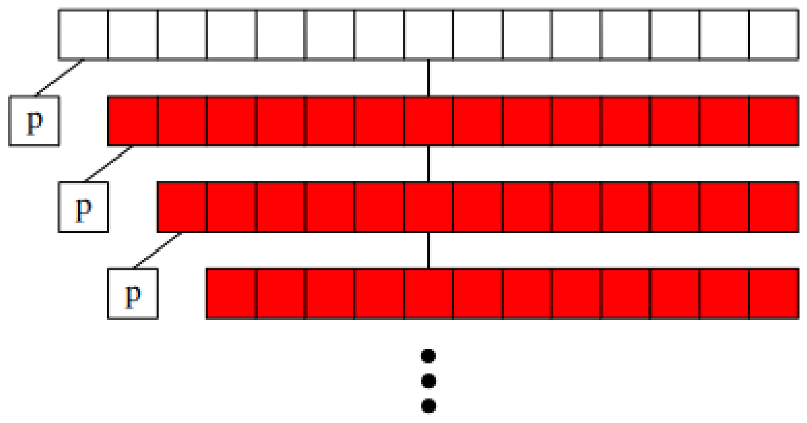
Level 0 $\Rightarrow$ 1 call to merge with n items
Level 1 $\Rightarrow$ 2 call to merge with $n/2$ items in each half
Level 2 $\Rightarrow$ 2^2 call to merge with $n/2^2$ items in each half
$\vdots$
Level $\log n \Rightarrow 2^{\log_2 n}$ call to merge with $n/2^{\log_2 n}$ items in each half

