THE UNIVERSITY OF CHICAGO

MASTERS IN COMPUTATIONAL SOCIAL SCIENCE
THE UNIVERSITY OF CHICAGO

MACS 30111

NumPy

The fundamental package for scientific computing with Python

Chapter: 4.2
Documentation: https://numpy.org/devdocs/user/

Topics:

- Introduction and Motivation for array programming
- Creating *Numpy* arrays
- Indexing into *Numpy* arrays
- Working with arrays and array operations
- Advanced array manipulations
- An extended example of standardizing features

The need for high-dimensional array

Audio data (1-dimensional)

```
array([-0.00270751, -0.00303302, -0.00159557, ..., -0.0012889 ,
       -0.00184731, -0.00210062], dtype=float32)
```

Kaggle/UrbanSound8K

- Machine learning
- Deep learning

The need for high-dimensional array

Image data

0	0	0	0	0	0	0	0
0	1	1	0	0	1	1	0
0	1	1	0	0	1	1	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	0
0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0

0	2	15	0	0	11	10	0	0	0	0	9	9	0	0	0
0	0	0	4	60	157	236	255	255	177	95	61	32	0	0	29
0	10	16	119	238	255	244	245	243	250	249	255	222	103	10	0
0	14	170	255	255	244	254	255	253	245	255	249	253	251	124	1
2	98	255	228	255	251	254	211	141	116	122	215	251	238	255	49
13	217	243	255	155	33	226	52	2	0	10	13	232	255	255	36
16	229	252	254	49	12	0	0	7	7	0	70	237	252	235	62
6	141	245	255	212	25	11	9	3	0	115	236	243	255	137	0
0	87	252	250	248	215	60	0	1	121	252	255	248	144	6	0
0	13	113	255	255	245	255	182	181	248	252	242	208	36	0	19
1	0	5	117	251	255	241	255	247	255	241	162	17	0	7	0
0	0	0	4	58	251	255	246	254	253	255	120	11	0	1	0
0	0	4	97	255	255	255	248	252	255	244	255	182	10	0	4
0	22	206	252	246	251	241	100	24	113	255	245	255	194	9	0
0	111	255	242	255	158	24	0	0	6	39	255	232	230	56	0
0	218	251	250	137	7	11	0	0	0	2	62	255	250	125	3
0	173	255	255	101	9	20	0	13	3	13	182	251	245	61	0
0	107	251	241	255	230	98	55	19	118	217	248	253	255	52	4
0	18	146	250	255	247	255	255	255	249	255	240	255	129	0	5
0	0	23	113	215	255	250	248	255	255	248	248	118	14	12	0
0	0	6	1	0	52	153	233	255	252	147	37	0	0	4	1
0	0	5	5	0	0	0	0	0	14	1	0	6	6	0	0

[Kaggle/Animals 151]

- Machine learning
- Deep learning

The need for high-dimensional array

Video data

[Kaggle/youtube-video-dataset]

- Machine learning
- Deep learning

The need for high-dimensional array

1D Array

2D Array

3D Array

4D Array

Nested Lists

```
[["M", "M", "F", "M", "M"],
 ["F", "B", "B", "B", "F"]
 ["M", "M", "M", "M", "B"]
 ["B", "B", "B", "F", "B"]
 ["B", "M", "M", "M", "F"]]
```

Benefits: Easy to code up
(you already know how!)

(0,0) 0	(0,1) 0	(0,2) 1	(0,3) 1	(0,4) 0
(1,0) 0	(1,1) 1	(1,2) 1	(1,3) 2	(1,4) 0
(2,0) 0	(2,1) 0	(2,2) 2	(2,3) 2	(2,4) 1
(3,0) 1	(3,1) 2	(3,2) 1	(3,3) 1	(3,4) 2
(4,0) 0	(4,1) 1	(4,2) 1	(4,3) 0	(4,4) 2

Drawbacks

Sum the elements in 1, 2, and 3-dimensional arrays:

```
sum = 0
for i in range(N):
   sum += one_dim_list[i]
```

```
sum = 0
for i in range(N):
   for j in range(M):
         sum +=
two_dim_list[i][j]
```

```
sum = 0
for i in range(N):
   for j in range(M):
      for k in range(N):
               sum +=
three_dim_list[i][j][k]
```

- Writing dimension-independent **generic** code is difficult when using lists-of-lists.

