Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)

Systems Software &
Architecture Lab.

Seoul National University

Jan. 6 – 17, 2020

Python for Data Analytics

NumPy II

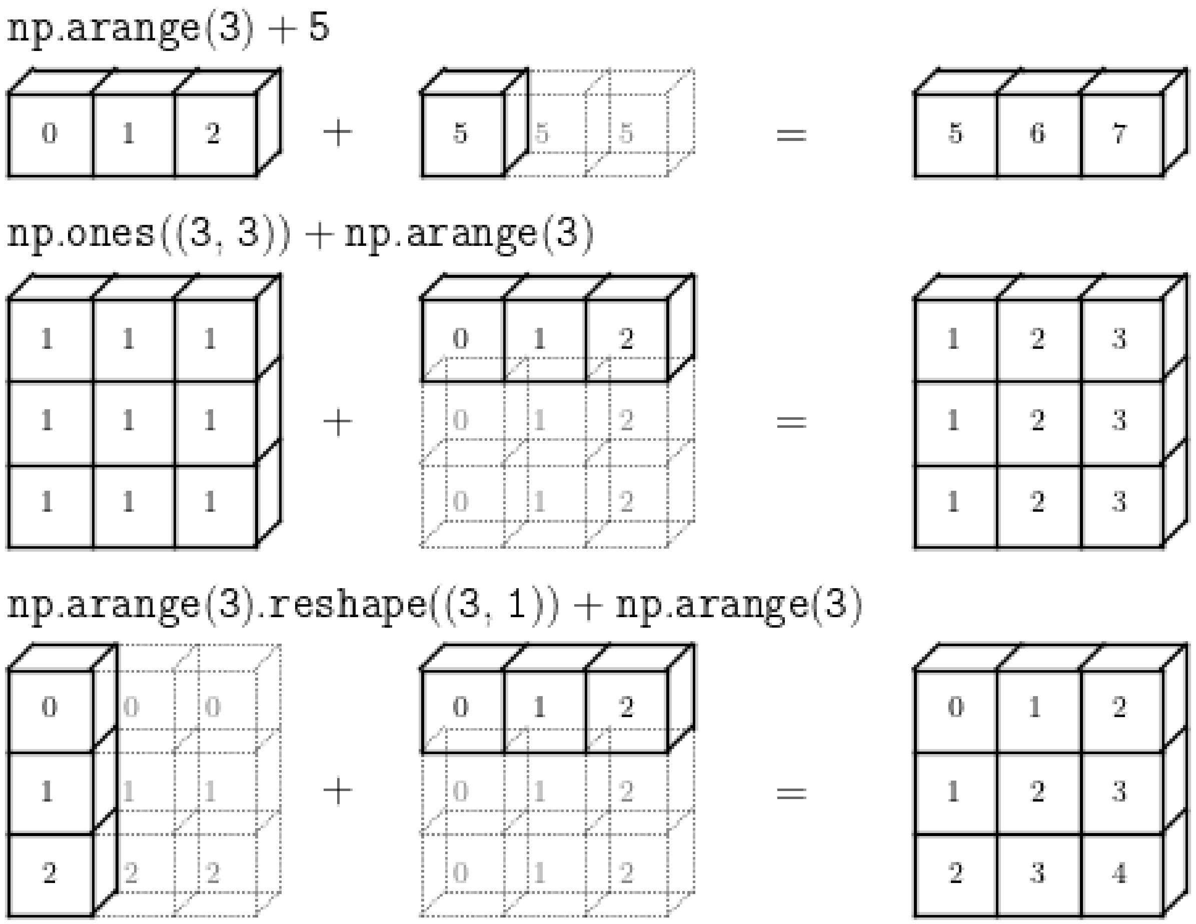
Outline

- What is NumPy?
- Creating Arrays
- Manipulating Arrays
- Array Broadcasting
- Statistical Operations
- Matrix Operations

Array Broadcasting

Broadcasting

- Allows arithmetic operations on arrays with different shapes
- The smaller array is "broadcast" across the larger array so that they have compatible shapes



http://astroml.org/book_figures/appendix/fig_broadcast_visual.html

Broadcasting Rule

- The size of the trailing axes for both arrays in an operation must either be the same size or one of them must be one

Image	(3d array)	256 x	256 x	3
Scale	(1d array)			3
Result	(3d array)	256 x	256 x	3

A	(4d array)	8 x	1 x	6 x	1
B	(3d array)		7 x	1 x	5
Result	(4d array)	8 x	7 x	6 x	5

