NumPy for Linear Algebra

Created by Ben Poole, Summer 2024

Edited by Razvan Bunescu, Fall 2024

Table of Notation

Symbol	Meaning	Symbol	Meaning
D	dataset	h f	hypothesis function
$\mathbf{X}$	input / data matrix	$x_{i,j}$	ith row and jth column
$\mathbf{x}$ $\vec{x}$	feature / input vector	x_i	ith element
$\mathbf{Y}$ $\mathbf{T}$	labels / targets matrix	$\mathbf{y}$ $\mathbf{t}$	labels/targets vector
N	number of features / columns	M	number of data samples / rows
$(\mathbf{x}, \mathbf{y})$	training sample	$(\mathbf{x}', \mathbf{y}')$	testing sample
		$\hat{\mathbf{y}}$ $f(\mathbf{x}; \mathbf{w})$ $h(\mathbf{x}; \mathbf{w})$	predictions

NumPy Review

NumPy is a scientific library that is frequently used for its highly optimized linear algebra powers. One of NumPy's main attractions is its N-dimensional array objects which work great for storing and manipulating datasets! As we will come to see, most datasets are just 2D arrays where the number of data samples corresponds to the number of rows and the number of features corresponds to the number of columns.

```
import traceback
import numpy as np

# Sets NumPy gloabl seed such that any
# randomly generation done by NumPy is seeded
np.random.seed(0)
```

Generating Arrays

A numpy array is a grid of values, typically all of the same type. Array values can be indexed by either a integer or tuple/list of nonnegative integers. The number of dimensions is usually referred to as the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension [1].

Let's make this definition concrete by first creating a rank 1 array or an array with 1 dimension (1D)! Notice, we can make an array simply by passing a list as input!

```
a = np.array([1, 2, 3])   # Create a rank 1 array
a
```

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

We can check the type just like before.

```
type(a)
```

```
numpy.ndarray
```

Further, we can check the shape of the array which specifies the length of each dimensions. Since we have a 1D array we can only see one dimension!

```
a.shape
```

```
(3,)
```

We can also use Python's `len` built-in function however it only checks the length of the first dimension!

```
len(a)
```

```
3
```

NumPy also let's generate arrays of different sizes that are automatically filled with values. For example, we can generate many different types of 2x2 arrays that are filled with different values automatically. Notice, we pass a tuple of `(2,2)` to define the shape we want.

See the [Array Creation Routines docs] to see all the different ways NumPy can generate different arrays.

`np.zeros` creates an array of zeros.

```
zeros_arr = np.zeros((2, 2))
zeros_arr
```

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

If we check the shape we can see `zeros_arr` does indeed have 2 rows and 2 columns!

```
zeros_arr.shape
```

```
(2, 2)
```

`np.ones` creates an array of ones.

```
ones_arr = np.ones((2, 2))
ones_arr
```

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

`np.full` fills and array with any specified number.

```
tens_arr = np.full((2, 2), 10)
tens_arr
```

```
array([[10, 10],
       [10, 10]])
```

`np.arange` makes an array of evenly spaced values within a given interval (much like Python's range function).

```
aranged_arr = np.arange(10)
aranged_arr
```

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

`np.random.rand` creates an array with random values. Note that you can use `np.random.seed` to set a global seed such that the same random array will always be generated. You can also use `rng = np.random.RandomState` to set a local seed and then use `rng.rand` to use said local seed!

```
# Creates an array with random values using global seed
random_arr = np.random.rand(3,2)
random_arr
```

```
array([[0.5488135 , 0.71518937],
       [0.60276338, 0.54488318],
       [0.4236548 , 0.64589411]])
```

```
# Creates an array with random values using local seed
rng = np.random.RandomState(2)
random_arr = rng.rand(3,2)
random_arr
```

```
array([[0.4359949 , 0.02592623],
       [0.54966248, 0.43532239],
       [0.4203678 , 0.33033482]])
```

Indexing Arrays

Indexing works just like with lists. Let's take a look at some examples.

Indexing a rank 1 array or 1D array is very straight forward and most similar to indexing lists.

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

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

```
print(one_d[0], one_d[1], one_d[2])
```

```
1 2 3
```

Furthermore, we can index NumPy arrays with lists or tuples such that we index multiple elements at once!

Notice, we have to add a comma after the tuple, otherwise NumPy attempts to check multiple dimensions which we don't have as we only have a rank 1 array!

```
print(one_d[(0,1,2),])
print(one_d[[0,1,2]])
```

```
[1 2 3]
[1 2 3]
```

We can change an element just like lists.

```
one_d[0] = 5
one_d
```

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

Now let's check indexing with a rank 2 array or a 2D array as follows.

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

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

Like before we can get the shape of the array. Notice, since we are a 2D array we now have two dimensions we can index!

```
two_d.shape
```

```
(2, 3)
```

Also we can use `len` but notice only the length of the first dimension is returned!

```
len(two_d)
```

```
2
```

Next, we can index each element by specifying two indices (one for the first dimension and one for the second).

```
print(two_d[0, 0], two_d[0, 1], two_d[1, 0])
```

```
1 2 4
```

Likewise, we can slice arrays just like lists.

```
print(f"Returns the first column for all rows: {two_d[:, 0]}")
print(f"Return all columns for the first row: {two_d[0, :]}")
print(f"Returns the first two columns for all rows: \n {two_d[:, :2]}")
```

```
Returns the first column for all rows: [1 4]
Return all columns for the first row: [1 2 3]
Returns the first two columns for all rows:
 [[1 2]
 [4 5]]
```

Finally, once again we can index using tuples or lists.

```
print(f"Tuple Indexing - Returns the first and last colums for all rows:\n {two_d[:, (0, -1)]}")
print(f"List Indexing - Returns the first and last colums for all rows:\n {two_d[:, [0, -1]]}")
```

```
Tuple Indexing - Returns the first and last colums for all rows:
 [[1 3]
 [4 6]]
List Indexing - Returns the first and last colums for all rows:
 [[1 3]
 [4 6]]
```

Copying and Slices

An important and sometimes confusing concept regarding NumPy arrays is copying. There are typically two methods for copying arrays (and copying in general): shallow copying and deep copying.

