Trường Đại học Tôn Đức Thắng

Collection Data Types

Giảng viên:	Tiến sĩ Bùi Thanh Hùng
	Trưởng Lab Khoa học Phân tích dữ liệu và Trí tuệ nhân tạo
	Giám đốc chương trình Hệ thống thông tin
	Đại học Thủ Dầu Một
Email:	tuhungphe@gmail.com
Website:	https://sites.google.com/site/hungthanhbui1980/

Understanding string in build methods

Function	Description
capitalize()	It capitalizes the first letter of a string.
center(width, fillchar)	It returns a space-padded string with the original string centered to.
count(str, beg= 0,end=len(string))	It counts how many times ‘str’ occurs in a string or in the substring of a string if the starting index ‘beg’ and the ending index ‘end’ are given.
encode(encoding=’UTF-8′,errors=’strict’)	It returns an encoded string version of a string; on error, the default is to raise a ValueError unless errors are given with ‘ignore’ or ‘replace’.
endswith(suffix, beg=0, end=len(string))	It determines if a string or the substring of a string (if the starting index ‘beg’ and the ending index ‘end’ are given) ends with a suffix; it returns true if so, and false otherwise.
expandtabs(tabsize=8)	It expands tabs in a string to multiple spaces; defaults to 8 spaces per tab if the tab size is not provided.
find(str, beg=0 end=len(string))	It determines if ‘str’ occurs in a string or in the substring of a string if starting index ‘beg’ and ending index ‘end’ are given and returns the index if found, and −1 otherwise.

Understanding string in build methods

index(str, beg=0, end=len(string))	It works just like find() but raises an exception if ‘str’ not found.
isalnum()	It returns true if a string has at least one character and all characters are alphanumeric, and false otherwise.
isalpha()	It returns true if a string has at least one character and all characters are alphabetic, and false otherwise.
isdigit()	It returns true if a string contains only digits, and false otherwise.
islower()	It returns true if a string has at least one cased character and all other characters are in lowercase, and false otherwise.
isupper()	It returns true if a string has at least one cased character, and all other characters are in uppercase, and false otherwise.
len(string)	It returns the length of a string.
max(str)	It returns the max alphabetical character from the string str.

Understanding string in build methods

min(str)	It returns the min alphabetical character from the string str.
upper()	It converts lowercase letters in a string to uppercase.
rstrip()	It removes all trailing whitespace of a string.
split(str="", num=string.count(str))	It is used to split strings in Python according to the delimiter str (space if not provided any) and returns the list of substrings in Python
splitlines(num=string.count('\n'))	It splits a string at the newlines and returns a list of each line with newlines removed.

Collection Data Types

- Lists
- Dictionaries
- Set
- Tuples

- **Matrix and vector in Python**

1. Lists

A list is the Python equivalent of an array, but is resizeable and can contain elements of different types

```
xs = [3, 1, 2]
print(xs, xs[2])
print(xs[-1])
xs[2] = 'foo'
print(xs)
xs.append('bar')
print(xs)
x = xs.pop()
print(x, xs)
```

1. Lists

```
nums = list(range(5))
print(nums)
print(nums[2:4])
print(nums[2:])
print(nums[:2])
print(nums[:])
print(nums[:-1])
nums[2:4] = [8, 9]
print(nums)
```

1 Lists

```
animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print(animal)
```

```
animals = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
```

1 Lists

```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
    print(squares)
```

1 Lists

```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
    print(squares)

nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares)
```

1 Lists

```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
print(even_squares)
```

The += operator for lists

```
a = [1, 3, 5]
a += [7]
print(a)
```

```
a= [1, 3, 5, 7]
a += [“the-end”]
print(a)
```

Lists

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

2. Dictionaries

A dictionary stores (key, value) pairs, similar to a Map in Java or an object in Javascript

```
d = {'cat': 'cute', 'dog': 'furry'}
print(d['cat'])
print('cat' in d)
d['fish'] = 'wet'
print(d['fish'])
print(d.get('monkey', 'N/A'))
print(d.get('fish', 'N/A'))
del d['fish']
print(d.get('fish', 'N/A'))
```

2. Dictionaries

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal in d:
    legs = d[animal]
    print('A %s has %d legs' % (animal, legs))
```

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))
```