Broadcasting Example (1)

```
>>> a = np.array([1, 2, 3])
>>> b = 2
>>> a * b
array([2, 4, 6])
>>> a = np.array([[ 0.0,  0.0,  0.0],
...                [10.0, 10.0, 10.0],
...                [20.0, 20.0, 20.0],
...                [30.0, 30.0, 30.0]])
>>> b = array([1.0, 2.0, 3.0])
>>> a + b
array([[  1.,   2.,   3.],
       [ 11.,  12.,  13.],
       [ 21.,  22.,  23.],
       [ 31.,  32.,  33.]])
```

```
a:              3
b:              1
result:         3

a:      4   x   3
b:              3
result: 4   x   3
```

Broadcasting Example (2)

```
>>> a = np.array([0.0, 10.0, 20.0, 30.0])
>>> b = np.array([1.0, 2.0, 3.0])
>>> a[:, np.newaxis] + b
array([[  1.,   2.,   3.],
       [ 11.,  12.,  13.],
       [ 21.,  22.,  23.],
       [ 31.,  32.,  33.]])
>>> a = np.arange(12).reshape(4, 3)
>>> b = np.array([1, 2, 3, 4])
>>> a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast
together with shapes (4,3) (4,)
```

Increase a dimension:
(4,) → (4, 1)

```
a:      4  x  1
b:            3
result: 4  x  3
```

```
a:      4  x  3
b:            4
result:
```

Statistical Operations

Statistical Operations

- Mean

$$\boldsymbol{Mean(x) = \bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i}$$

- Median
  - The value in the middle
  - If n is an even number, take the average of the two middle values

$$\boldsymbol{Median(x) = \frac{x_{\lfloor (n+1)/2 \rfloor} + x_{\lceil (n+1)/2 \rceil}}{2}}$$

- Variance

$$\boldsymbol{Var(x) = \sigma^2 = \frac{1}{n}\sum_{i=1}^{n} (x_i - \bar{x})^2}$$

- Standard deviation

$$\boldsymbol{Std(x) = \sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (x_i - \bar{x})^2}}$$

sum() and prod()

- *a*.sum([*axis*], [*dtype*], …)
  - Return the sum of the array elements over the given *axis*
  - If *axis* is not given, the sum of all the elements is computed
  - Similar to *np*.sum(*a*, ...)
- *a*.prod([*axis*], [*dtype*], …)
  - Return the product of the array elements over a given *axis*
  - If *axis* is not given, the product of all the elements is computed
  - Similar to *np*.prod(*a*, ...)

```
>>> a = np.arange(1, 11)
>>> a.sum()
55
>>> a.prod()
3628800
>>> x = np.arange(1,13).reshape(3,4)
>>> x
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> x.sum(axis=0) # along the rows
array([15, 18, 21, 24])
>>> x.sum(axis=1) # along the cols
array([10, 26, 42])
```

mean() and var()

- *a*.mean([*axis*], [*dtype*], …)
  - Return the average of the array elements over the given *axis*
  - If *axis* is not given, compute the mean of the flattened array
  - Similar to *np*.mean(*a*, ...)
- *a*.var([*axis*], [*dtype*], …)
  - Return the variance of the array elements over a given *axis*
  - If *axis* is not given, compute the variance of the flattened array
  - Similar to *np*.var(*a*, ...)

```
>>> a = np.arange(1, 11)
>>> a.mean()
5.5
>>> a.var()
8.25
>>> x = np.arange(1,13).reshape(3,4)
>>> x
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> x.mean()
6.5
>>> x.var
11.916666666666666
```

std() and median()

- *a*.std([*axis*], [*dtype*], …)
  - Return the standard deviation of the array elements over the given *axis*
  - If *axis* is not given, compute the standard deviation of the flattened array
  - Similar to *np*.std(*a*, ...)
- *np*.median([*axis*], [*dtype*], …)
  - Return the median along the given *axis*
  - If *axis* is not given, compute the median of the flattened array
  - No *a*.median(...) form available!

```
>>> a = np.arange(1, 11)
>>> a.std()
2.8722813232690143
>>> a.median()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'numpy.ndarray' object
has no attribute 'median'
>>> np.median(a)
5.5
x = np.arange(1,13).reshape(3,4)
>>> x.std()
3.452052529534663
>>> np.median(x)
6.5
```

Why NumPy?