Shallow copying either copies the object or elements memory addresses such that if a copied element is changed then the change is reflected in the original and vice-versa. Further, NumPy allows for what they call *views* or *slices*. Views or slices can be naively thought of as shallow copies of an entire or only part of an array.

> View/Slice: An array that does not own its data, but refers to another array's data instead. For example, we may create a view that only shows every second element of another array:

Deep copying make stores data in a new memory address such that if we make changes to a copied object or element the changes are **not** reflected in the original and vica-versa.

- Additional Sources
  - Offical NumPy slicing docs
  - Views versus copies in NumPy
  - Copy and View in NumPy Array
  - What's the difference between a view and a shallow copy of a numpy array?

Below is an example of which operations will result in a shallow copy and which operations will result in a deep copy. We can check whether a certain operation acted as a shallow or deep copy by looking at the memory address and by physically changing an element of the original array and seeing if the change is reflected in the copied array.

Notice that all the copied arrays that mention deep copy have a different memory ID from `a`, the original array. Also notice that the deep copy arrays second index remains unchanged by the change to the original arrays.

```
import copy

# Original array
a = np.arange(10)

object_copy = a # Copy object
slice_shallow1 = a[1:5] # Copy select elements
```

```
slice_shallow2 = a[:] # Copy all elements
slice_deep1 = a[1:5].copy() # deep copy select elements
deep1 = a.copy() # deep copy all elemenets using NumPy
deep2 = copy.deepcopy(a) # deep copy all elements using Python

c = np.zeros(a.shape)
c[:] = a[:] # deep copy elements of a into a new array

print(f"Original before change: {a} Memory address: {id(a)}")

# Update index 1
a[1] = 999

# Debug information
print('-'*90)
print(f"Original after change \n Array: {a} Memory address: {id(a)}")
print(f"Simple shallow copy of entire array \n Array: {object_copy} Memory address: {id(object_copy)}")
print(f"Slice shallow copy \n Array: {slice_shallow1} Memory address: {id(slice_shallow1)}")
print(f"Slice enitre array and shallow copy \n Array: {slice_shallow2} Memory address: {id(slice_shallow2)}")
print(f'Slice array and deep copy \n Array: {slice_deep1} Memory address: {id(slice_deep1)}')
print(f'Entire array and deep copy with NumPy \n Array: {deep1} Memory address: {id(deep1)}')
print(f'Enitre array and deep copy with Python \n Array: {deep2} Memory address: {id(deep2)}')
print(f'Deep copy elements of `a` into new array `c` \n Array: {c} Memory address: {id(c)}')
```

```
Original before change: [0 1 2 3 4 5 6 7 8 9] Memory address: 139966613075216
------------------------------------------------------------------------------------------
Original after change
 Array: [  0 999   2   3   4   5   6   7   8   9] Memory address: 139966613075216
Simple shallow copy of entire array
 Array: [  0 999   2   3   4   5   6   7   8   9] Memory address: 139966613075216
Slice shallow copy
 Array: [999   2   3   4] Memory address: 139966613076368
Slice enitre array and shallow copy
 Array: [  0 999   2   3   4   5   6   7   8   9] Memory address: 139966613076560
Slice array and deep copy
 Array: [1 2 3 4] Memory address: 139966613076752
Entire array and deep copy with NumPy
 Array: [0 1 2 3 4 5 6 7 8 9] Memory address: 139966613076656
Enitre array and deep copy with Python
 Array: [0 1 2 3 4 5 6 7 8 9] Memory address: 139966613077040
Deep copy elements of `a` into new array `c`
 Array: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.] Memory address: 139966613077136
```

Reshaping and Adding New Dimensions

Reshaping allows you to change the shapes of an array without changing data of array. Reshaping is a frequently used functions for making sure array mathematical operations such as the dot product work as intended and errors don't arise (we will see these specific errors shortly).

```
reshaping_array = np.arange(1, 7)
reshaping_array
```

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

If we wanted to change our rank 1 array into a rank 2 array we could then simply do a reshape as follows.

```
reshaping_array.reshape(2, 3)
```

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

Notice $2 * 3 = 6$ where 6 is the total number of elements. This means if we tried to reshape to a (2,4) we would get an error as follows.

```
try:
    reshaping_array.reshape(2, 4)
except ValueError as e:
    traceback.print_exc()
```

```
Traceback (most recent call last):
  File "/tmp/ipykernel_55364/830660679.py", line 2, in <module>
    reshaping_array.reshape(2, 4)
ValueError: cannot reshape array of size 6 into shape (2,4)
```

Further, note we can use `-1` in a dimensions. This allows NumPy to automatically determine what the dimension size should be based on the other dimensions!