2. Dictionaries

```
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print(even_num_to_square)
```

Dictionary comprehensions:
These are similar to list comprehensions, but allow you to easily construct dictionaries

Constructing dictionaries from lists

If we have separate lists for the keys and values, we can combine them into a dictionary using the zip function and a dict constructor:

```
keys = ['david', 'chris', 'stewart']
vals = ['504', '637', '921']
d = dict(zip(keys, vals))
print(d)
```

Dictionary manipulation

Method	Description
clear()	Remove all items form the dictionary.
copy()	Return a shallow copy of the dictionary.
fromkeys(seq[, v])	Return a new dictionary with keys from seq and value equal to v (defaults to None).
get(key[,d])	Return the value of key. If key doesnot exit, return d (defaults to None).
items()	Return a new view of the dictionary's items (key, value).

Dictionary manipulation

Method	Description
keys()	Return a new view of the dictionary's keys.
pop(key[,d])	Remove the item with key and return its value or d if key is not found. If d is not provided and key is not found, raises KeyError.
popitem()	Remove and return an arbitary item (key, value). Raises KeyError if the dictionary is empty.
setdefault(key[,d])	If key is in the dictionary, return its value. If not, insert key with a value of d and return d (defaults to None).
update([other])	Update the dictionary with the key/value pairs from other, overwriting existing keys.
values()	Return a new view of the dictionary's values

Dictionary manipulation

Method	Description
all()	Return True if all keys of the dictionary are true (or if the dictionary is empty).
any()	Return True if any key of the dictionary is true. If the dictionary is empty, return False.
len()	Return the length (the number of items) in the dictionary.
cmp()	Compares items of two dictionaries.
sorted()	Return a new sorted list of keys in the dictionary.

3. Set

- A set is an unordered collection of distinct elements.
- Curly braces or the set() function can be used to create sets.
- Note: to create an empty set you have to use set(), not {}; the latter creates an empty dictionary.

3. Set

```
animals = {'cat', 'dog‘, 'cat‘, 'cat'}
print(animals)
print('cat' in animals)
print('fish' in animals)
animals.add('fish')
print('fish' in animals)
print(len(animals))
animals.add('cat')
print(len(animals))
animals.remove('cat')
print(len(animals))
```

3. Set

```
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
        print('#%d: %s' % (idx + 1, animal))
```

```
print('The solution are’, x, y)
print('The solution are {0} and {1}'.format(sol1,sol2))
```

3. Set

```
from math import sqrt
nums = {int(sqrt(x)) for x in range(30)}
print(nums)
```

Set

FUNCTION	DESCRIPTION
add()	Adds an element to a set
remove()	Removes an element from a set. If the element is not present in the set, raise a KeyError
clear()	Removes all elements form a set
copy()	Returns a shallow copy of a set
pop()	Removes and returns an arbitrary set element. Raise KeyError if the set is empty
update()	Updates a set with the union of itself and others
union()	Returns the union of sets in a new set
difference()	Returns the difference of two or more sets as a new set

Set

difference_update()	Removes all elements of another set from this set
discard()	Removes an element from set if it is a member. (Do nothing if the element is not in set)
intersection()	Returns the intersection of two sets as a new set
intersection_update()	Updates the set with the intersection of itself and another
isdisjoint()	Returns True if two sets have a null intersection
issubset()	Returns True if another set contains this set
issuperset()	Returns True if this set contains another set
symmetric_difference()	Returns the symmetric difference of two sets as a new set
symmetric_difference_update()	Updates a set with the symmetric difference of itself and another

4. Tuples

A tuple is an (immutable) ordered list of values.

A tuple is in many ways similar to a list; one of the most important differences is that tuples can be used as keys in dictionaries and as elements of sets, while lists cannot.

4. Tuple

Methods that add items or remove items are not available with tuple. Only the following two methods are available.