- NumPy statistics is much easier than nested list statistics
- Python List vs. Numpy.array's sum() of 2D data

```
>>> a = np.arange(1,10).reshape(3,3)
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> a.sum()
45
```

```
def nested_sum(L):
    sum = 0
    for i in L:
        if isinstance(i, list):
            sum += nested_sum(i)
        else:
            sum += i
    return sum;
>>> b = [[1,2,3], [4,5,6], [7,8,9]]
>>> nested_sum(b)
45
```

Covariance

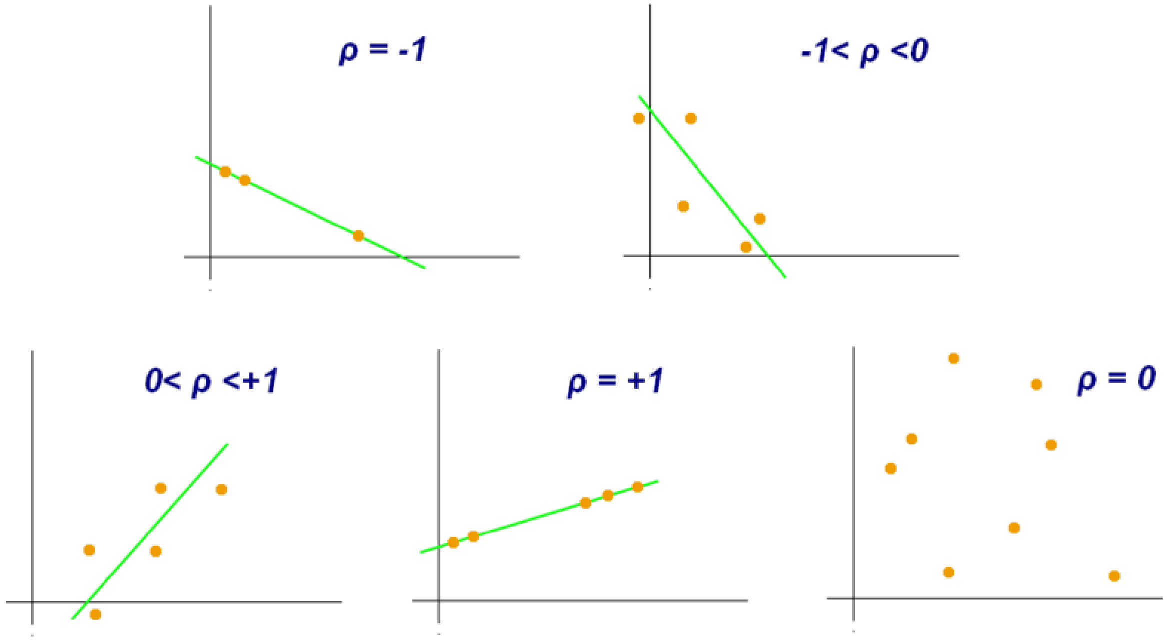
- Covariance (공분산)
  - A measure of the joint variability of two random variables

$$Cov(X, Y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

  - If the greater values of one variable mainly correspond with the greater values of the other variable, and the same holds for the lesser values (i.e., the variables tend to show similar behavior), the covariance is positive
  - In the opposite case, the covariance is negative
  - The sign of the covariance therefore shows the tendency in the linear relationship between the variables

Correlation Coefficient

- Pearson (product-moment) correlation coefficient (상관계수), *r*
  - A measure of correlation between two variables X and Y
  - -1 ≤ r ≤ where
    0: no linear correlation,
    1: total positive correlation,
    -1: total negative correlation



$$r = \frac{Cov(X,Y)}{\sigma_X \sigma_Y}$$

$$= \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

cov() and corrcoef()