Below is an example of automatically determining the size of the column dimension.

```
print(reshaping_array.reshape(2, -1))
print(reshaping_array.reshape(2, -1).shape)
```

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

Below is an example of automatically determining the size of the row dimension.

```
print(reshaping_array.reshape(-1, 3))
print(reshaping_array.reshape(-1, 3).shape)
```

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

Lastly, we can also add new dimensions when using reshape by adding a 1.

```
reshaping_array.reshape(-1 ,3 ,1).shape
```

```
(2, 3, 1)
```

Alternatively, we can do the following as well to add a new dimensions.

```
print(f"reshaping_array original shape: {reshaping_array.shape}")
reshaping_array[:, None, None].shape
```

```
reshaping_array original shape: (6,)
(6, 1, 1)
```

What is an Axis?

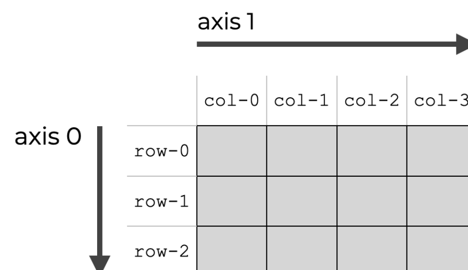
Axis is a very common parameter found in many NumPy functions that determines which dimension the function will be applied across. In other words, the axis parameter essentially refers to which axis gets collapsed! Below we cover the simple rank 2 array or a 2D array. This idea of axes is correlated with dimensions so if we have a N-dimensional array we can apply a function across any of the possible dimensions.

This idea can be really tricky for beginners so don't worry if it doesn't make complete sense at first. Please seek help or check the additional resources below if you are left really confused.

```
axis=None: Apply function or operation across the entire array-wise.

axis=0: Apply operation column-wise, function or operation is applied across all rows for each column
(i.e., 1 output for each column).

axis=1: Apply operation row-wise, function or operation is applied across all columns for each row
(i.e., 1 output for each row).
```



- Additional Sources
  - [How to Set Axis for Rows and Columns in NumPy]
  - [NumPy Axes explained]

Let's create a simply array with a shape of (3, 4). Notice, we use NumPy's `arange` function (docs) to quickly generate 3 1D arrays and `vstack` (docs) function to stack said arrays into a (3, 4) array.

```
axis_array = np.vstack([np.arange(1,5), np.arange(1,5), np.arange(1,5)])

print(f"axis_array output: \n {axis_array}")
print(f"axis_array shap|e: {axis_array.shape}")
```

```
axis_array output:
 [[1 2 3 4]
 [1 2 3 4]
 [1 2 3 4]]
axis_array shap|e: (3, 4)
```

Alternatively, we could have used NumPy's `stack` function (docs) and specified `axis=0` to indicate we have to vertically stack each array on top of one another.

```
axis_array = np.stack([np.arange(1,5), np.arange(1,5), np.arange(1,5)], axis=0)

print(f"axis_array output: \n {axis_array}")
print(f"axis_array shape: {axis_array.shape}")
```

```
axis_array output:
 [[1 2 3 4]
 [1 2 3 4]
 [1 2 3 4]]
axis_array shape: (3, 4)
```

Here we take the sum with no axis parameter given (in other words, `axis=None` by default). Thus, the sum of the entire array is computed.

```
np.sum(axis_array)
```

```
np.int64(30)
```

Likewise, we can apply the same idea to the `max` function where the max of the entire array is taken.

```
np.max(axis_array)
```

```
np.int64(4)
```

Here we take the sum and max for each column by passing `axis=0` which means the 1st dimension (the row dimension) is collapsed. Meaning, we now want to take the sum/max across columns!

Alternatively, think about the axis visually. Recall, `axis=0` points down vertically (see above picture). This means the operation is applied for each column.

```
# Column-wise: 1 number for each column
np.sum(axis_array,  axis=0)
```

```
array([ 3,  6,  9, 12])
```

```
# Another example but now using max
np.max(axis_array, axis=0)
```

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

Here we take the sum and max row-wise (for each row) by passing `axis=1` which means the 2nd dimension (the column dimension) is collapsed. Meaning, we now want to take the sum/max across rows!

Alternatively, think about the axis visually. Recall, `axis=1` points across the array horizontally (see above picture). This means the operation is applied for each row.

```
# Row-wise: 1 number for each row
np.sum(axis_array,  axis=1)
```

```
array([10, 10, 10])
```

```
# Another example but now using max
np.max(axis_array, axis=1)
```

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

Datatypes

Every NumPy array is a grid of elements of the same type. NumPy provides a large set of numeric datatypes that you can use to construct arrays. NumPy tries to guess a datatype when you create an array, but functions that construct arrays usually also includes an optional argument to explicitly specify the datatype [1].

As we can see below, NumPy automatically detects our list being converted into an array is integers. We use the `dtype` method to check the type of the elements.

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

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

Likewise the same applies to floats.

```
x = np.array([1.0, 2.0])
x.dtype
```

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

Lastly, we can specify the type either when initializing the array or after the array has been initialized.

```
x = np.array([1, 2], dtype=np.float64)
x.dtype
```

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

```
x = x.astype(np.int64)
x.dtype
```

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

List Comprehension with Arrays

The same idea of list comprehension applies to NumPy arrays. However, now when we a 2D array you we need nested loops!

```
x_list = np.arange(1, 11).reshape(2,5)
x_list
```

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

Normally, If we wanted to loop through this array to find all the even numbers we would need a nested for loop.

```
even = []
for row in x_list:
    for col in row:
        if col%2 == 0:
            even.append(col)
np.array(even)
```

```
array([ 2,  4,  6,  8, 10])
```

However, we can easily write this in one line of code with list comprehension, which basically flattens the above nested for loop.