Drawbacks: *for* loop VS *numpy* operation

```
X = make_big_list(10000000)

total_sum = 0
for i in range(10000000):
    total_sum += X[i]
```

1.17 seconds

```
X = make_big_array(10000000)

total_sum = np.sum(X)
```

0.007 seconds

Numpy operation is faster

- Writing your own for loops can be **slow**. Performance matters when working with large datasets!

Numpy: scientific computing with n-d array

NumPy (pronounced “numb-pie”) is an open-source library in Python. It supports:

1. Multidimensional data structure (the ndarray)
2. Fast mathematical operations
3. An interface that reuses much of the Python list interface

Tradeoff: NumPy arrays can only hold values of the same *type*

Python list can hold different data types.

A NumPy array can only hold the same type: integers, floats, Boolean.

- Most numpy operations are performed on all elements together, different data types have different operations.

```
a = [1, 2.0, 'cat', False]
type(a)
```

```
list
```

```
import numpy as np
b = np.array([True, False, False])
type(b)
```

```
numpy.ndarray
```

Reference:

- https://numpy.org/devdocs/user/absolute_beginners.html
- https://numpy.org/doc/stable/user/basics.types.html

Topics:

- Introduction and Motivation for array programming
- **Creating *Numpy* arrays**
- Indexing into *Numpy* arrays
- Working with arrays and array operations
- Advanced array manipulations
- An extended example of standardizing features

Creating new NumPy arrays

- From existing lists
- Creating arrays with the same value
- Creating arrays with a range of values (e.g., *arange(), linspace*)
- The data type, or “dtype”, of an array
- Array Shape

Coding practice: 4.2.2

Creating an array from an existing list

First, we usually **import** the NumPy library using a shorter name, “np”

```
import numpy as np
```

Then we can **create** a new array from a list

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

We can then **index** into the array like a list:

```
print(x[2])
```

```
3
```

Coding practice: 4.2.2

Creating multi-dimensional arrays from nested lists

To create a two-dimensional array, we can use two nested lists:

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

To print this array:

```
print(y)
```

```
[[1 2 3]
 [4 5 6]]
```

To access elements in this array:

```
print(y[1,2])
```

```
6
```

Coding practice: 4.2.2

Creating arrays: constant values

If you need to create an array of all zeros, you can use *np.zeros*

```
np.zeros(10)
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

You can create multidimensional arrays like this as well, by passing a **shape**-like tuple

```
np.zeros((3, 5))
```

```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

Numpy also supports creating arrays of ones like this

```
np.ones((5, 8))
```

```
array([[1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1.]])
```

Coding practice: 4.2.2

Creating arrays: ranges

Numpy supports creating arrays of numerical ranges with the ***arange*** and ***linspace*** functions.

```
np.arange(start, stop, step)
```

exclusive, just like Python's range()

step size

```
np.linspace(start, stop, numsteps)
```

first value in array

last value in array, **inclusive**

total number of steps in array

```
np.arange(1,5)
```

```
array([1, 2, 3, 4])
```

```
np.linspace(0, 1, 5)
```

```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

Reference:

- https://numpy.org/doc/stable/reference/generated/numpy.arange.html
- https://numpy.org/doc/stable/reference/generated/numpy.linspace.html#numpy.linspace

Coding practice: 4.2.2

Array types

Remember that the contents of NumPy arrays must all be of the **same type**.

***.dtype*:** the array's data type

```
x = np.array([1, 2, 3])
print(x)
```

```
[1 2 3]
```

```
x.dtype
```

```
dtype('int64')
```

Integers (64-bit)