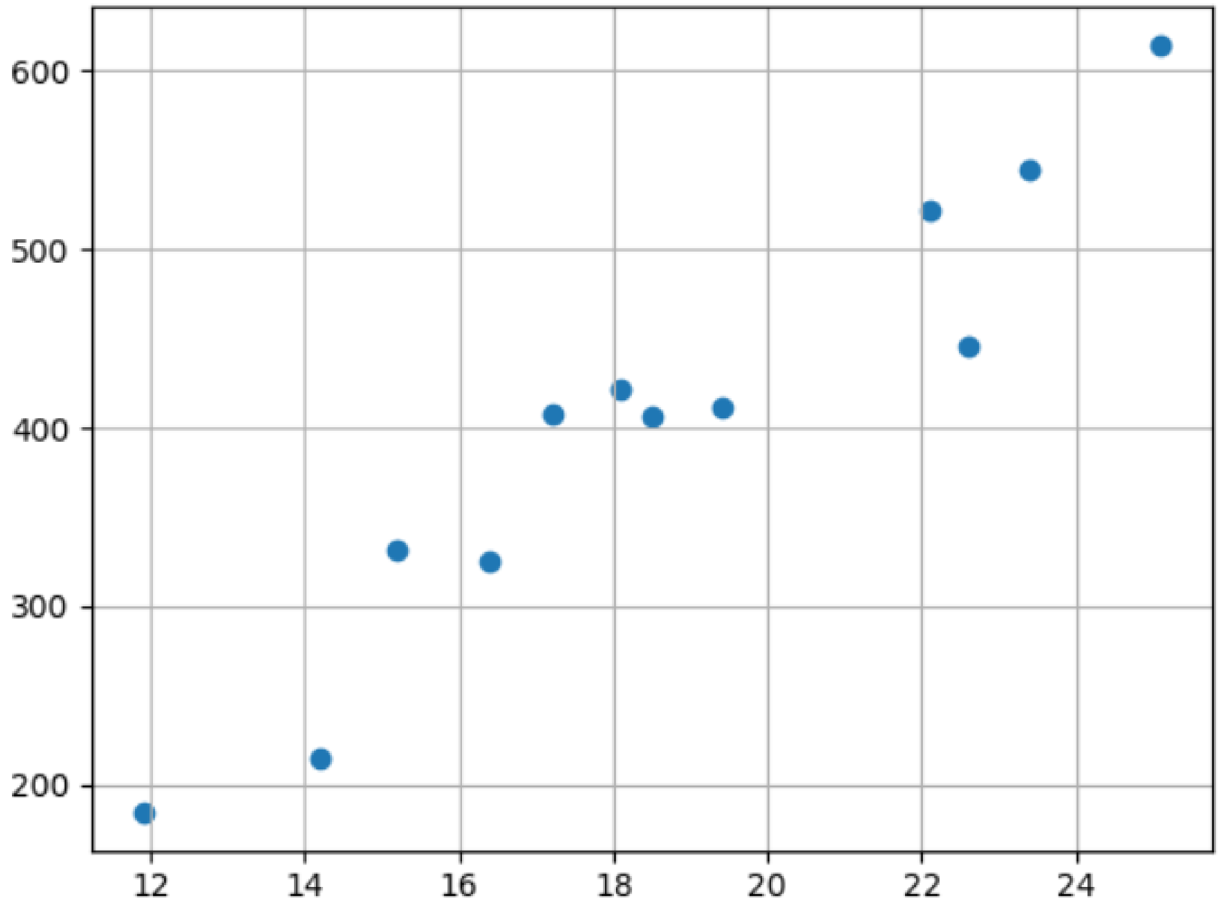
- *np*.cov(*m*, [*bias*], ...)
  - Estimate a covariance matrix, given data and weights
  - Covariance indicates the level to which two variables vary together
  - For N-dimensional samples, $X = [x_1, x_2, ..., x_n]^T$, the covariance matrix element C_{ij} is the covariance of x_i and x_j. (C_{ii} is the variance of x_i)
  - *m*: A 1-D or 2-D array containing multiple variables and observations
  - *bias*: If False, normalization is by $(N - 1)$, otherwise by N (default: False)

- *a*.corrcoef(*x*, ...)
  - Return Pearson product-moment correlation coefficients
  - The relationship between the correlation coefficient matrix, *R*, and the covariance matrix, *C*, is:

$$R_{ij} = \frac{C_{ij}}{\sqrt{C_{ii} * C_{jj}}}$$

Example: Temperature vs. Ice Cream Sales

Temperature (°C)	Ice Cream Sales
14.2	$215
16.4	$325
11.9	$185
15.2	$332
18.5	$406
22.1	$522
19.4	$412
25.1	$614
23.4	$544
18.1	$421
22.6	$445
17.2	$408



Example: Temperature vs. Ice Cream Sales

```python
t = np.array([14.2, 16.4, 11.9, 15.2,
        18.5, 22.1, 19.4, 25.1, 23.4,
        18.1, 22.6, 17.2])
s = np.array([215, 325, 185, 332, 406,
        522, 412, 614, 544, 421, 445, 408])
C = np.cov([t, s])
R = np.corrcoef([t, s])
print(C)
print(R)
# import matplotlib.pyplot as plt
# plt.scatter(t, s)
# plt.grid(True)
# plt.show()
```

```
[[16.08931818  484.09318182]
 [484.09318182 15886.81060606]]

[[1.         0.95750662]
 [0.95750662 1.        ]]
```