The basic outline for list comprehension with nested loops and a condition is as follows:

```
[expression for item in list for item2 in item condition]
```

```
evens_array = np.array([col for row in x_list for col in row if col%2==0])
evens_array
```

```
array([ 2,  4,  6,  8, 10])
```

Searching Arrays: Finding Specific Values and Indexes

Frequently in machine learning when you have a dataset you will need to select only certain data samples. For instance, you might want to select only data samples that belong to certain class (in other words, selecting data samples with certain a "label"). Luckily, NumPy makes this idea of finding values relatively simply by using concepts such as subsetting.

- Additional Sources
  - numpy.where() – Explained with examples
  - fast python numpy where functionality?
  - np.where docs

Let's say we have a fake dataset where `X` which contains the features and `y` contains the labels which correspond to the class each data sample belongs to.

Further, let's say we our fake dataset has 20 data samples (rows) and there are 5 features (columns). Additionally, our fake label array `y` contains 20 labels whose values can be either 0, 1, or 2 where each value indicates which class each data sample in `X` belongs to. **Keep in mind, each row in `y` corresponds to the same row in `X`.**

Below creates a fake data set with corresponding labels. We use some NumPy functions to randomly generate them.

```
# Here we set a seed such that everytime we run this cell we can the same random array!
rng = np.random.RandomState(0)
X = rng.normal(size=(20, 5))
y = rng.randint(0, 3, size=20)
```

```
X
```

```
array([[ 1.76405235,  0.40015721,  0.97873798,  2.2408932 ,  1.86755799],
       [-0.97727788,  0.95008842, -0.15135721, -0.10321885,  0.4105985 ],
       [ 0.14404357,  1.45427351,  0.76103773,  0.12167502,  0.44386323],
       [ 0.33367433,  1.49407907, -0.20515826,  0.3130677 , -0.85409574],
       [-2.55298982,  0.6536186 ,  0.8644362 , -0.74216502,  2.26975462],
       [-1.45436567,  0.04575852, -0.18718385,  1.53277921,  1.46935877],
       [ 0.15494743,  0.37816252, -0.88778575, -1.98079647, -0.34791215],
       [ 0.15634897,  1.23029068,  1.20237985, -0.38732682, -0.30230275],
       [-1.04855297, -1.42001794, -1.70627019,  1.9507754 , -0.50965218],
       [-0.4380743 , -1.25279536,  0.77749036, -1.61389785, -0.21274028],
       [-0.89546656,  0.3869025 , -0.51080514, -1.18063218, -0.02818223],
       [ 0.42833187,  0.06651722,  0.3024719 , -0.63432209, -0.36274117],
       [-0.67246045, -0.35955316, -0.81314628, -1.7262826 ,  0.17742614],
       [-0.40178094, -1.63019835,  0.46278226, -0.90729836,  0.0519454 ],
       [ 0.72909056,  0.12898291,  1.13940068, -1.23482582,  0.40234164],
       [-0.68481009, -0.87079715, -0.57884966, -0.31155253,  0.05616534],
       [-1.16514984,  0.90082649,  0.46566244, -1.53624369,  1.48825219],
       [ 1.89588918,  1.17877957, -0.17992484, -1.07075262,  1.05445173],
       [-0.40317695,  1.22244507,  0.20827498,  0.97663904,  0.3563664 ],
       [ 0.70657317,  0.01050002,  1.78587049,  0.12691209,  0.40198936]])
```

```
y.reshape(-1, 1)
```

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

```
       [0],
       [1],
       [2],
       [0],
       [2],
       [2],
       [1],
       [1],
       [1],
       [2],
       [0],
       [0],
       [1]])
```

Remember that the first data sample `X[0, :]` corresponds to the first label `y[0]`!

```
print(f"1st data sample: {X[0, :]} \n1st data sample's label: {y[0]}")
```

```
1st data sample: [1.76405235 0.40015721 0.97873798 2.2408932  1.86755799]
1st data sample's label: 0
```

Subsetting (Boolean Indexing)

Subsetting entails finding values in a array based on some condition. This condition creates a boolean array, which is then used to select all the data samples that are true (i.e., meets the specified condition).

```
y
```

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

First, we can make a condition where we only want the labels whose class is 1. Notice that all the 1 elements in the below output are set to True while all the 0 and 2 elements are False.

```
y == 1
```

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

Now we can perform some sort of computation on our selected data. Let's take the mean for each feature/column for all the data samples whose class labels are 1.

```
np.mean(X[y == 1], axis=0)
```

```
array([-0.57034902, -0.20348499,  0.31765296,  0.0670375 ,  0.57322077])
```

We can also check the means of the other classes as well!

```
np.mean(X[y == 0], axis=0)
```

```
array([ 0.24218808,  0.82097514,  0.09421713, -0.21502782,  0.4300825 ])
```

```
np.mean(X[y == 2], axis=0)
```

```
array([-0.30273568,  0.16981485,  0.10546403, -1.03953571,  0.04722023])
```

Additionally, we can combine conditions (make sure to include parenthesis surrounding each condition). So we can want to take the mean of the data samples whose class labels are 1 and 2.

```
np.mean( X[(y == 1) | (y == 2)], axis=0)
```

```
array([-0.46742081, -0.05990812,  0.23604183, -0.35856758,  0.37091287])
```

Finding Data Locations

Subsetting is great for accessing data but what if we want the indexes or locations instead? For instance, what do we do if we want to get all the data sample **indexes** whose class label is 1? This is where the NumPy `where()` function comes into play as it will find all the indexes in the array that satisfy some condition.