Method	Description
count(x)	Returns the number of items x
index(x)	Returns the index of the first item that is equal to x

4. Tuples

```
t = 12345, 54321, 'hello!‘
print(t[0])
print(t)
u = t, (1, 2, 3, 4, 5)
print(u)
t[0] = 88888    #error
v = ([1, 2, 3], [3, 2, 1])
print(v)
```

4. Tuples

```
d = {(x, x + 1): x for x in range(10)}
print(d)
t = (5, 6)
print(type(t))
print(d[t])
print(d[(1, 2)])
```

More general keys

- Python includes several built-in container types: lists, dictionaries, sets, and tuples

- Keys don't have to be strings; they can be any immutable data type, including tuples
- This might be good for representing sparse matrices, for example

```
Matrix = {}        # start a blank dictionary
Matrix[(1,0,1)]  = 0.5 # key is a tuple
Matrix[(1,1,4)]  = 0.8
```

Length of collections

len() returns the length of a tuple, list, or dictionary (or the number of characters of a string):

```
x= (“Tony”,)
print(type(x))
print(len((“Tony”,)))
print(len(“Tony”))
print(len([0, 1, 'boom']))
```

The “is” operator

- Python “variables” are really object references. The “is” operator checks to see if these references refer to the *same* object (note: could have two identical objects which are not the same object...)
- References to integer constants should be identical. References to strings may or may not show up as referring to the same object. Two identical, mutable objects are not necessarily the same object

is-operator

```
x = "hello"
y = "hello"
print(x is y)

x = [1,2]
y = [1,2]
print(x is y)


x = (1,2)
y = (1,2)
print(x is y)



x = []
print(x is not None)
```

“in” operator

For collection data types, the “in” operator determines whether something is a member of the collection (and “not in” tests if not a member):

```
team = (“David”, “Robert”, “Paul”)
print(“Howell” in team)
print(“Stewart” not in team)
```

Iteration: for ... in

To traverse a collection type, use for ... in

```
numbers = (1, 2, 3)
for i in numbers:
    print(i)
```

Copying collections

Using assignment just makes a new reference to the same collection, e.g.,

```
A = [0,1,3]

B = A
C = A[:2]

B[1] = 5

C[1] = 7

print(A, B, C)

print(A.copy())
```

5. Matrix and Vector in Python

```
import numpy as np
```

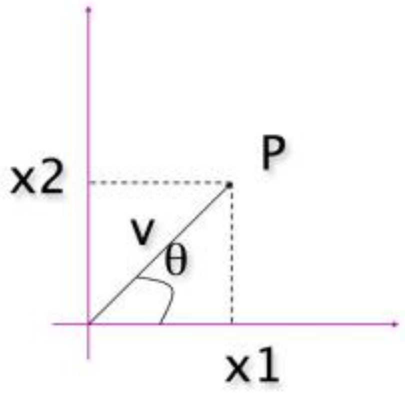
A supremely-optimized, well-maintained scientific computing package for Python.

As time goes on, you’ll learn to appreciate NumPy more and more.

Years later I’m **still** learning new things about it!

Vector

$$\mathbf{v} = (x_1, x_2)$$



Magnitude: $\| \mathbf{v} \| = \sqrt{x_1^{\,2} + x_2^{\,2}}$

If $\| \mathbf{v} \| = 1$, $\mathbf{v}$ Is a UNIT vector

$$\frac{\mathbf{v}}{\| \mathbf{v} \|} = \left(\frac{x_1}{\| \mathbf{v} \|}, \frac{x_2}{\| \mathbf{v} \|} \right)$$ Is a unit vector

Orientation: $\theta = \tan^{-1}\left(\frac{x_2}{x_1} \right)$

Vector

$$\mathbf{v}+\mathbf{w}=(x_1,x_2)+(y_1,y_2)=(x_1+y_1,x_2+y_2)$$
$$\mathbf{v}-\mathbf{w}=(x_1,x_2)-(y_1,y_2)=(x_1-y_1,x_2-y_2)$$
$$a\mathbf{v}=a(x_1,x_2)=(ax_1,ax_2)$$

Matrix

$$A_{n \times m} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & \boxed{a_{nm}} \end{bmatrix} \longleftrightarrow$$

Pixel's intensity value

Sum: $C_{n\times m} = A_{n\times m} + B_{n\times m}$ $\quad c_{ij} = a_{ij} + b_{ij}$

A and B must have the same dimensions!