Matrix Operations

Array vs. Matrix

- Numpy matrices are strictly 2-dimensional, while numpy arrays (ndarrays) are N-dimensional
- Matrix objects are a subclass of ndarray, so they inherit all the attributes and methods of ndarrays
- Numpy matrices provide a convenient notation for matrix multiplication
  - If a and b are matrices, then a*b is their matrix product

```
>>> a = np.array([[4, 3], [2, 1]])
>>> b = np.array([[1, 2], [3, 4]])
>>> a * b
array([[4, 6],
       [6, 4]])
```

```
>>> ma = np.mat(a)
>>> mb = np.mat(b)
>>> ma * mb
matrix([[13, 20],
        [ 5,  8]])
```

Array vs. Matrix: Comparison

	`array`	`matrix`
Dimensions	Number of dimensions can be larger than 2	Exactly two dimensions
Operator *	Element-wise multiplication	Matrix multiplication
Operator @	Matrix multiplication	Matrix multiplication
`np.multiply()`	Element-wise multiplication	Element-wise multiplication
`np.dot()`	Matrix multiplication	Matrix multiplication
Handling vectors	1-dimensional	2-dimensional with 1xN (row vector) or Nx1 (column vector) shape
Attributes	`.T` (transpose)	`.T` (transpose), `.A` (`asarray()`), `.H` (conjugate transpose), `.I` (inverse)
Initialization	Can use Python sequences e.g., `array([[1,2,3], [4,5,6]])`	Additionally, can use a convenient string initializer e.g., `matrix('[1 2 3; 4 5 6]')`

Creating Matrices

- *np*.`matrix`(*data*[, *dtype*][, *copy*])
  - Return a matrix from an array-like object, or from a string of data
  - If *data* is a string, it is interpreted as a matrix with commas or spaces separating columns, and semicolons separating rows
- *np*.`mat`(*data*[, *dtype*])
  - Interpret the input as a matrix
  - Unlike `matrix()`, `mat()` does not make a copy if the input is already a matrix or an ndarray

```
>>> m = np.matrix('1 2; 3 4')
>>> m
matrix([[1, 2],
        [3, 4]])
>>> np.matrix([[1, 2], [3, 4]])
matrix([[1, 2],
        [3, 4]])
>>> y = np.array([[1,2], [3,4]])
>>> my = np.mat(y)
>>> my
matrix([[1, 2],
        [3, 4]])
>>> np.asarray(my)
array([[1, 2],
       [3, 4]])
```

Product Operations in Vector/Matrix

- For vectors
  - Inner product (벡터내적) Supported by `np.dot()` or `np.inner()`
  - Outer product (벡터외적) Supported by `np.outer()`
  - Cross product (벡터곱) Supported by `np.cross()`

- For matrices
  - Matrix multiplication (행렬곱) Supported by `np.dot()` or `np.matmult()`
  - Inner product (행렬내적) Supported by `np.inner()`

Vector Inner Product (벡터내적)

- Vector ·Vector → Scalar
- The inner product of two vectors in matrix form:

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

$$= \begin{pmatrix} a_1 & a_2 & \cdots & a_n \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

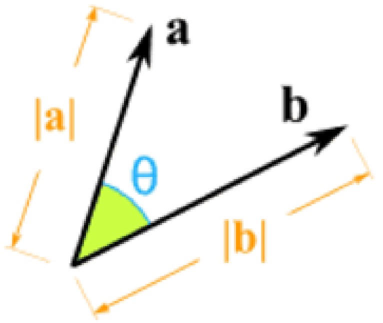
$$= a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

$$= \sum_{i=1}^{n} a_i b_i$$

$$\begin{pmatrix} a & b & c \end{pmatrix} \begin{pmatrix} 1 \\ 4 \\ 7 \end{pmatrix} = a + 4b + 7c$$