Note, we index `np.where(y == 1)[0]` at 0 because `np.where` returns a tuple of rows and columns indexes. Since our labels `y` is a 1D array it only returns a tuple with only row information.

```
locs = np.where(y == 1)[0]
```

Here we can see `locs` holds all the index values which have labels equal to 1.

```
locs
```

```
array([ 2,  4,  5,  8, 13, 14, 15, 19])
```

We can see what the corresponding data samples are by indexing `X` with `locs`.

```
X[locs]
```

```
array([[ 0.14404357,  1.45427351,  0.76103773,  0.12167502,  0.44386323],
       [-2.55298982,  0.6536186 ,  0.8644362 , -0.74216502,  2.26975462],
       [-1.45436567,  0.04575852, -0.18718385,  1.53277921,  1.46935877],
       [-1.04855297, -1.42001794, -1.70627019,  1.9507754 , -0.50965218],
       [-0.40178094, -1.63019835,  0.46278226, -0.90729836,  0.0519454 ],
       [ 0.72909056,  0.12898291,  1.13940068, -1.23482582,  0.40234164],
       [-0.68481009, -0.87079715, -0.57884966, -0.31155253,  0.05616534],
       [ 0.70657317,  0.01050002,  1.78587049,  0.12691209,  0.40198936]])
```

Moreover, we can check to make sure each sample has label 1 by indexing `T` with `locs`.

```
y[locs]
```

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

Once again we can perform some operation on all the data samples whose class is 1 just like with subsetting.

```
X[locs].mean(axis=0)
```

```
array([-0.57034902, -0.20348499,  0.31765296,  0.0670375 ,  0.57322077])
```

Additionally, we can combine conditions just like with subsetting (make sure to include parenthesis surrounding each condition).

```
np.where((y == 1) | (y == 2))[0]
```

```
array([ 2,  3,  4,  5,  8,  9, 11, 12, 13, 14, 15, 16, 19])
```

Matmul, Shape Mismatch, and Broadcasting Errors

Matmul and Shape Mismatch Errors

Throughout this course you will frequently run into what we refer to as "matmul" errors. Matmul errors typically arise when taking the dot product of two arrays and the shapes of the arrays aren't compatible. Recall that the dot product requires the columns of the first array and rows of the second array to match!

For this example, let's assume `A` is some fake data and `b` is a weight vector. Let's then try to make a prediction by taking the dot product of our data and weights (this is something we will do frequently throughout the semester).

Below are a few different ways to compute the dot product. The main difference between `@` or `np.matmul` and `np.dot` only really comes into play when dealing with arrays with dimensions greater than 3. For those interested, see this post on the differences between the methods. For now, we can assume that all operations are roughly doing the same thing - computing the dot product.

```
A = np.ones((5, 3))
b = np.arange(3).reshape(1, -1)
```

```
print(f"A shape: {A.shape}")
print(f"b shape: {b.shape}")
```

```
A shape: (5, 3)
b shape: (1, 3)
```

Based on these shapes given above, can you see the error we are about to run into? Take a second to think about what conditions need to be met in order for the dot product to be taken.

```
try:
    A @ b
except ValueError as e:
    traceback.print_exc()
```

```
Traceback (most recent call last):
  File "/tmp/ipykernel_55364/4201264362.py", line 2, in <module>
    A @ b
ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with gufunc signature (n?,k),(k,m?)->(n?,m?) (si
```

```
try:
    np.matmul(A, b)
except ValueError as e:
    traceback.print_exc()
```

```
Traceback (most recent call last):
  File "/tmp/ipykernel_55364/46849502.py", line 2, in <module>
    np.matmul(A, b)
ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with gufunc signature (n?,k),(k,m?)->(n?,m?) (si
```

```
try:
    np.dot(A, b)
except ValueError as e:
    traceback.print_exc()
```

```
Traceback (most recent call last):
  File "/tmp/ipykernel_55364/391474462.py", line 2, in <module>
    np.dot(A, b)
ValueError: shapes (5,3) and (1,3) not aligned: 3 (dim 1) != 1 (dim 0)
```

Notice that none of the above methods work because the shapes of the arrays don't match which means the dot product can't be computed! Also, note that the first two throw slightly different errors than the last method.

Templating print statements for equations: Often times these `matmul` errors leave beginners scratching their heads at what they did wrong. The issue here is that our shapes don't match. One useful method for debugging shape issues is to print the equation of interest where you replace the variables with their shapes. This is a useful practice when you are first learning to covert matrix equations to code.

For instance given

```
A @ b
```

It can be useful to add the shapes next to the variables in print statements.

```
A(5, 3) @ b(1, 3)
```

```
print(f"A{A.shape} @ b{b.shape}")
```

```
A(5, 3) @ b(1, 3)
```

See, notice the rows of `b` don't match the columns of `A`. What do we need to do to fix this? One possible solution is simply to transpose `b` using the `.T` method for NumPy arrays.

```
print(f"A{A.shape} @ b{b.T.shape}")
```

```
A(5, 3) @ b(3, 1)
```

Now, notice the columns of `A` and rows of `b` match! Let's try recomputing the dot product with the rows and columns matching.

```
result = A @ b.T
print(f"Result shape: {result.shape}")
result
```

```
Result shape: (5, 1)
array([[3.],
       [3.],
       [3.],
       [3.],
       [3.]])
```