Example: $\begin{bmatrix} 2 & 5 \\ 3 & 1 \end{bmatrix} + \begin{bmatrix} 6 & 2 \\ 1 & 5 \end{bmatrix} = \begin{bmatrix} 8 & 7 \\ 4 & 6 \end{bmatrix}$

Array

- ❖ A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the *rank* of the array; the *shape* of an array is a tuple of integers giving the size of the array along each dimension.
- ❖ We can initialize numpy arrays from nested Python lists, and access elements using square brackets.

Array

```
import numpy as np
a = np.array([1, 2, 3])
print(type(a))
print(a.shape)
print(a[0], a[1], a[2])
a[0] = 5
print(a)
b = np.array([[1,2,3],[4,5,6]])
print(b.shape)
print(b[0, 0], b[0, 1], b[1, 0])
```

Array

Numpy also provides many functions to create arrays:

```
a=np.zeros((3,2))
print(a)

b=np.ones((1,2))
print(b)

c=np.full((3,4),7)
print(c)

d=np.eye(4)
print(d)
e=np.random.random((4,3))
print(e)
```

Other Ways to Create Matrices and Vectors

```
v1 = np.array([1, 2, 3])
v2 = np.array([4, 5, 6])
v3 = np.array([7, 8, 9])
M = np.vstack([v1, v2, v3])
print(M)
```

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \quad v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

```
# There is also a way to do this horizontally => hstack
```

Array

```
import numpy as np
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
b = a[:2, 1:3]
print(b)
print(a[0, 1])
b[0, 0] = 77
print(a[0, 1])
```

Array

```
import numpy as np
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
row_r1 = a[1, :]
row_r2 = a[1:2, :]
print(row_r1, row_r1.shape)
print(row_r2, row_r2.shape)
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)
print(col_r2, col_r2.shape)
```

Array

```
import numpy as np
a = np.array([[1,2], [3, 4], [5, 6]])
print(a[[0, 1, 2], [0, 1, 0]])
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))
print(a[[0, 0], [1, 1]])
print(np.array([a[0, 1], a[0, 1]]))
```

Array

```
import numpy as np
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
print(a)
b = np.array([0, 2, 0, 1])
print(a[np.arange(4), b])
a[np.arange(4), b] += 10
print(a)
```

Array

```
import numpy as np
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
print(a)
b = np.array([0, 2, 0, 1])       # Create an array of indices
# Select one element from each row of a using the indices in b
print(a[np.arange(4), b])      # Prints "[ 1  6  7 11]"
 # Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10
print(a)
          #array([[11,  2,  3],
          #         [ 4,  5, 16],
          #         [17,  8,  9],
          #         [10, 21, 12]])
```

Array

```
import numpy as np
a = np.array([[1,2], [3, 4], [5, 6]])
bool_idx = (a > 2)
print(bool_idx)
print(a[bool_idx])
print(a[a > 2])            #all in one
```

Array - Datatype

```
import numpy as np
x = np.array([1, 2])
print(x.dtype)
x = np.array([1.0, 2.0])
print(x.dtype)
x = np.array([1, 2], dtype=np.int64)
print(x.dtype)
```

Array - Math

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \quad v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

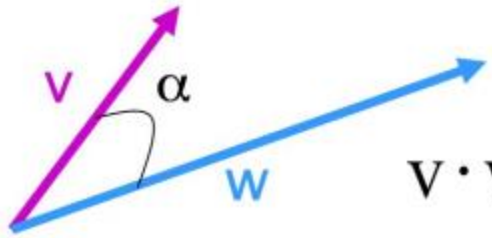
```
import numpy as np
M= np.array([[1, 2, 3],
                      [4, 5, 6],
                      [7, 8, 9]])
v= np.array([[1],[2],[3]])

print(v+v)
print( 3*v)
```

Array - Math

```
import numpy as np
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)
print(x + y)
print(np.add(x, y))
print(x - y)
print(np.subtract(x, y))
print(x * y)
print(np.multiply(x, y))
print(x / y)
print(np.divide(x, y))
print(np.sqrt(x))
```

Dot Product



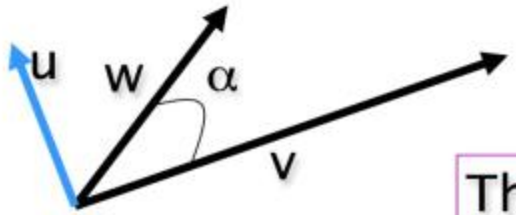
$$v \cdot w = (x_1, x_2) \cdot (y_1, y_2) = x_1 y_1 + x_2 y_2$$