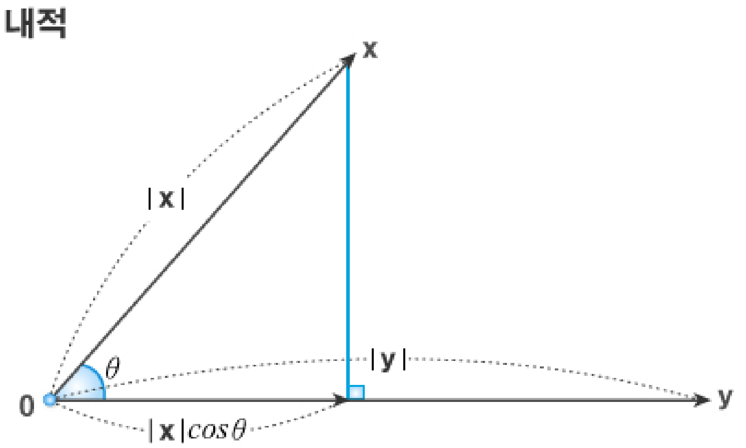
Application of Vector Inner Product

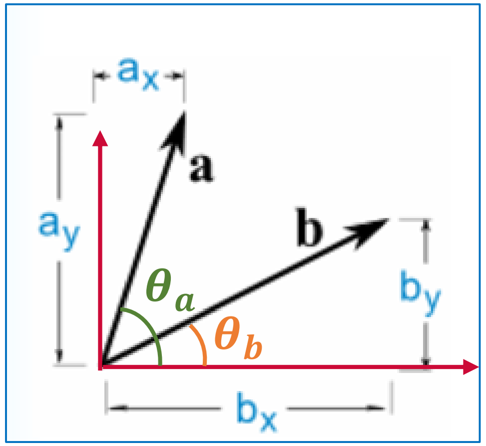
- Compute the angle between two vectors



a · **b** = |**a**| × |**b**| × cos(θ)

Where:
|**a**| is the magnitude (length) of vector **a**
|**b**| is the magnitude (length) of vector **b**
θ is the angle between **a** and **b**





$$a_x = |a|cos\theta_a \quad b_x = |b|cos\theta_b$$

$$a_y = |a|sin\theta_a \quad b_y = |b|sin\theta_b$$

$$\begin{aligned} a \cdot b &= |a||b|cos(\theta_a - \theta_b) \\ &= |a||b|(cos\theta_a cos\theta_b + sin\theta_a sin\theta_b) \\ &= |a|cos\theta_a|b|cos\theta_b + |a|sin\theta_a|a|sin\theta_b \\ &= a_x b_x + a_y b_y \end{aligned}$$

$$cos\theta = \frac{a \cdot b}{|a||b|}$$

$$\theta = arccos\left(\frac{a \cdot b}{|a||b|}\right)$$

dot() and inner()

- *np*.`dot`(*a*, *b*, …)
  - If both *a* and *b* are 1-D arrays, return the inner product of vectors
  - Otherwise, return different results depending on the input dimensions

- *np*.`inner`(*a*, *b*, …)
  - Return the inner product of vectors for 1-D arrays
  - In higher dimensions, return the sum product over the last axes
  - `== sum(a*b)`

```
>>> a = np.array([1, 2, 3], float)
>>> b = np.array([0, 1, 1], float)
>>> np.dot(a, b)
5.0
>> np.dot(a, 2)          # scalar
array([2., 4., 6.])
>>> np.inner(a, b)
5.0
>>> np.inner(3, b)       # scalar
array([0., 3., 3.])
>>> sum(a*b)
5.0
```

Vector Outer Product (벡터외적)

- Vector ·Vector → Matrix
- The outer product of two vectors in matrix form:

$$\boldsymbol{a} \otimes \boldsymbol{b} = \boldsymbol{a}\boldsymbol{b}^T$$
$$= \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} \begin{pmatrix} b_1 & b_2 & \cdots & b_n \end{pmatrix}$$
$$= \begin{pmatrix} a_1 b_1 & a_1 b_2 & \cdots & a_1 b_n \\ a_2 b_1 & a_2 b_2 & \cdots & a_2 b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_n b_1 & a_n b_2 & \cdots & a_n b_n \end{pmatrix}$$

$$\begin{pmatrix} 1 \\ 4 \\ 7 \end{pmatrix} \begin{pmatrix} a & b \end{pmatrix} = \begin{pmatrix} 1a & 1b \\ 4a & 4b \\ 7a & 7b \end{pmatrix}$$

