STATS 507
Data Analysis in Python

Week5-2: Intro to NumPy

Dr. Xian Zhang

Adapted from slides by Professor Jeffrey Regier

Recall Scope of this class

Part 1: Introduction to Python

Data types, functions, classes, objects, functional programming

Part 2: Numerical Computing and Data Visualization

numpy, scipy, matplotlib, scikit-learn, Seaborn

Part 3: Dealing with structured data

pandas, regular expressions, retrieving web data, SQL, real datasets

Part4: Intro to Deep Learning

PyTorch, Perceptron, Multi-layer perceptron, SGD, regularization, ConvNets

Overview

- NumPy (Fundamentals and Advanced)
- SciPy
- Matplotlib
- scikit-learn
- Seaborn

Part 1: NumPy

What's NumPy

Open-sourced add-on modules for **numerical computing**

1) NumPy: **numerical** python, have multidimensional arrays
2) Optimized library for **matric and vector computation**
3) Makes uses of C/C++ subroutines and memory-efficient data structure
4) Building block for other packages: SciPy, Matplotlib, scikit-learn, scikit-image and provides fast numerical computations and high-level math functions

Ref: https://numpy.org/doc/stable/user/absolute_beginners.html

Compared with MATLAB

A free competitor to MATLAB.

Numpy quickstart guide:

https://docs.scipy.org/doc/numpy-dev/user/quickstart.html

For MATLAB fans:

https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html

Closely related package SciPy for optimization

See https://docs.scipy.org/doc/

Why NumPy (v.s. built-in lists)

Python is very slow and NumPy are much more **efficient**

- 1000 x 1000 matrix multiply
  - Python triple loop takes > 10 min.
  - NumPy takes ~0.03 seconds

Have more advanced mathematical functions, **convenient**

- Have mathematical operations applied directly to arrays
  - Linear algebra, statistical operations...

Broadcasting and vectorization saves time and amount of code

1. NumPy as numerical computing (Basics)

2. Array indexing

3. Vector and Matrix Operations

3. Broadcasting

NumPy data types

NumPy has its own preliminary data types, which is optimized for numerical computations and efficient memory.

- boolean (`bool`)
- integer (`int32, int64`)
- unsigned integer (`uint`)
- floating point (`float32, float64`)
- complex (`complex`)

`import … as …` lets us import a package and give it a shorter name.

```
import numpy as np

x = np.float32(3.1415)
type(x)
```

```
numpy.float32
```

```
x
```

```
3.1415
```

Note that this is not the same as a Python int.

```
x = np.int(8675309)
x
```

```
8675309
```

Many more complicated data types are available
e.g., each of the numerical types can vary in how many bits it uses
https://docs.scipy.org/doc/numpy/user/basics.types.html

NumPy data types

```
1 x = np.float64(3.1415)
2 x
```

```
3.1415
```

```
1 y = np.float32(3.1415)
2 type(y)
```

```
numpy.float32
```

```
1 x==y
```

```
False
```

```
1 x==np.float64(y)
```

```
False
```

```
1 x = np.int_(8675309)
2 type(x)
```

```
numpy.int64
```

32-bit and 64-bit representations are distinct!