The inner product is a SCALAR!

$$v \cdot w = (x_1, x_2) \cdot (y_1, y_2) = \| v \| \cdot \| w \| \cos\alpha$$

$$\text{if} \quad v \perp w, \quad v \cdot w = ? = 0$$

Cross Product



$$u = v \times w$$

The cross product is a VECTOR!

$$\text{Magnitude:} \|u\| = \|v \times w\| = \|v\|\|w\| \sin \alpha$$

Orientation:

$$u \perp v \Rightarrow u \cdot v = (v \times w) \cdot v = 0$$

$$u \perp w \Rightarrow u \cdot w = (v \times w) \cdot w = 0$$

$$\text{if} \quad v // w \,? \quad \rightarrow u = 0$$

Cross Product

$$\mathbf{i} = (1,0,0) \qquad \|\mathbf{i}\| = 1 \qquad \mathbf{i} = \mathbf{j} \times \mathbf{k}$$

$$\mathbf{j} = (0,1,0) \qquad \|\mathbf{j}\| = 1 \qquad \mathbf{j} = \mathbf{k} \times \mathbf{i}$$

$$\mathbf{k} = (0,0,1) \qquad \|\mathbf{k}\| = 1 \qquad \mathbf{k} = \mathbf{i} \times \mathbf{j}$$

$$\mathbf{u} = \mathbf{v} \times \mathbf{w} = (x_1, x_2, x_3) \times (y_1, y_2, y_3)$$

$$= (x_2 y_3 - x_3 y_2)\mathbf{i} + (x_3 y_1 - x_1 y_3)\mathbf{j} + (x_1 y_2 - x_2 y_1)\mathbf{k}$$

Matrix Multiplication

$$A_{n\times m} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix} \quad \mathbf{a}_i$$

$$B_{m\times p} = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1p} \\ b_{21} & b_{22} & \dots & b_{2p} \\ \vdots & \vdots & \vdots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mp} \end{bmatrix}$$

$$\mathbf{b}_j$$

Product:

$$C_{n\times p} = A_{n\times \boxed{m}} B_{\boxed{m}\times p}$$

$$c_{ij} = \mathbf{a}_i \cdot \mathbf{b}_j = \sum_{k=1}^{m} a_{ik} b_{kj}$$

A and B must have compatible dimensions!

$$A_{n\times n} B_{n\times n} \neq B_{n\times n} A_{n\times n}$$

Array - Math

dot function

- to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices.
- available both as a function in the numpy module and as an instance method of array objects:

```
import numpy as np
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])
v = np.array([9,10])
w = np.array([11, 12])
print(v.dot(w))
print(np.dot(v, w))
```

```
print(x.dot(v))
print(np.dot(x, v))
print(x.dot(y))
print(np.dot(x, y))
```

Array Math: Dot Multiplication

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \quad v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

```
import numpy as np
M= np.array([[1, 2, 3],
               [4, 5, 6],
               [7, 8, 9]])
v= np.array([[1],[2],[3]])
print(M.dot(v))
print(v.dot(v) )
print(v.T.dot(v))
```

Array Math: Dot Multiplication

```
import numpy as np
M= np.array([[1, 2, 3],
             [4, 5, 6],
             [7, 8, 9]])
v= np.array([[1],[2],[3]])

print(M.dot(v))
print(v.dot(v) )
print( v.T.dot(v))
```

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \quad v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

```
ValueError: shapes (3,1) and (3,1) not aligned: 1 (dim 1) != 3 (dim 0)
[[14]] # Why these brackets? Because it's (1,1)-shaped
```

Array Math: Dot Multiplication

$$v_1 = \begin{bmatrix} 3 \\ -3 \\ 1 \end{bmatrix}, \quad v_2 = \begin{bmatrix} 4 \\ 9 \\ 2 \end{bmatrix}$$

```
import numpy
v1= numpy.array([[3],[-3],[1]])
v2= numpy.array([[4],[9],[2]])
print(v1.cross(v2))
```