Alternatively, we can reshape `b` to fix this issue as well.

```
result = A @ b.reshape(-1, 1)
print(f"Result shape: {result.shape}")
result
```

```
Result shape: (5, 1)
array([[3.],
       [3.],
       [3.],
       [3.],
       [3.]])
```

Broadcasting

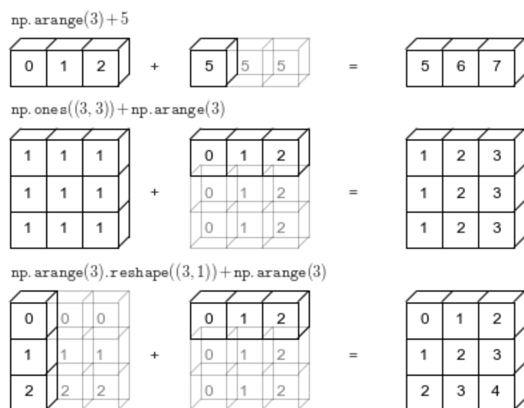
Broadcasting is an implicit functionality of NumPy to allow arithmetic between arrays of different dimensions or shapes. This is a functionality you will be aware of as it will naturally arise in your code and can potentially cause you headaches if you don't know how it works! Below are the three general rules that broadcasting follows.

> Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions has its shape padded with ones on its leading (left) side.

> Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.

> Rule 3: If in any dimension sizes disagree and neither is equal to 1, an error is raised.

Given these rules, let's take a deeper look at what they are actually saying. We are going to reproduce the examples given in the image below and step through them and see how each rule applies to each example. Lastly, note that idea of braodcasting is not limited to the addition operation!



- Additional Sources
  - [Gentle Introduction to Broadcasting]
  - [Numpy Broadcasting Docs]
  - [Introduction to Broadcasting]

Example 1 - Adding a scalar to an array

1. Initial shapes: Note that `a` is a 1D array and `b` is a scalar and therefore has no shape!

```
a.shape = (3,)
b.shape = ()
```

2. Convert to array: NumPy first turns the scalar `b` into a 1D array.

```
a.shape = (3,)
b.shape = (1,)
```

3. Apply rule 2: NumPy stretches `b` by copying the scalar value of `b` repeatedly until the first dimension of `b` matches the first dimension of `a`. Refer to the above picture where the 5 is copied two more times!

```
a.shape = (3,)
b.shape = (3,)
```

```
a = np.arange(3)
b = 5

a + 5
```

```
array([5, 6, 7])
```

Example 2 - Adding a 1D array to a 2D array (MOST COMMON SCENARIO)

1. Beginning shapes:

```
A.shape = (3, 3)
b.shape = (3,)
```

2. Apply rule 1 to `b`: NumPy does so by adding a new dimension to the left of the existing dimensions of `b`.

```
A.shape = (3, 3)
b.shape = (1, 3)
```

3. Apply rule 2 to `b`: NumPy stretches `b` by copying `b` repeatedly until the first dimension of `b` matches the first dimension of `A`. Refer to the above picture where `b` is copied two more additional times!

```
A.shape = (3, 3)
b.shape = (3, 3)
```

```
A = np.ones((3, 3))
b = np.arange(3,)

A + b
```

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

Example 3 - Adding 1D array to a 2D array

1. Beginning shapes:

```
A.shape = (3, 1)
b.shape = (3,)
```

2. Apply rule 1 to `b`: NumPy does so by adding a new dimension to the left of the existing dimensions for `b`.

```
A.shape = (3, 1)
b.shape = (1, 3)
```

3. Apply rule 2 to `A` and `b`: Notice now NumPy needs to stretch both `A` and `b` so their dimension lengths match! NumPy does this once again by copying the values of `A` and `b` each twice.

```
A.shape = (3, 3)
b.shape = (3, 3)
```

```
A = np.arange(3).reshape((3, 1))
b = np.arange(3)

A + b
```

```
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

Example 4 - Broadcasting Error (No corresponding image)

1. Beginning shapes:

```
A.shape = (3, 2)
b.shape = (3,)
```

2. Apply rule 1 to `b`: NumPy does so by adding a new dimension to the left of the existing dimensions for `b`.

```
A.shape = (3, 2)
b.shape = (1, 3)
```

3. Apply rule 2 to `b`: NumPy stretches `b` by copying `b` repeatedly until the first dimension of `b` matches the first dimension of `A`.

```
A.shape = (3, 2)
b.shape = (3, 3)
```

4. Mismatch error is thrown because `A`'s second dimension does not have a length of `1` which means rule 2 can't be applied to it! Meaning, the shapes can not be made to match!

```
(3, 2) != (3, 3)
```

```
A =  np.ones((3, 2))
b = np.arange(3)

try:
    A + b
except ValueError as e:
    traceback.print_exc()
```

```
Traceback (most recent call last):
  File "/tmp/ipykernel_55364/4033666338.py", line 5, in <module>
    A + b
ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

Linear Algebra with NumPy

In this section, we will briefly review basic linear algebra concepts and provide code examples using NumPy. Thoroughly go through the below notes as you will need to understand and code ALL concepts covered below.

Vectors

Here we will look at **vector** operations and code examples.

Defining a 1D Vector

Below, we quickly define a 1D vector and perform check some basic properties of the vector `x`.