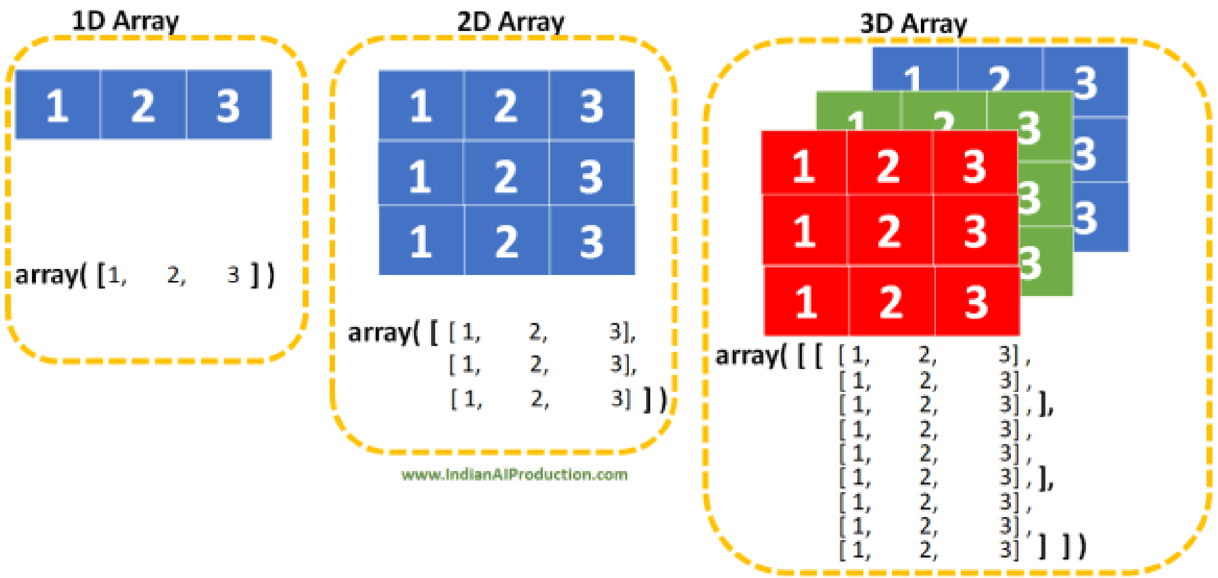
Data type followed by underscore uses the default number of bits. This default varies by system.

As a rule, it's best never to check for equality of floats. Instead, check whether they are within some error tolerance of one another.

NumPy main data types

One of the key data type of NumPy is its **N-dimensional array object** (also referred as: array, NumPy array, np.ndarray).

- A numpy array is a grid of values, **all of the same type**
- *Rank* of the array: the dimension of the arrays
- *Shape*: a tuple of integers giving the size of the array along each dimension



Creating NumPy array

There are several ways to create a NumPy array.
1) Converting Python sequences to NumPy arrays (lists and tuples)

```
# From a list
arr1 = np.array([1, 2, 3, 4, 5], dtype = 'uint')
print(type(arr1))
# From a tuple
arr2 = np.array((1, 2, 3, 4, 5), dtype = 'uint')
print(type(arr2))
```

```
<class 'numpy.ndarray'>
<class 'numpy.ndarray'>
```

Creating arrays with specific data type

https://numpy.org/doc/stable/user/basics.creation.html

https://docs.scipy.org/doc/numpy/user/basics.creation.html

Using NumPy array functions – 1D

```python
# Create an array of zeros
zeros_arr = np.zeros(5)  # [0. 0. 0. 0. 0.]
# Create an array of ones
# [[1. 1. 1.]
#  [1. 1. 1.]]
ones_2d = np.ones((2, 3))
# Create an array with a range of values
range_arr = np.arange(0, 10, 2)  # [0 2 4 6 8]
# Create an array with evenly spaced values
linspace_arr = np.linspace(0, 1, 5)  # [0.   0.25 0.5  0.75 1.  ]
```

`np.zeros` and `np.ones` generate arrays of 0s or 1s, respectively. Shape parameter (2,3) means to create a 2-D array with two rows and three columns.

[start, end, step]

[start, end, specified number of elements]

https://docs.scipy.org/doc/numpy/user/basics.creation.html

More on `numpy.arrange` creation

- np.arange(x): array version of Python's range(x), like [0,1,2,...,x-1]
- np.arange(x,y): array version of range(x,y), like [x,x+1,...,y-1]
- np.arange(x,y,z): array of elements [x,y) in z-size increments.
- Related useful functions, that give better/clearer control of start/endpoints and allow for multidimensional arrays:

https://docs.scipy.org/doc/numpy/reference/generated/numpy.linspace.html
https://docs.scipy.org/doc/numpy/reference/generated/numpy.ogrid.html
https://docs.scipy.org/doc/numpy/reference/generated/numpy.mgrid.html

Using NumPy array functions – 2D

Besides `np.zeros`, `np.ones`...