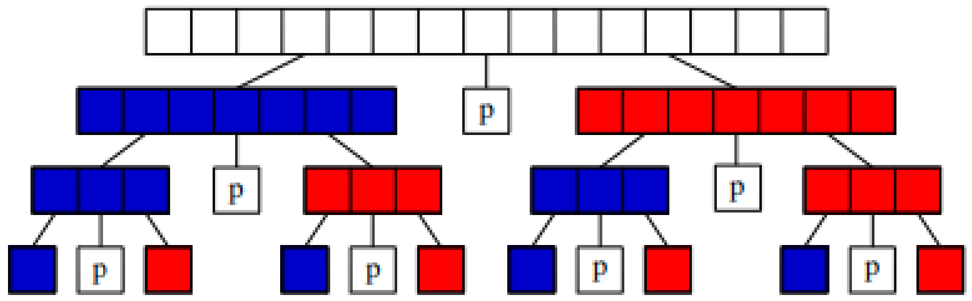
Quick Sort - 0



國立臺北大學 National Taipei University

Quick Sort - 1





Let $T(n)$ be the runtime

$$T(n) = T(k-1) + T(n-k) + n$$

To get the worst case

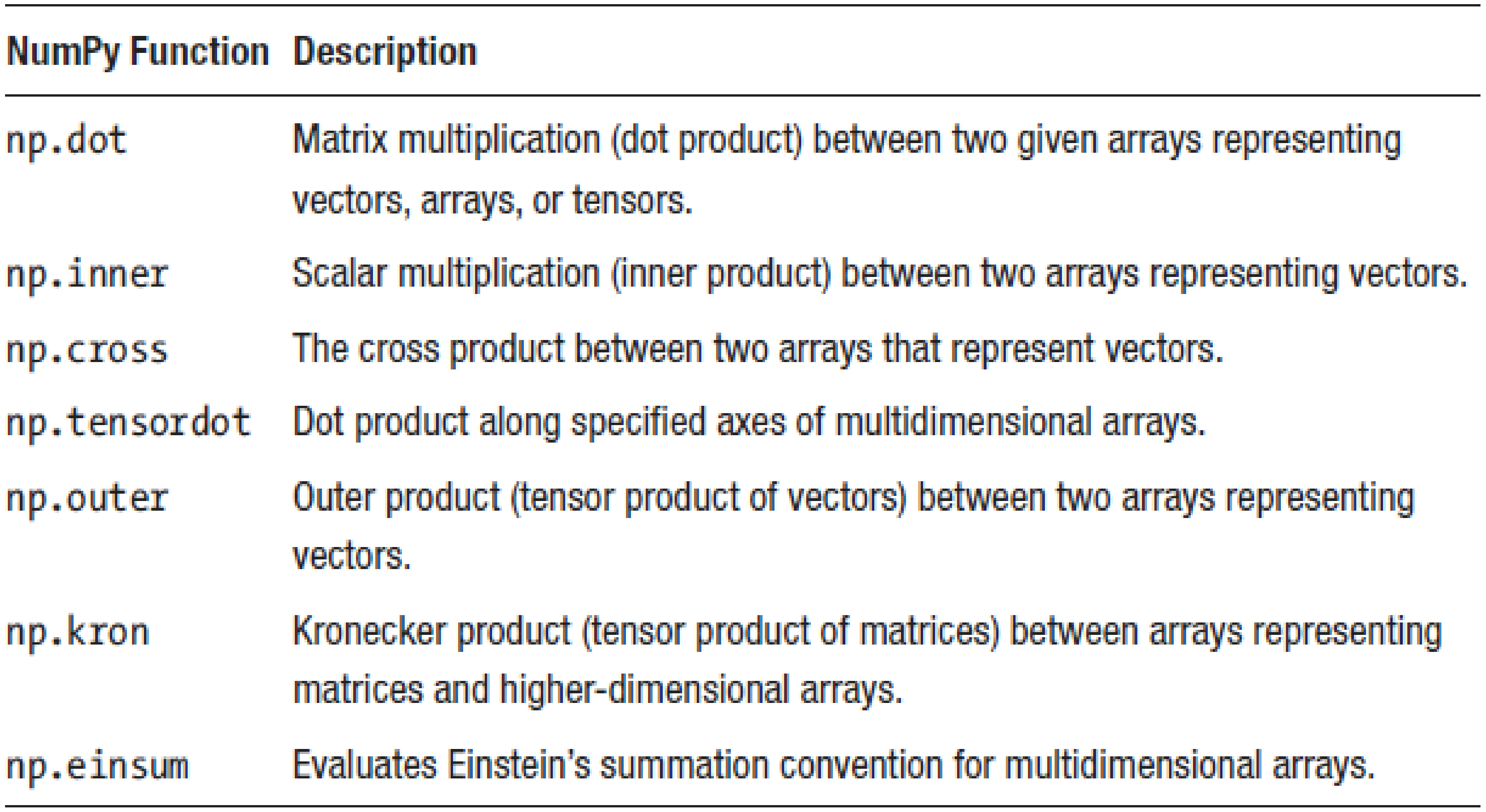
$$T(n) \le T(0) + T(n-1) + n$$
$$= T(n-1) + (n+1)$$
$$= T(n-2) + n + (n+1)$$
$$\vdots$$
$$= T(1) + \sum_{i=-1}^{n-3} (n-i)$$
$$= 1 + \big(3 + 4 + \cdots + (n+1)\big)$$
$$\le \frac{(n+1)(n+2)}{2} = \mathcal{O}(n^2)$$

Average case

$$\frac{1}{n}\sum_{k=1}^{k}\{T(k-1) + T(n-k) + n\}$$
$$= \mathcal{O}(n \log n)$$

Matrix Operation

- In NumPy, the * operator is used for elementwise multiplication. For two 2-dim arrays A and B, Python adopted the @ symbol for denoting matrix multiplication.



NumPy Function	Description
`np.dot`	Matrix multiplication (dot product) between two given arrays representing vectors, arrays, or tensors.
`np.inner`	Scalar multiplication (inner product) between two arrays representing vectors.
`np.cross`	The cross product between two arrays that represent vectors.
`np.tensordot`	Dot product along specified axes of multidimensional arrays.
`np.outer`	Outer product (tensor product of vectors) between two arrays representing vectors.
`np.kron`	Kronecker product (tensor product of matrices) between arrays representing matrices and higher-dimensional arrays.
`np.einsum`	Evaluates Einstein's summation convention for multidimensional arrays.

Inner product

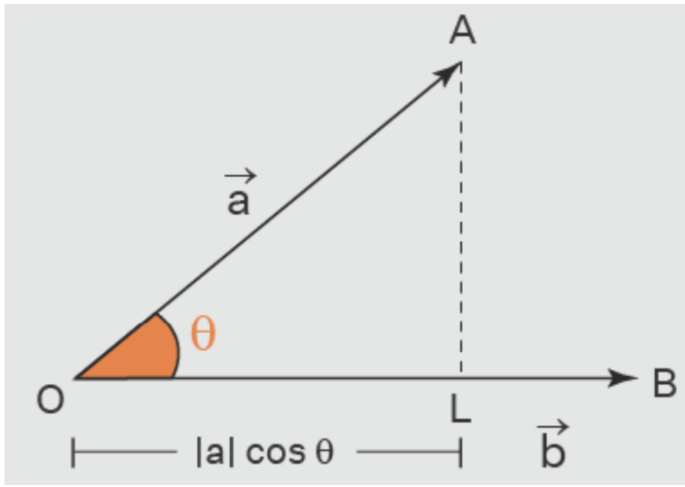
- If vectors are identified with column vectors, the dot product can also be written as a matrix product.