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

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

Here we check the `len()` which returns the length of the 1st dimenision.

```
len(x)
```

```
3
```

Next we check the shape which returns a tuple containing the length of each dimension. Notice, we have 1 dimension of length 3.

```
x.shape
```

```
(3,)
```

Finally, we check both the data type of `x` which is a `np.ndarray` and the data type of an element which is a integer (see NumPy data types).

```
print(f"x type: {type(x)}")
print(f"x[0] or element type: {type(x[0])}")
```

```
x type: <class 'numpy.ndarray'>
x[0] or element type: <class 'numpy.int64'>
```

Defining 2D Array as a Vector

Recall, we can define a 2D array to represent a vector. Thus, in coding terms the `x_2d` will be a 2d array, but technically it is still a vector since one of the dimensions has a length of 1. This can be a confusing concept to grasp, and a good example of where code and theoretical math differ.

Below we define a 2D vector with the shape (1, 3) where the first dimension has length one. Technically, this can be thought of as a row vector as the vector has a single row. A column vector would be the opposite, the shape of (3, 1).

```
x_2d = np.array([[1, 2, 3]])
x_2d
```

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

```
x_2d.shape
```

```
(1, 3)
```

```
type(x_2d), type(x_2d[0])
```

```
(numpy.ndarray, numpy.ndarray)
```

Vector Operations

Below are basic vector operations using both 1D and 2D vectors. Before we do, we define another 1D vector `y`.

```
y = np.array([2, 4, 6])
print(f"y: {y}")
print(f"y.shape: {y.shape}")
```

```
y: [2 4 6]
y.shape: (3,)
```

Addition

Below is code for computing vector addition using the syntax `+`.

First, is matrix-matrix addition, where the vectors are added element-wise.

```
x + y
```

```
array([3, 6, 9])
```

Next, is matrix-scalar addition, where the scalar is added to each element.

```
x + 3
```

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

Subtraction

Below is code for computing vector subtraction using the syntax `-`.

First, is vector-vector subtraction, where the vectors are subtracted element-wise.

```
x - y
```

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

```
x - 3
```

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

Division

Below is code for computing vector-scalar division using the syntax `/`. Notice the divisor number divides each element.

```
x / 3
```

```
array([0.33333333, 0.66666667, 1.        ])
```

```
x_2d / 3
```

```
array([[0.33333333, 0.66666667, 1.        ]])
```

Power

Below is code for computing raising a vector to a power using the `**` syntax. Notice the power is applied to each element.

```
x ** 2
```

```
array([1, 4, 9])
```

```
x_2d ** 2
```

```
array([[1, 4, 9]])
```

The square-root can then be computed two ways, as seen below.

```
x ** (1/2)
```

```
array([1.        , 1.41421356, 1.73205081])
```

```
np.sqrt(x)
```

```
array([1.        , 1.41421356, 1.73205081])
```

Scalar Multiplication

Below is code for computing vector-scalar multiplication using the syntax `*`. Notice, each element of `x` is just multiplied by 3.

```
x * 3
```

```
array([3, 6, 9])
```

```
3 * x
```

```
array([3, 6, 9])
```

Element-wise Multiplication

Below is code for computing element-wise product using the syntax `*`. This is also called the **Hadamard product** (denoted by the $\odot$ or $*$ operators). Notice, each element in `x` is multiplied by `y`.

```
x * y
```

```
array([ 2,  8, 18])
```

Dot Product

Recall that the dot product is defined as:

> algebraic operation that takes two equal-length sequences of numbers (usually coordinate vectors), and returns a single number.

To compute the dot product, you can use `np.dot`, `np.matmul` or the overloaded simple notation of `@`. It is worth noting that `@` technically refers to matrix multiplication but also works for computing the dot product in NumPy. The main difference between `@` or `np.matmul` and `np.dot` only really comes into play when dealing with arrays with dimensions greater than 3. For those interested, see this post on the differences between the functions.

```
x @ y
```

```
np.int64(28)
```

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

```
np.int64(28)
```

```
np.matmul(x, y)
```

```
np.int64(28)
```

```
x_2d @ y
```

```
array([28])
```

```
np.dot(x_2d, y)
```

```
array([28])
```

Below is an example of reshaping an 2D vector so that we do not get a matmul error!

```
y @ x_2d.reshape(-1, 1)
```

```
array([28])
```

Transposition

Below we display how to take the transpose of a vector.

Now let's transpose a vector using the `.T` syntax. Notice, transposing has no effect on a 1D vector.

```
x_t = x.T
x_t
```

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

```
x_t.shape
```

```
(3,)
```

Yet, if we transpose a 2D vector, the dimension of length 1 changes and now we have a column vector.

```
x_2d_t = x_2d.T
x_2d_t
```

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

```
x_2d_t.shape
```

```
(3, 1)
```

We can recover the original 2D row vector by simply transposing again.

```
x_2d_t_t = x_2d_t.T
x_2d_t_t
```

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

```
x_2d_t_t.shape
```

```
(1, 3)
```

Matrices

Here we will look at **matrix** operations and code examples.

Defining a 2D Matrix

Below, we quickly define a 2D matrix and perform check some basic properties of the matrix `X`.

To define `X` we use `np.arange` to create a 1D vector of shape (6), then we use `.reshape` to reshape it into a (3, 2) matrix.

```
X = np.arange(6).reshape((3,2))
X
```

```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

```
X.shape
```

```
(3, 2)
```

Now, notice that we have to index both dimensions (row and column) to get a specific element data type. If we just index the row or column we get a vector which is another `np.ndarray` type.

```
print(f"X type: {type(X)}")
print(f"X[0] or row type: {type(X[0])}")
print(f"X[0, 0] or element type: {type(X[0, 0])}")
```

```
X type: <class 'numpy.ndarray'>
X[0] or row type: <class 'numpy.ndarray'>
```

```
X[0, 0] or element type: <class 'numpy.int64'>
```

Matrix Operations

Below are basic matrix for 2D matrices. Before we do, we define another 2D matrix `A`.