2D identify matrices

```
import numpy as np
print(np.eye(3))
print(np.eye(3, 5))
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]]
```

2D square matrices with diagonal terms

```
np.diag([1, 2, 3])
```

```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

...or even reading directly from a file

see more in https://docs.scipy.org/doc/numpy/user/basics.creation.html

Based on existing arrays

3) Create NumPy array based on the properties of existing arrays

```
import numpy as np

# Create a sample array
sample = np.array([[1, 2, 3], [4, 5, 6]])

# Create a new array with the same shape as sample, filled with 7
full_like_arr = np.full_like(sample, 7)
print(full_like_arr)
```

```
[[7 7 7]
 [7 7 7]]
```

```
# Can be replaced with ones like
zeros_like_arr = np.zeros_like(sample)
print(zeros_like_arr)
```

```
[[0 0 0]
 [0 0 0]]
```

Can help in initializing arrays for calculations, creating masks, or setting up default values.

NumPy arrays attributes

NumPy array is used for storage of homogeneous data

i.e., all elements the same type

Every array has attributes like `ndim`, `shape`, `dtype` and `size`

```
b = np.arange(10).reshape(2,5)
print(b)
```

```
[[0 1 2 3 4]
 [5 6 7 8 9]]
```

```
# Get array attributes
print(b.ndim) # dimension of the array
print(b.shape) # shape of the array
print(b.dtype) # data type
print(b.size) # no. of elements
```

Return a tuple

https://numpy.org/doc/stable/reference/arrays.ndarray.html#array-attributes

1. NumPy as numerical computing (Basics)

2. Array indexing

3. Vector and Matrix Operations

3. Broadcasting

Slicing

Just like Python lists, NumPy array can be sliced.

Since arrays maybe multidimensional, you **MUST** specify a slice for each dimension of the array.

```
import numpy as np
# Create the following dim 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]
```

Slices, strides, indexing from the end, etc.
Just like with Python lists.

Integer array indexing

When you index into NumPy arrays using slicing, you will always get a **subarray** of the original array. In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array

```
a = np.array([[1, 2], [3, 4], [5, 6]])
# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"
```

```
[1 4 5]
```

```
# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"
```

```
[1 4 5]
```

Boolean array indexing

Boolean array indexing lets you pick our **<u>arbitrary</u>** elements of an array and uses arrays of `True/False` values to select elements. This is particularly useful for conditional selection.

```
a = np.array([[1,2], [3, 4], [5, 6]])
bool_idx = (a > 2)   # Find the elements of a that are bigger than 2;
                     # this returns a numpy array of Booleans of the same
                     # shape as a, where each slot of bool_idx tells
                     # whether that element of a is > 2.
print(bool_idx)      # Prints "[[False False]
                     #          [ True  True]
                     #          [ True  True]]"
# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])  # Prints "[3 4 5 6]"
```

```
[[False False]
 [ True  True]
 [ True  True]]
[3 4 5 6]
```

```
# We can do all of the above in a single concise statement:
print(a[a > 2])     # Prints "[3 4 5 6]"
```

Boolean operations: `np.all()` and `any()`

```
x = np.arange(10)
np.all(x>7)
```

```
False
```

```
np.any(x>7)
```

```
True
```

```
np.any([x>7,x<2])
```

```
True
```

```
np.any([x>7,x<2], axis=1)
```

```
array([ True,  True], dtype=bool)
```

```
np.any([x>7,x<2], axis=0)
```

```
array([ True,  True, False, False, False, False, False, False,  True,  True], dtype=bool)
```

`axis` argument picks which axis along which to perform the Boolean operation. If left unspecified, it treats the array as a single vector.

Setting `axis` to be the first (i.e., 0-th) axis yields the entrywise behavior we wanted.

Boolean operations: `np.logical_and()`

`numpy` also has built-in Boolean vector operations, which are simpler/clearer at the cost of the expressiveness of `np.any()`, `np.all()`.

```
x = np.arange(10)
x[np.logical_and(x>3,x<7)]
```

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

```
np.logical_or(x<3,x>7)
```

```
array([ True,  True,  True, False, False, False, False, False,  True,  True], dtype=bool)
```

```
x[np.logical_xor(x>3,x<7)]
```

```
array([0, 1, 2, 3, 7, 8, 9])
```

```
x[np.logical_not(x>3)]
```

```
array([0, 1, 2, 3])
```

This is an example of a numpy “universal function” (ufunc), which we’ll discuss more later.

1. NumPy as numerical computing (Basics)

2. Array indexing

3. Math, Vector and Matrix Operations

4. Broadcasting

Array Math

Basic mathematical functions operate **elementwise** on arrays, and are available both as:

1) operator overloads

2) as functions in the NumPy module.