Array Math: Dot Multiplication

$$v_1 = \begin{bmatrix} 3 \\ -3 \\ 1 \end{bmatrix}, \quad v_2 = \begin{bmatrix} 4 \\ 9 \\ 2 \end{bmatrix}$$

```
import numpy
v1= numpy.array([[3],[-3],[1]])
v2= numpy.array([[4],[9],[2]])
print(v1.cross(v2))

Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'numpy.ndarray' object has no attribute 'cross‘

#Yeah... Slightly convoluted because np.cross() assumes
# horizontal vectors.

print(numpy.cross(v1, v2, axisa=0, axisb=0).T)
```

Array - Math

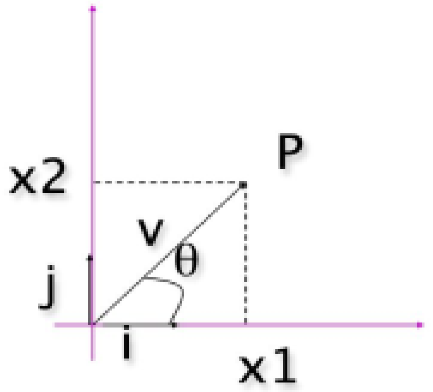
```
import numpy as np
x = np.array([[1,2],[3,4]])
print(np.sum(x))
print(np.sum(x, axis=0))
print(np.sum(x, axis=1))
```

Array Math: Element-wise Multiplication

$$M = \begin{bmatrix} 3 & 0 & 2 \\ 2 & 0 & -2 \\ 0 & 1 & 1 \end{bmatrix}, \quad v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

```
import numpy as np
M= np.array([[3, 0, 2],
                        [2, 0, -2],
                        [0, 1, 1]])
v= np.array([[1],[2],[3]])
print(np.multiply(M, v))
print(np.multiply(v, v) )
```

Orthonormal Basis (Orthogonal and Normalized Basis)



$$\mathbf{i} = (1,0) \qquad \|\,\mathbf{i}\,\| = 1 \qquad \mathbf{i}\cdot\mathbf{j} = 0$$

$$\mathbf{j} = (0,1) \qquad \|\,\mathbf{j}\,\| = 1$$

$$\mathbf{v} = (x_1, x_2) \qquad \mathbf{v} = \mathrm{x}_1\mathbf{i} + \mathrm{x}_2\mathbf{j}$$

$$\mathbf{v}\cdot\mathbf{i} = ? \;\; = (\mathrm{x}_1\mathbf{i} + \mathrm{x}_2\mathbf{j})\cdot\mathbf{i} = \mathrm{x}_1 1 + \mathrm{x}_2 0 = \mathrm{x}_1$$

$$\mathbf{v}\cdot\mathbf{j} = (\mathrm{x}_1\mathbf{i} + \mathrm{x}_2\mathbf{j})\cdot\mathbf{j} = \mathrm{x}_1.0 + \mathrm{x}_2.1 = \mathrm{x}_2$$

Transpose

Definition:

$$\mathbf{C}_{m\times n} = \mathbf{A}^T_{n\times m}$$
$$c_{ij} = a_{ji}$$

Identities:

$$(\mathbf{A}+\mathbf{B})^T = \mathbf{A}^T + \mathbf{B}^T$$
$$(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$$

If $\mathbf{A} = \mathbf{A}^T$, then $\mathbf{A}$ is *symmetric*

Array Math: Transpose

$$M = \begin{bmatrix} 3 & 0 & 2 \\ 2 & 0 & -2 \\ 0 & 1 & 1 \end{bmatrix}, \quad v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

```
import numpy as np
M= np.array([[3, 0, 2],
                          [2, 0, -2],
                          [0, 1, 1]])
v= np.array([[1],[2],[3]])
print(M.T)
print(v.T)
print(M.T.shape, v.T.shape)
```

Array – Transpose Matrix

```
import numpy as np
x = np.array([[1,2], [3,4]])
print(x)
print(x.T)
v = np.array([1,2,3])
print(v)
print(v.T)
```

Matrix Determinant

Useful value computed from the elements of a *square* matrix **A**

$$\det \begin{bmatrix} a_{11} \end{bmatrix} = a_{11}$$

$$\det \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = a_{11}a_{22} - a_{12}a_{21}$$

$$\det \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32}$$

$$- a_{13}a_{22}a_{31} - a_{23}a_{32}a_{11} - a_{33}a_{12}a_{21}$$

Matrix Inverse

Does not exist for all matrices, necessary (but not sufficient) that the matrix is square