```
A = np.arange(5,11).reshape((3,2))
A
```

```
array([[ 5,  6],
       [ 7,  8],
       [ 9, 10]])
```

```
A.shape
```

```
(3, 2)
```

Addition

Below is code for computing matrix addition using the syntax `+`.

First, is matrix-matrix addition, where the matrices are added element-wise.

```
X + A
```

```
array([[ 5,  7],
       [ 9, 11],
       [13, 15]])
```

Next, is matrix-scalar addition, where the scalar is added to each element.

```
A + 3
```

```
array([[ 8,  9],
       [10, 11],
       [12, 13]])
```

Subtraction

Below is code for computing matrix subtraction using the syntax `-`.

First, is vector-vector subtraction, where the vectors are subtracted element-wise.

```
X - A
```

```
array([[-5, -5],
       [-5, -5],
       [-5, -5]])
```

Next, is matrix-scalar subtraction, where the scalar is subtracted to each element.

```
X - 3
```

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

Division

Below is code for computing matrix-scalar division using the syntax `/`. Notice the divisor number divides each element.

```
X / 3
```

```
array([[0.        , 0.33333333],
       [0.66666667, 1.        ],
       [1.33333333, 1.66666667]])
```

Power

Below is code for computing raising a matrix to a power using the `**` syntax. Notice the power is applied to each element.

```
X ** 2
```

```
array([[ 0,  1],
       [ 4,  9],
       [16, 25]])
```

The square-root can then be computed two ways, as seen below.

```
X ** (1/2)
```

```
array([[0.        , 1.        ],
       [1.41421356, 1.73205081],
       [2.        , 2.23606798]])
```

```
np.sqrt(X)
```

```
array([[0.        , 1.        ],
       [1.41421356, 1.73205081],
       [2.        , 2.23606798]])
```

Scalar Multiplication

Below is code for computing matrix-scalar multiplication using the syntax `*`. Notice, each element of `X` is just multiplied by 3.

```
X * 3
```

```
array([[ 0,  3],
       [ 6,  9],
       [12, 15]])
```

```
3 * X
```

```
array([[ 0,  3],
       [ 6,  9],
       [12, 15]])
```

Element-wise Multiplication

Below is code for computing element-wise product using the syntax `*`. This is also called the **Hadamard product** (denoted by the $\odot$ or $*$ operators). Notice, each element in `X` is multiplied by `A`.

```
X * A
```

```
array([[ 0,  6],
       [14, 24],
       [36, 50]])
```

Matrix Multiplication

Recall the definition of matrix multiplication as follows:

> matrix multiplication is a binary operation that produces a matrix from two matrices. For matrix multiplication, the number of columns in the first matrix must be equal to the number of rows in the second matrix. The resulting matrix, known as the matrix product, has the number of rows of the first and the number of columns of the second matrix.

The equivalent of the dot product for matrices is referred to as matrix multiplication. Often times people will overload the term dot product though to also refer to matrix multiplication, which is technically incorrect.

To compute the matrix product, you can use `np.dot`, `np.matmul` or the overloaded simple notation of `@`. It is worth noting that `@` technically refers to matrix multiplication but also works for computing the dot product in NumPy. The main difference between `@` or `np.matmul` and

`np.dot` only really comes into play when dealing with arrays with dimensions greater than 3. For those interested, see this post on the differences between the functions.

```
B = np.ones((2,3))
B
```

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

```
np.dot(X, B)
```

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

```
np.matmul(X, B)
```

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

```
X @ B
```

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

If we want to compute the matrix product between `X` and `A` we need to reshape or transpose one of the matrices.

```
X
```

```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

```
A.T
```

```
array([[ 5,  7,  9],
       [ 6,  8, 10]])
```

```
X @ A.T
```

```
array([[ 6,  8, 10],
       [28, 38, 48],
       [50, 68, 86]])
```

Transposition

Now let's transpose a matrix using the `.T` syntax

```
X_t = X.T
X_t
```

```
array([[0, 2, 4],
       [1, 3, 5]])
```

```
X_t.shape
```

```
(2, 3)
```

Once again, we can take the transpose of the transpose `X_t` to get back the original shape.

```
X_t.T.shape
```

```
(3, 2)
```

Trace

To compute the trace, simply pass a matrix to the `np.trace()` function.

```
C = np.arange(1,10).reshape(3,3)
C
```

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

```
np.trace(C)
```

```
np.int64(15)
```

If we do not pass a square matrix, it will still only add those values along the diagonal.

```
X
```

```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

```
np.trace(X)
```

```
np.int64(3)
```

Rank

To compute the rank, simply pass a matrix to the `np.linalg.matrix_rank()` function.

```
np.linalg.matrix_rank(X)
```

```
np.int64(2)
```

Inverse

Likewise, to compute the inverse of a matrix, there are two functions to use: `np.inv` and `np.pinv`. `np.inv` is used to compute the matrix inverse normally, while `np.pinv` using pseudo-inverse to compute inverse for matrices that might not have an inverse.

```
D = np.array([[4, 2], [-5, -3]])
D
```

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

```
Dinv = np.linalg.inv(D)
Dinv
```

```
array([[ 1.5,  1. ],
       [-2.5, -2. ]])
```

```
np.linalg.inv(Dinv)
```

```
array([[ 4.,  2.],
       [-5., -3.]])
```