```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))

# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

Array `axis`

NumPy provides many useful functions for performing computations on arrays.

```
# [[1 2]
#  [3 4]]
x = np.array([[1,2],[3,4]])
print(np.sum(x, axis=0))  # Compute sum of each column
# Output: [4 6]
print(np.sum(x, axis=1))  # Compute sum of each row
print(np.sum(x))  # Compute sum of all elements
# Output: 10
```

1) The function applies the operation along that axis.
2) The result reduces the specified axis to a single value.

Array `axis`

NumPy provides many useful functions for performing computations on arrays.

```python
# [[1 2]
#  [3 4]]
x = np.array([[1,2],[3,4]])
print(np.sum(x, axis=0))  # Compute sum of each column
# Output: [4 6]
print(np.sum(x, axis=1))  # Compute sum of each row
print(np.sum(x))  # Compute sum of all elements
# Output: 10
```

- axis=0 (first axis): Operates vertically, down through rows.
- For a matrix, this is the direction of column vectors.
- Index changes along this axis correspond to moving between rows.

- axis=1 (second axis): Operates horizontally, across columns.
- For a matrix, this is the direction of row vectors.
- Index changes along this axis correspond to moving between columns.

Vector operations in NumPy

1. inner product
2. outer product
3. cross product

dot is method of array objects available both as a

1) function in the NumPy module and
2) as an instance method

```
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])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))

# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

Vector operations in NumPy

Note:

Unlike MATLAB, * is elementwise multiplication, not matrix multiplication. We instead use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices.

```python
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])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))

# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

Matric operations in NumPy

```
import numpy.linalg
```

`array()`	creates a matrix
`dot()`	performs matrix multiplication
`transpose()`	transposes a matrix
`linalg.inv()`	calculates the inverse of a matrix
`linalg.det()`	calculates the determinant of a matrix
`flatten()`	transforms a matrix into 1D array

More on this later!

Read more on: https://www.programiz.com/python-programming/numpy/matrix-operations
https://numpy.org/doc/stable/reference/routines.linalg.html

1. NumPy as numerical computing (Basics)

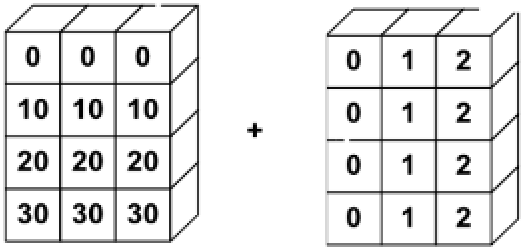
2. Array indexing

3. Math, Vector and Matrix Operations

4. Broadcasting

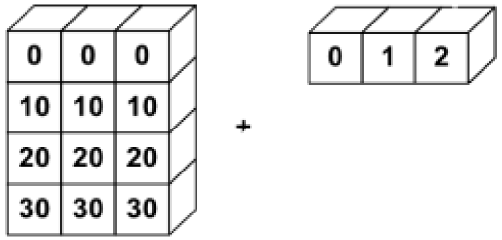
Array broadcast

A powerful mechanism that allows NumPy to work with arrays of **different shapes** when performing **arithmetic** operations.



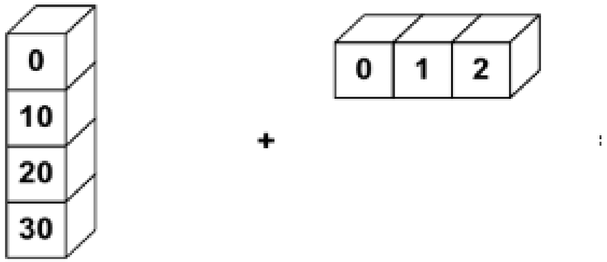

```python
# Both a and b with shape (4, 3)
a = np.array([[0, 0, 0],
              [10, 10, 10],
              [20, 20, 20],
              [30, 30, 30]])
b = np.array([[0, 1, 2],
              [0, 1, 2],
              [0, 1, 2],
              [0, 1, 2]])
print(a+b)
```

```
[[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]
```




```python
# Create array 'a' with shape (4, 3)
a = np.array([[0, 0, 0],
              [10, 10, 10],
              [20, 20, 20],
              [30, 30, 30]])
# Create array 'b' with shape (1, 3)
b = np.array([[0, 1, 2]])
print(a+b)
```

```
[[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]
```