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$$

$$\mathbf{A}^{-1} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}^{-1} = \frac{1}{\det \mathbf{A}} \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix}, \det \mathbf{A} \neq 0$$

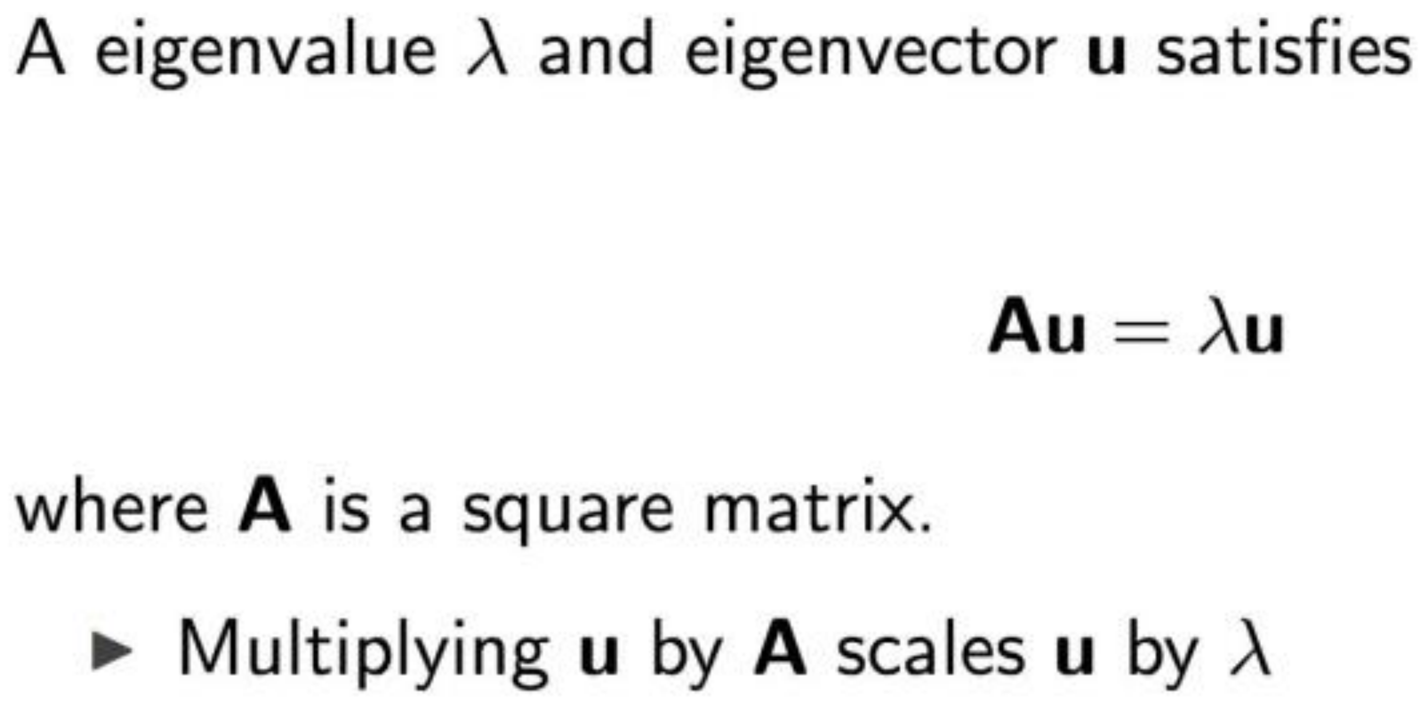
If $\det \mathbf{A} = 0$, $\mathbf{A}$ does not have an inverse.

Array Math: Determinant and Inverse

$$M = \begin{bmatrix} 3 & 0 & 2 \\ 2 & 0 & -2 \\ 0 & 1 & 1 \end{bmatrix}, \quad v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

```
import numpy as np
M= np.array([[3, 0, 2],
             [2, 0, -2],
             [0, 1, 1]])
v= np.array([[1],[2],[3]])
print(np.linalg.inv(M))
#Be careful of matrices that are not invertible!
print(np.linalg.det(M))
```

Matrix Eigenvalues and Eigenvectors



A eigenvalue λ and eigenvector $\mathbf{u}$ satisfies

$$\mathbf{A}\mathbf{u} = \lambda\mathbf{u}$$

where $\mathbf{A}$ is a square matrix.