Application of Vector Outer Product

- Matrix multiplication can be implemented using vector outer product

$$\mathbf{AB} = \begin{pmatrix} \bar{\mathbf{a}}_1 & \bar{\mathbf{a}}_2 & \cdots & \bar{\mathbf{a}}_m \end{pmatrix} \begin{pmatrix} \bar{\mathbf{b}}_1 \\ \bar{\mathbf{b}}_2 \\ \vdots \\ \bar{\mathbf{b}}_m \end{pmatrix}$$

$$= \bar{\mathbf{a}}_1 \otimes \bar{\mathbf{b}}_1 + \bar{\mathbf{a}}_2 \otimes \bar{\mathbf{b}}_2 + \cdots + \bar{\mathbf{a}}_m \otimes \bar{\mathbf{b}}_m$$

$$= \sum_{i=1}^{m} \bar{\mathbf{a}}_i \otimes \bar{\mathbf{b}}_i$$

where this time

$$\bar{\mathbf{a}}_i = \begin{pmatrix} A_{1i} \\ A_{2i} \\ \vdots \\ A_{ni} \end{pmatrix}, \quad \bar{\mathbf{b}}_i = \begin{pmatrix} B_{i1} & B_{i2} & \cdots & B_{ip} \end{pmatrix}.$$

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} a & d \\ b & e \\ c & f \end{pmatrix} = \begin{pmatrix} 1 \\ 4 \\ 7 \end{pmatrix} \otimes \begin{pmatrix} a & d \end{pmatrix} + \begin{pmatrix} 2 \\ 5 \\ 8 \end{pmatrix} \otimes \begin{pmatrix} b & e \end{pmatrix} + \begin{pmatrix} 3 \\ 6 \\ 9 \end{pmatrix} \otimes \begin{pmatrix} c & f \end{pmatrix}$$

$$= \begin{pmatrix} 1a & 1d \\ 4a & 4d \\ 7a & 7d \end{pmatrix} + \begin{pmatrix} 2b & 2e \\ 5b & 5e \\ 8b & 8e \end{pmatrix} + \begin{pmatrix} 3c & 3f \\ 6c & 6f \\ 9c & 9f \end{pmatrix}$$

$$= \begin{pmatrix} 1a + 2b + 3c & 1d + 2e + 3f \\ 4a + 5b + 6c & 4d + 5e + 6f \\ 7a + 8b + 9c & 7d + 8e + 9f \end{pmatrix}.$$

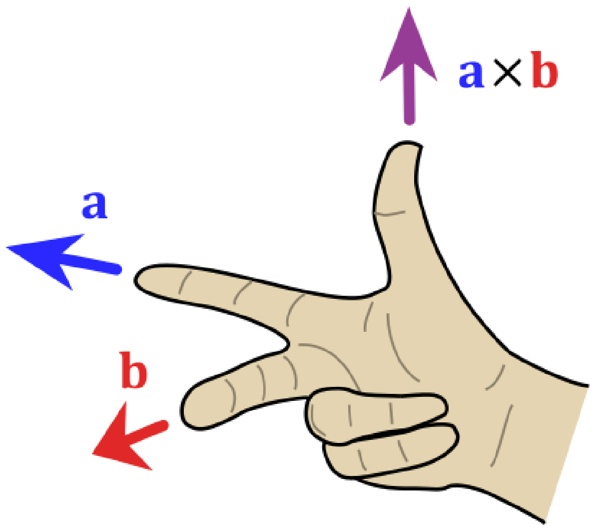
Vector Cross Product (벡터곱)

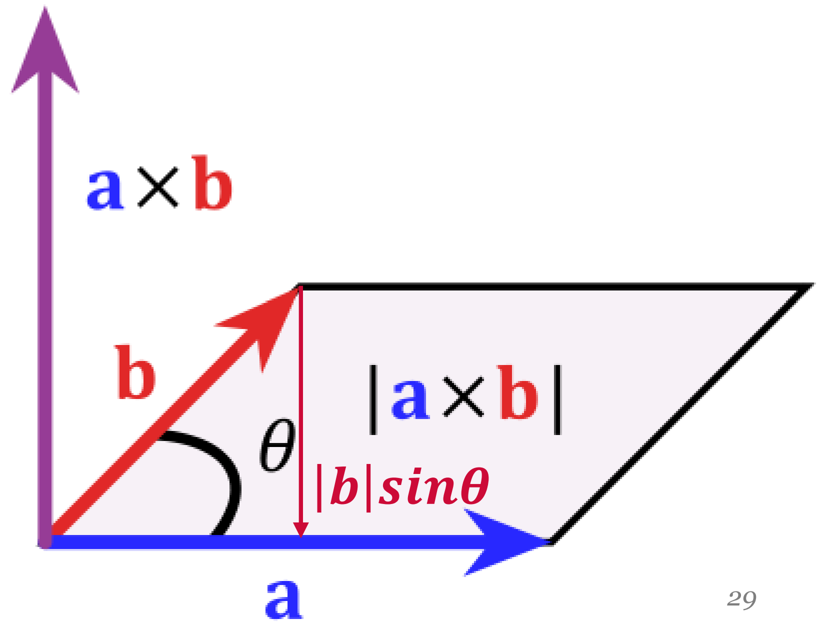
- Vector x Vector → Vector
- The cross product is defined by the formula