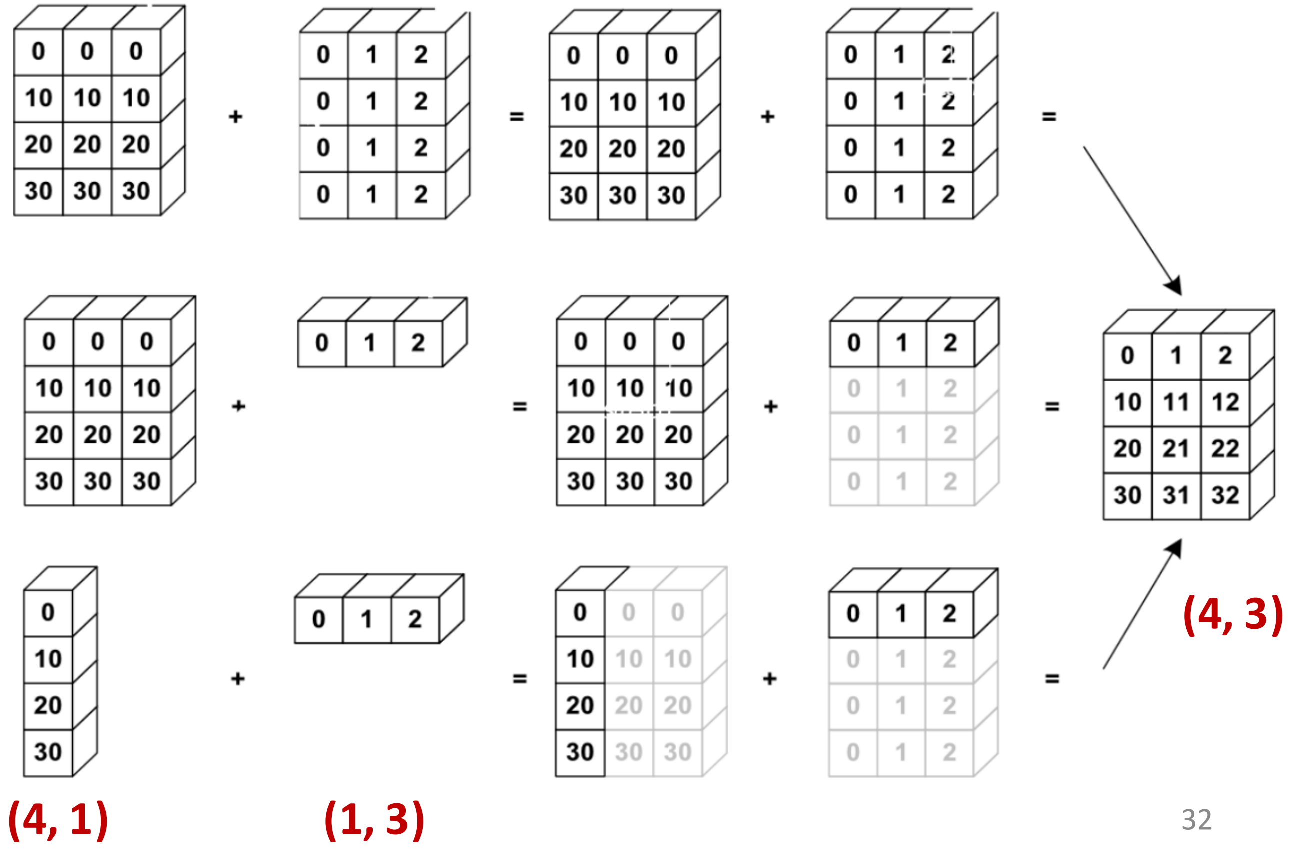



```python
a = np.array([[0], [10], [20], [30]])
b = np.array([0, 1, 2])
print(a+b)
```

```
[[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]
```