$$\boldsymbol{a} \cdot \boldsymbol{b} = \boldsymbol{a}^{\intercal}\boldsymbol{b}$$

- Angle between vectors can be defined as

$$\boldsymbol{\theta} = \cos^{-1}\left(\frac{\boldsymbol{a} \cdot \boldsymbol{b}}{||\boldsymbol{a}||\,||\boldsymbol{b}||}\right)$$

$$\boldsymbol{a} \cdot \boldsymbol{b} = ||\boldsymbol{a}||\,||\boldsymbol{b}|| \cos(\boldsymbol{\theta})$$



- Vector projection

$$\mathrm{proj}_{\boldsymbol{b}}\boldsymbol{a} = \frac{\boldsymbol{a} \cdot \boldsymbol{b}}{||\boldsymbol{b}||^2}\boldsymbol{b} = \left(\boldsymbol{a} \cdot \frac{\boldsymbol{b}}{||\boldsymbol{b}||}\right)\frac{\boldsymbol{b}}{||\boldsymbol{b}||}$$

Kronecker Product - 0

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix},$$

more explicitly:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix}
a_{11}b_{11} & a_{11}b_{12} & \cdots & a_{11}b_{1q} & \cdots & \cdots & a_{1n}b_{11} & a_{1n}b_{12} & \cdots & a_{1n}b_{1q} \\
a_{11}b_{21} & a_{11}b_{22} & \cdots & a_{11}b_{2q} & \cdots & \cdots & a_{1n}b_{21} & a_{1n}b_{22} & \cdots & a_{1n}b_{2q} \\
\vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\
a_{11}b_{p1} & a_{11}b_{p2} & \cdots & a_{11}b_{pq} & \cdots & \cdots & a_{1n}b_{p1} & a_{1n}b_{p2} & \cdots & a_{1n}b_{pq} \\
\vdots & \vdots & & \vdots & \ddots & & \vdots & \vdots & & \vdots \\
\vdots & \vdots & & \vdots & & \ddots & \vdots & \vdots & & \vdots \\
a_{m1}b_{11} & a_{m1}b_{12} & \cdots & a_{m1}b_{1q} & \cdots & \cdots & a_{mn}b_{11} & a_{mn}b_{12} & \cdots & a_{mn}b_{1q} \\
a_{m1}b_{21} & a_{m1}b_{22} & \cdots & a_{m1}b_{2q} & \cdots & \cdots & a_{mn}b_{21} & a_{mn}b_{22} & \cdots & a_{mn}b_{2q} \\
\vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\
a_{m1}b_{p1} & a_{m1}b_{p2} & \cdots & a_{m1}b_{pq} & \cdots & \cdots & a_{mn}b_{p1} & a_{mn}b_{p2} & \cdots & a_{mn}b_{pq}
\end{bmatrix}$$

National Taipei University

Kronecker Product - 1

The Kronecker product has the following properties (see [19])