```
x = np.array([1.2, 2.5, 3.7])
print(x)
```

```
[1.2 2.5 3.7]
```

```
x.dtype
```

```
dtype('float64')
```

Floating-point (64-bit)

Coding practice: 4.2.2

The dimension of an arrays: shape

The ***.shape*** attribute: a tuple describing the length in each dimension

```
x = np.array([1, 2, 3])
x.shape
```

```
(3,)
```

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

```
(2, 3)
```

- The length of the shape attribute : the number of dimensions
- One-dimension: row/column vectors

Coding practice: 4.2.2

Topics:

- Introduction and Motivation for array programming
- Creating *Numpy* arrays
- **Indexing into *Numpy* arrays**
- Working with arrays and array operations
- Advanced array manipulations
- An extended example of standardizing features

One-dimensional array indexing

Same as a Python list:

```
x = np.array([1, 2, 3])
print(x[2])
```

```
3
```

Coding practice: 4.2.3

Multi-dimensional array indexing

We can create a two dimensional array

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

```
array([[1, 2, 3],
       [4, 5, 6]])
```

What if we only use one dimension? *y[0]*

```
[[1 2 3]
 [4 5 6]]
```

And index with *y[0, 1]*

```
[[1 2 3]
 [4 5 6]]
```

two indices separated by a comma

Coding practice: 4.2.3

Array slicing

The same Python slicing operations

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

y[0,:3]

```
[[1  2  3  4]
 [5  6  7  8]
 [9 10 11 12]
```

y[1:,:3]

```
[[1  2  3  4]
 [5  6  7  8]
 [9 10 11 12]
```

y[:,2]

```
[[1  2  3  4]
 [5  6  7  8]
 [9 10 11 12]
```

Coding practice: 4.2.3

Quiz

X is an array with X.shape = (5, 10). How many elements are returned by X[:3, :2]?

a. 3
b. 5
c. 6

Topics:

- Introduction and Motivation for array programming
- Creating *Numpy* arrays
- Indexing into *Numpy* arrays
- **Working with arrays and array operations**
- Advanced array manipulations
- An extended example of standardizing features

Mathematical operations with arrays

- List operation: whole list as a unit
- Array operation: element-by-element
- +, -, *, /, %
- >, <, ==
- np.sin, np.cos

Array*array: not matrix operation

Coding practice: 4.2.4

Reference:

- https://numpy.org/doc/stable/reference/routines.statistics.html
- https://numpy.org/doc/stable/reference/routines.linalg.html
- https://numpy.org/doc/stable/reference/routines.logic.html

Reshaping arrays

.*reshape*: change the shape of an array:

- one-dimensional to two-dimensional
- two dimensional back into one-dimensional

Note:

- the new array must have the **same number of elements** as the original array
- element updates on reshaped array mirrors back to the original array

Coding practice: 4.2.5

Reshaping arrays

Coding practice: 2.1.11

Reductions on arrays

Reduce arrays to lower dimensions:

- operation across all elements or along specific dimensions

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
np.sum(x, axis=0)
```

```
x[:,0], x[:,1], x[:2]
```

```
array([5, 7, 9])
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
np.sum(x, axis=1)
```

```
x[0,:], x[1,:]
```

```
array([ 6, 15])
```

- slicing in the axis dimension
- every index in the non-axis dimension

Coding practice: 4.2.6

Quiz

x = np.array([1, 2, 3, 4]). What is the value of x * 2?