Array broadcast

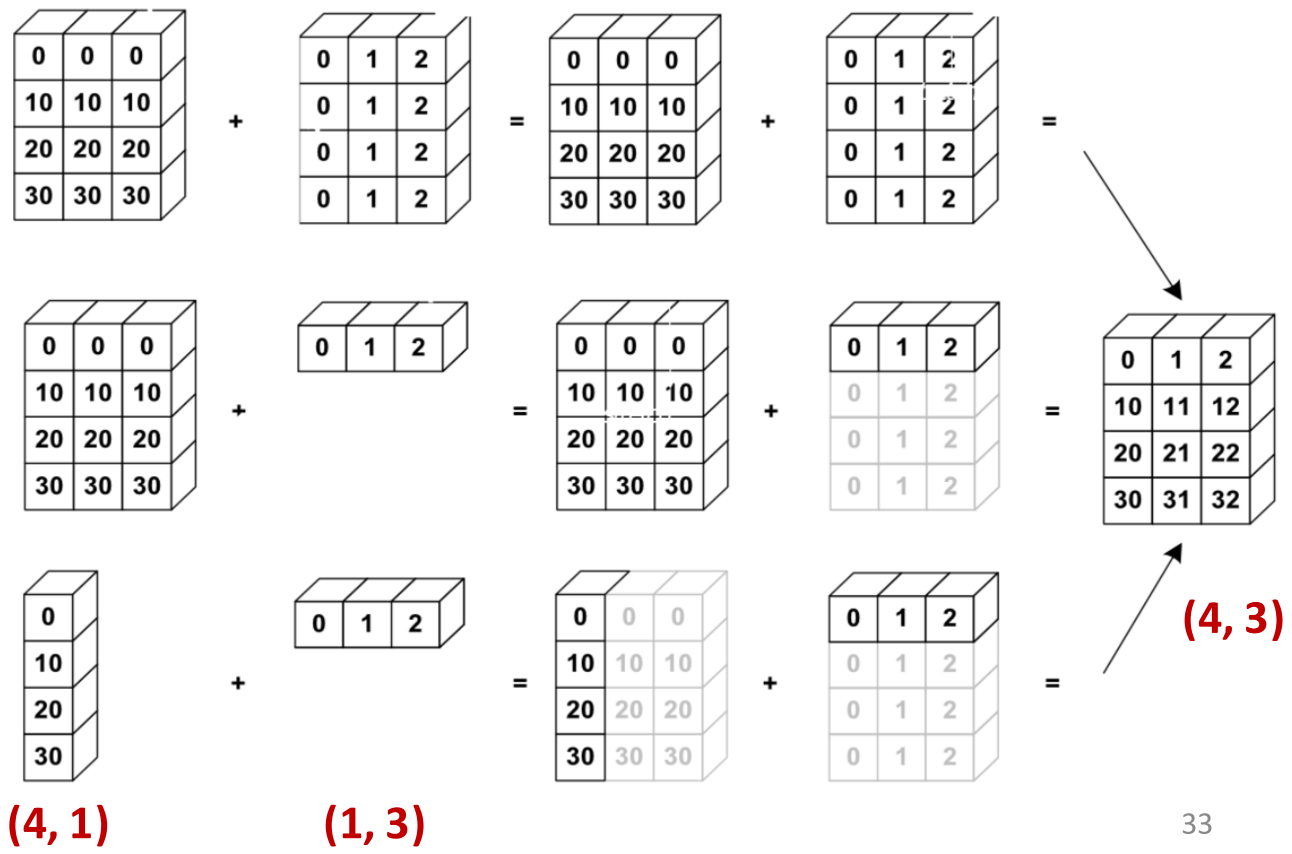
1. When operating on two arrays, NumPy compares **shapes**. Two dimensions are **compatible** when:
   1. They are of equal size
   2. One of them is 1



Array broadcast

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

One of the most useful things about NumPy and one of the its defining features.



Slides adapted from Sven Schmit

Array broadcast

The arrays can be broadcast together only if they are compatible in all dimensions.

1. They are of equal size
2. One of them is 1

```python
import numpy as np
# Create array 'a' with shape (4, 3)
a = np.array([[0, 0, 0],
              [10, 10, 10],
              [20, 20, 20],
              [30, 30, 30]])
# Create array 'b' with shape (2, 3)
b = np.array([[0, 1, 2],
              [3, 4, 5]])
print(a+b)
```