$$\begin{aligned}
\mathbf{A}\otimes(\mathbf{B}+\mathbf{C}) &= \mathbf{A}\otimes\mathbf{B}+\mathbf{A}\otimes\mathbf{C}\\
\mathbf{A}\otimes\mathbf{B} &\neq \mathbf{B}\otimes\mathbf{A} \qquad \text{in general}\\
\mathbf{A}\otimes(\mathbf{B}\otimes\mathbf{C}) &= (\mathbf{A}\otimes\mathbf{B})\otimes\mathbf{C}\\
(\alpha_A\mathbf{A}\otimes\alpha_B\mathbf{B}) &= \alpha_A\alpha_B(\mathbf{A}\otimes\mathbf{B})\\
(\mathbf{A}\otimes\mathbf{B})^T &= \mathbf{A}^T\otimes\mathbf{B}^T\\
(\mathbf{A}\otimes\mathbf{B})(\mathbf{C}\otimes\mathbf{D}) &= \mathbf{AC}\otimes\mathbf{BD}\\
(\mathbf{A}\otimes\mathbf{B})^{-1} &= \mathbf{A}^{-1}\otimes\mathbf{B}^{-1}\\
(\mathbf{A}\otimes\mathbf{B})^{+} &= \mathbf{A}^{+}\otimes\mathbf{B}^{+}\\
\operatorname{rank}(\mathbf{A}\otimes\mathbf{B}) &= \operatorname{rank}(\mathbf{A})\operatorname{rank}(\mathbf{B})\\
\operatorname{Tr}(\mathbf{A}\otimes\mathbf{B}) &= \operatorname{Tr}(\mathbf{A})\operatorname{Tr}(\mathbf{B}) = \operatorname{Tr}(\mathbf{\Lambda}_A\otimes\mathbf{\Lambda}_B)\\
\det(\mathbf{A}\otimes\mathbf{B}) &= \det(\mathbf{A})^{\operatorname{rank}(\mathbf{B})}\det(\mathbf{B})^{\operatorname{rank}(\mathbf{A})}\\
\{\operatorname{eig}(\mathbf{A}\otimes\mathbf{B})\} &= \{\operatorname{eig}(\mathbf{B}\otimes\mathbf{A})\} \qquad \text{if } \mathbf{A},\mathbf{B} \text{ are square}\\
\{\operatorname{eig}(\mathbf{A}\otimes\mathbf{B})\} &= \{\operatorname{eig}(\mathbf{A})\operatorname{eig}(\mathbf{B})^T\}\\
&\qquad \text{if } \mathbf{A},\mathbf{B} \text{ are symmetric and square}\\
\operatorname{eig}(\mathbf{A}\otimes\mathbf{B}) &= \operatorname{eig}(\mathbf{A})\otimes\operatorname{eig}(\mathbf{B})
\end{aligned}$$

Application - 0

- Sierpinski Triangle matrix

$$\boldsymbol{H}_k = \begin{bmatrix} \boldsymbol{H}_{k-1} & \boldsymbol{H}_{k-1} \\ \boldsymbol{H}_{k-1} & \boldsymbol{O}_{k-1} \end{bmatrix}$$

can be expressed by

$$\boldsymbol{H}_k = \boldsymbol{H}_1 \otimes \boldsymbol{H}_{k-1}$$

- Since

$$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{C}^\intercal \\ \boldsymbol{C} & \boldsymbol{D} \end{bmatrix} = \begin{bmatrix} \boldsymbol{A}^{-1} + \boldsymbol{A}^{-1}\boldsymbol{C}^\intercal\boldsymbol{S}^{-1}\boldsymbol{C}\boldsymbol{A}^{-1} & -\boldsymbol{A}^{-1}\boldsymbol{C}^\intercal\boldsymbol{S}^{-1} \\ -\boldsymbol{S}^{-1}\boldsymbol{C}\boldsymbol{A}^{-1} & \boldsymbol{S}^{-1} \end{bmatrix}$$

where

$$\boldsymbol{S} = \boldsymbol{D} - \boldsymbol{C}\boldsymbol{A}^{-1}\boldsymbol{C}^\intercal$$

- The inverse of $\boldsymbol{H}_k$

$$\boldsymbol{H}_k^{-1} = \begin{bmatrix} \boldsymbol{O}_{k-1} & \boldsymbol{H}_{k-1}^{-1} \\ \boldsymbol{H}_{k-1}^{-1} & -\boldsymbol{H}_{k-1}^{-1} \end{bmatrix}$$

can be expressed by

$$\boldsymbol{H}_k^{-1} = \boldsymbol{H}_1^{-1} \otimes \boldsymbol{H}_k^{-1}$$

Application - 1

- Consider Varying Coefficient model (VCM) with the form

$$Y = a_1(U) + a_2(U)X$$

adopt the local polynomial approach for estimating functional coefficient, for $k = 1,2$

$$a_k(U) \approx \sum_{j=0}^{p} \frac{a^{(j)}(u_0)}{j!}(U - u_o)^j = \sum_{j=0}^{p} \beta_{k,j}(u_0)(U - u_o)^j$$

- Then the estimation is carried out by weighted least squares

$$\min_{\beta} \sum_{i=1}^{n} \left\{ Y_i - \left[\sum_{j=0}^{p} \beta_{k,j}(u_0)(U - u_o)^j \right] X_i \right\} K_h(U_i - u_0)$$

yield

$$\hat{\beta} = \left(\hat{\beta}_{1,0}, \cdots, \hat{\beta}_{1,p}, \hat{\beta}_{2,0}, \cdots, \hat{\beta}_{2,p}\right)^{\intercal}$$

$$\left(\hat{a}_1(u_0), \hat{a}_1(u_0)\right)^{\intercal} = \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \right) \hat{\beta}$$