$$\boldsymbol{a} \times \boldsymbol{b} = |\boldsymbol{a}||\boldsymbol{b}|\boldsymbol{sin\theta}\ \boldsymbol{n}$$

where θ is the angle between ***a*** and ***b***, and ***n*** is a unit vector perpendicular to the plane containing ***a*** and ***b*** with a magnitude equal to the area of the parallelogram that the vectors span

$$|\boldsymbol{a} \times \boldsymbol{b}| = |\boldsymbol{a}||\boldsymbol{b}|\boldsymbol{sin\theta}$$





outer() and cross()

- *np*.`outer`(*a*, *b*, …)
  - Compute the outer product of two vectors

```
>>> x = np.outer(np.ones((5,)),
np.linspace(-2, 2, 5))
>>> x
array([[-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.]])

>>> x = np.array([1,4,0])
>>> y = np.array([2,2,1])
>>> np.cross(x,y)
array([ 4, -1, -6])
```

- *np*.`cross`(*a*, *b*, …)
  - Return the cross product of two vectors

Matrix Multiplication (행렬곱)

$$\mathbf{A} = \begin{pmatrix} a & b & c \\ x & y & z \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} \alpha & \rho \\ \beta & \sigma \\ \gamma & \tau \end{pmatrix},$$

their matrix products are:

$$\mathbf{AB} = \begin{pmatrix} a & b & c \\ x & y & z \end{pmatrix} \begin{pmatrix} \alpha & \rho \\ \beta & \sigma \\ \gamma & \tau \end{pmatrix} = \begin{pmatrix} a\alpha + b\beta + c\gamma & a\rho + b\sigma + c\tau \\ x\alpha + y\beta + z\gamma & x\rho + y\sigma + z\tau \end{pmatrix};$$

and

$$\mathbf{BA} = \begin{pmatrix} \alpha & \rho \\ \beta & \sigma \\ \gamma & \tau \end{pmatrix} \begin{pmatrix} a & b & c \\ x & y & z \end{pmatrix} = \begin{pmatrix} \alpha a + \rho x & \alpha b + \rho y & \alpha c + \rho z \\ \beta a + \sigma x & \beta b + \sigma y & \beta c + \sigma z \\ \gamma a + \tau x & \gamma b + \tau y & \gamma c + \tau z \end{pmatrix}.$$

dot() and matmul()

- *np*.dot(*a*, *b*, ...)
  - If both *a* and *b* are 2-D arrays, return the result of matrix multiplication
  - The use of `matmul()` or *a* @ *b* is preferred

- *np*.matmul(*a*, *b*, ...)
  - Return the matrix product of two arrays

```
>>> a = np.array([[0, 1], [2, 3]])
>>> b = np.array([2, 3])
>>> c = np.array([[1, 1], [4, 0]])
>>> np.dot(b, a)
array([ 6, 11])
>>> np.dot(a, b)
array([ 3, 13])
>>> np.matmul(a, c)
array([[ 4,  0],
       [14,  2]])
>>> c @ a
array([[2, 4],
       [0, 4]])
```

$b \bullet a$

$\rightarrow \begin{bmatrix} 2 & 3 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$

$a \bullet b$

$\rightarrow \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \end{bmatrix}$

$a \bullet c$

$\rightarrow \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 4 & 0 \end{bmatrix}$

$\mathrm{c} \bullet \mathrm{a}$

$\rightarrow \begin{bmatrix} 1 & 1 \\ 4 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$

Summary

- NumPy Core
  - Array creation, Array manipulation, Binary operations, String operations, ...
- Submodules
  - `numpy.rec`: Creating record arrays
  - `numpy.char`: Creating character arrays
  - `numpy.ctypeslib`: C-types Foreign Function Interface
  - `numpy.dual`: Optionally Scipy-accelerated routines
  - `numpy.emath`: Mathematical functions with automatic domain
  - `numpy.fft`: Discrete Fourier Transform
  - `numpy.linalg`: Linear Algebra
  - `numpy.matlib`: Matrix Library
  - `numpy.random`: Random sampling
  - `numpy.testing`: Test support