- A. np.array([1, 2, 3, 4, 1, 2, 3, 4])
- B. np.array([3, 4, 5, 6])
- C. np.array([2, 4, 6, 8])

Topics:

- Introduction and Motivation for array programming
- Creating *Numpy* arrays
- Indexing into *Numpy* arrays
- Working with arrays and array operations
- **Advanced array manipulations**
- An extended example of standardizing features

Advanced array manipulations

1. Fancy indexing:
   - Indexing with a list/an arrays of indexes (array indexing)
   - Indexing with an array of Booleans to select elements (masked indexing)
2. Performing operations between arrays of different dimensions ("broadcasting")

Coding practice: 4.2.7

Index a subset of elements

```
a = np.array([1, 4, 9, 16, 25, 36, 49])
```

Array indexing with:

- **one index (e.g., *a[3]*)**

```
a = np.array([1, 4, 9, 16, 25, 36, 49])
```

- **a list of indexes (e.g., *a[3, 1, 6]*)**

```
a = np.array([1, 4, 9, 16, 25, 36, 49])
```

```
array([16,  4, 49])
```

- **a multi-dimensional arrays of indexes (e.g., *a[np.array([[1,3], [5,2]])]*)**

```
a = np.array([1, 4, 9, 16, 25, 36, 49])
```

```
array([[ 4, 16],
       [36,  9]])
```

Coding practice: 4.2.7

Masked indexing

Select a subset of elements via a boolean array:

Coding practice: 4.2.7

```
c = np.array([100, 200, 300])
```

```
mask = np.array([True, False, True])
```

Relational operations:

```
c>100
```

```
array([False,  True,  True])
```

```
c[c>100]
```

```
array([200, 300])
```

```
c[(c>100) & (c/100==2)]
```

```
array([200])
```

```
c[(c>100) | (c/100==2)]
```

```
array([200, 300])
```

Quiz

Consider the NumPy arrays x = np.array([1, 3, 5, 7, 9]) and y = np.array([1, 2, 3, 2, 1]). What is the value of the expression x [y < 2] ?

a. np.array([True, False, False, False, True])
b. np.array([True, 3, 5, 7, True])
c. np.array([1, 9])
d. np.array([1, 3, 7, 9])

Broadcasting

You can perform operations (e.g., +, *) on arrays with compatible shapes.

```
x = np.array([1, 2, 3])
y = np.array([7, 8])
x.shape, y.shape
```

```
((3,), (2,))
```

- same sizes for all dimensions
- one of the dimension is 1

```
x * y
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-99-e32109319f52> in <module>
----> 1 x * y

ValueError: operands could not be broadcast together with shapes (3,) (2,)
```

Reference: https://numpy.org/doc/stable/user/basics.broadcasting.html

Broadcasting

Multiply an array by a scalar

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

```
array([ 4,  8, 12])
```

Numpy handles dimensions when two arrays' shapes don't agree.
E.g., if a dimension is one, *numpy* magically stretches array b to match the shape of array a.

Broadcasting

Reference: https://numpy.org/doc/stable/user/basics.broadcasting.html

Broadcasting

a (4,3)

0	0	0
10	10	10
20	20	20
30	30	30

+

b (4)

1	2	3	4

Reference: https://numpy.org/doc/stable/user/basics.broadcasting.html

Broadcasting

Only valid for compatible arrays

Compare array shape from right to left by each dimension:

- equal
- or one of them is 1

```
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
```

```
A      (3d array):  15 x 3 x 5
B      (3d array):  15 x 1 x 5
Result (3d array):  15 x 3 x 5
```

```
A      (1d array):  3
B      (1d array):  4
```

```
A      (2d array):      2 x 1
B      (3d array):  8 x 4 x 3
```

Reference: https://numpy.org/doc/stable/user/basics.broadcasting.html

Topics:

- Introduction and Motivation for array programming
- Creating *Numpy* arrays
- Indexing into *Numpy* arrays
- Working with arrays and array operations
- Advanced array manipulations
- **An extended example of standardizing features**

The need for standardized features

Patient number	Height (inches)	Weight (lbs)
11471	72	190
9412	60	124
12911	65	170
42311	75	225
32141	50	85

Patient number	Height (mi)	Weight (g)
11471	0.001136	86182.6
9412	0.0009470	56245.5
12911	0.0010259	77110.7
42311	0.0011837	102058
32141	0.0007891	38555.4

The need for standardized features

We can take an unknown feature and "**standardize**" it to have **a mean of zero and a standard deviation of one**. This way we can have some intuition about how far away a value is from "average" (zero mean).