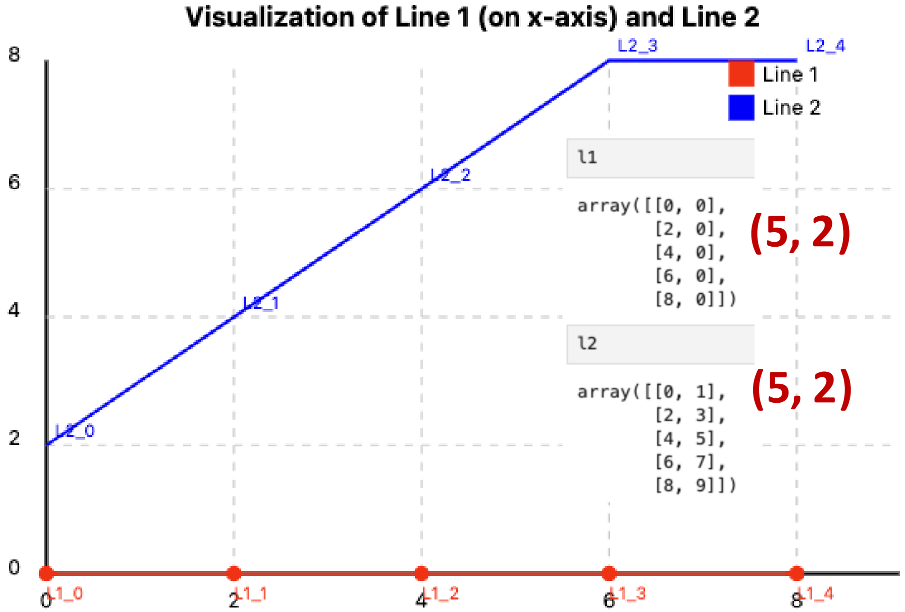
```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[29], line 12
      9 # Create array 'b' with shape (2, 3)
     10 b = np.array([[0, 1, 2],
     11               [3, 4, 5]])
---> 12 print(a+b)

ValueError: operands could not be broadcast together with shapes (4,3) (2,3)
```

Array (implicit) broadcast

Broadcasting typically makes your code more concise and faster, so you should strive to use it where possible.

```python
# Define the two lines
l1 = np.array([(0, 0), (2, 0), (4, 0), (6, 0), (8, 0)])
l2 = np.arange(10).reshape(5, 2)
```



Two for loops are much slower …

```python
distances_loop = []
for i in range(len(l1)):
    # Calculate the squared difference for each coordinate
    squared_diff_x = (l1[i][0] - l2[i][0])**2
    squared_diff_y = (l1[i][1] - l2[i][1])**2
    # Sum the squared differences and take the square root
    distance = np.sqrt(squared_diff_x + squared_diff_y)
    distances_loop.append(distance)
print(distances_loop)
```

```
[1.0, 3.0, 5.0, 7.0, 9.0]
```

```python
distances_broadcast = np.sqrt(((l1 - l2)**2 * np.array([1, 1])).sum(axis=1))
distances_broadcast
```

```
array([1., 3., 5., 7., 9.])
```

(5, 1)

By specify axis = 1, we are telling Python to perform the sum operation horizontally across each row.

In class practice

Math and ufuncs in NumPy

From the documentation:

> A universal function (or ufunc for short) is a function that operates on ndarrays in an element-by-element fashion, supporting array broadcasting, type casting, and several other standard features. That is, a ufunc is a “vectorized” wrapper for a function that takes a fixed number of scalar inputs and produces a fixed number of scalar outputs.
> https://docs.scipy.org/doc/numpy/reference/ufuncs.html

So ufuncs are vectorized operations, just like in R and MATLAB

Statistics in NumPy

NumPy implements all the standard **statistics distributions/functions** you can expect

```
die_rolls = np.random.randint(1, 7, size=10)
die_rolls
```

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

```
errors = np.random.normal(loc=0.0, scale=1.0, size=10)
errors
```

mean std

```
array([-0.90664245, -1.20080379,  1.18109637, -0.49519272,  1.5105014 ,
        1.9763521 , -0.85794091, -1.24809495,  0.32142042,  1.30210332])
```

Examples of statistical functions provided by NumPy:

```
mean = np.mean(die_rolls)
mean
```

```
3.1
```

```
std = np.std(die_rolls)
std
```

```
1.1357816691600546
```

```
result1 = np.percentile(die_rolls, 25)
result1
```

```
2.25
```

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

Other things

HW4 due today.

HW5 is out today.

Coming next:

NumPy practices