- Multiplying $\mathbf{u}$ by $\mathbf{A}$ scales $\mathbf{u}$ by λ

Matrix Eigenvalues and Eigenvectors

Rearranging the previous equation gives the system

$$\mathbf{Au} - \lambda\mathbf{u} = (\mathbf{A} - \lambda\mathbf{I})\mathbf{u} = 0$$

which has a solution if and only if $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$.

- The eigenvalues are the roots of this determinant which is polynomial in λ.
- Substitute the resulting eigenvalues back into $\mathbf{Au} = \lambda\mathbf{u}$ and solve to obtain the corresponding eigenvector.

Array Math: Eigenvalues, Eigenvectors

$$M = \begin{bmatrix} 0 & 1 \\ -2 & -3 \end{bmatrix}$$

```
import numpy as np
M= np.array([[0, 1],
                        [-2, -3]])
eigvals, eigvecs = np.linalg.eig(M)
print(eigvals)
print(eigvecs)
```

NOTE: Please read the NumPy docs on this function before using it, lots more information about multiplicity of igenvalues and etc there.

Singular Value Decomposition

Singular values: Non negative square roots of the eigenvalues of $\mathbf{A}^t\mathbf{A}$. Denoted σ_i, $i=1,\ldots,n$

SVD: If **A** is a real m by n matrix then there exist orthogonal matrices **U** $(\in\mathbb{R}^{m\times m})$ and **V** $(\in\mathbb{R}^{n\times n})$ such that

$$A = U\,\Sigma\,V^{-1} \qquad \mathrm{U}^{-1}\mathrm{AV} = \Sigma = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \cdot & \\ & & & \sigma_N \end{bmatrix}$$

Singular Value Decomposition

Suppose we know the singular values of **A** and we know *r* are non zero

$$\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r \geq \sigma_{r+1} = \dots = \sigma_p = 0$$

- Rank(**A**) = *r*.
- Null(**A**) = span$\{\mathbf{v}_{r+1},\dots,\mathbf{v}_n\}$
- Range(**A**)=span$\{\mathbf{u}_1,\dots,\mathbf{u}_r\}$

$||A||_F^2 = \sigma_1^2 + \sigma_2^2 + \dots + \sigma_p^2$ $\qquad ||A||_2 = \sigma_1$

Numerical rank: If *k* singular values of *A* are larger than a given number ε. Then the ε rank of A is *k*.

Distance of a matrix of rank *n* from being a matrix of rank $k = \sigma_{k+1}$

Array Math: Singular Value Decomposition

$$M = \begin{bmatrix} 3 & 0 & 2 \\ 2 & 0 & -2 \\ 0 & 1 & 1 \end{bmatrix}, \quad v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

```
import numpy as np
M= np.array([[3, 0, 2],
                        [2, 0, -2],
                        [0, 1, 1]])
v= np.array([[1],[2],[3]])
U, S, Vtranspose = np.linalg.svd(M)
print(U)
print(S)
print(Vtranspose)
```

Recall SVD is the factorization of a matrix into the product of 3 matrices, and is formulated like so:

$$M = U\Sigma V^T$$

Array – Broadcasting

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

Array – Broadcasting

```
import numpy as np
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)
print(y)
for i in range(4):
        y[i, :] = x[i, :] + v
print(y)
```

Array – Broadcasting

```
import numpy as np
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))
print(vv)
y = x + vv
print(y)
```

Array – Broadcasting

```
import numpy as np
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v
print(y)
```

5 Array – Broadcasting

```
import numpy as np
v = np.array([1,2,3])
w = np.array([4,5])
print(np.reshape(v, (3, 1)) * w)
x = np.array([[1,2,3], [4,5,6]])
print(x + v)
print((x.T + w).T)
print(x + np.reshape(w, (2, 1)))
print(x * 2)
```

Array – Broadcasting

Broadcasting two arrays together follows these rules:
- If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
- The two arrays are said to be *compatible* in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
- The arrays can be broadcast together if they are compatible in all dimensions.
- After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

FEED
BACK