This is a very common operation for most machine learning and statistics tasks, where often you have more than just two features -- sometimes you have two million! How would you ever make sense of all those units?

How to standardize data

```
data = [[89.0, 66.0, 23.0, 94.0],
        [137.0, 40.0, 35.0, 168.0],
        [78.0, 50.0, 32.0, 88.0],
        [197.0, 70.0, 45.0, 543.0],
        [189.0, 60.0, 23.0, 846.0],
        [166.0, 72.0, 19.0, 175.0],
        [118.0, 84.0, 47.0, 230.0],
        [103.0, 30.0, 38.0, 83.0],
        [115.0, 70.0, 30.0, 96.0],
        [126.0, 88.0, 41.0, 235.0]]
```

$$\mu_j = 1/N * \sum_{i=0}^{N-1} m_{i,j}$$

$$\sigma_j = \sqrt{1/N * \sum_{i=0}^{N-1} (m_{i,j} - \mu_j)^2}$$

$$m'_{i,j} = (m_{i,j} - \mu_j)/\sigma_j$$

The regular python way of standardizing features

```
def standardize_features(data):
    N = len(data)
    M = len(data[0])

    # initialize the result w/ NxM list of lists of zeros.
    rv = []
    for i in range(N):
        rv.append([0] * M)

    # for each feature
    for j in range(M):
        mu = compute_feature_mean(data, j)
        sigma = compute_feature_stdev(data, j)

       # standardized feature
       for i in range(N):
          rv[i][j] = (data[i][j] - mu)/sigma
    return rv
```

$$\mu_j = 1/N * \sum_{i=0}^{N-1} m_{i,j}$$

```python
def compute_feature_mean(data, j):
    '''
    Compute the mean of feature (column) j

    Inputs:
      data (list of list of floats)

    Returns (float): mean of feature j
    '''

    N = len(data)
    total = 0
    for i in range(N):
        total += data[i][j]
    return total/N
```

$$\sigma_j = \sqrt{1/N * \sum_{i=0}^{N-1} (m_{i,j} - \mu_j)^2}$$

```python
def compute_feature_stdev(data, j):
    '''
    Compute the standard deviation of feature (column) j

    Inputs:
      data (list of lists of floats)

    Returns (float): standard deviation of feature j
    '''

    N = len(data)
    mu = compute_feature_mean(data, j)
    total = 0
    for i in range(N):
        total = total + (data[i][j] - mu) ** 2
    return math.sqrt(1 / N * total)
```

Feature standardization with Numpy

```
def standardize_features(data):
      '''
    Standardize features to have mean 0.0 and standard deviation 1.0.
     Inputs:
       data (2D NumPy array): data to be standardized
      Returns (2D NumPy array): standardized data
    '''

    mu_vec = data.mean(axis=0)
    sigma_vec = data.std(axis=0)

    return (data - mu_vec)/ sigma_vec
```

Simply feature standardization with Numpy

We significantly simplify the above code with NumPy:

1. The NumPy code is less work to write
2. The functions in NumPy are already thoroughly debugged, reducing the likelihood of a mistake
3. The NumPy code will be faster, and will therefore let us handle more data.

Quiz

Assume *x = np.array([1.0, 2.0, 3.0])*. Which is closest to the output of *x / x.sum()*

A. 2
B. np.array([0.166, 0.333, 0.5])
C. np.array([1.0, 2.0, 3.0])

List VS array

- Different data type
- List as a whole unit
- Operation:
  - +: concatenation
- Nested list: [][][]
- Slow: for…

- Same data type
- Element-wise
- Operation:
  - +/-/*
- Index: [, ,]
- Faster
- Broadcasting

List VS array

- Different data types
- +: concatenation
- Pop(), append(), extend()
- List as a whole unit

- Efficiency
- Index[], [], []

- Same data type
- Mathematical
- Element-wise
- Broadcasting
- Fast, generic
- Index, [, , ,]
- reshape