`import numpy as np`

February 9, 2020

Week 6 of CS 41

Today in CS41: NumPy!

- Classes, Continued!
  - Inheritance
  - Magic Methods
- NumPy
  - N-dimensional arrays, constituent axes, and shapes.
  - Array indexing
  - Matrix Operations
  - Broadcasting
  - Reshaping
  - Parameter Fitting

A Brief Fireside Chat

(Mid-Quarter Assessment Debrief!)

Classes: Inheritance

Classes Can "Inherit" From Others

- By "inherit", we mean that attributes from the "base class" also become attributes in the "derived class".

```
class DerivedClassName(BaseClassName):
    # In reality you'd have an actual class here.
    # We've just added blank methods.
    def __init__(self):
        pass
    def method1(self):
        pass
```

A Brief Example

- By "inherit", we mean that attributes from the "base class" also become attributes in the "derived class".

```
class Canadian:
  def __init__(self, first_name, last_name):
    self.first_name = first_name
    self.last_name = last_name

  def print_name(self):
    print(self.firstname, self.lastname)

class BritishColumbian(Canadian):
    def print_name(self):
        print("{} is a British Columbian!".format(self.firstname))
```

Attribute Reference Moves Up the Chain

- The `print_name` method in the `BritishColumbian` class is unique – since we've redefined it in that class.
- But other methods, namely `__init__`, "inherit" from Canadian, so are the same as in the Canadian class.
- All Python classes inherit from the `object` class.

```
class Canadian:
  def __init__(self, first_name, last_name):
    self.first_name = first_name
    self.last_name = last_name

  def print_name(self):
    print(self.first_name, self.last_name)

class BritishColumbian(Canadian):
    def print_name(self):
        print("{} is a British Columbian!".format(self.first_name))
```

Classes: Multiple Inheritance

What Happens with Many Parents?

```
class Canadian:
  def __init__(self, first_name, last_name):
    self.first_name = first_name
    self.last_name = last_name

  def print_name(self):
    print("I'm {} {}, eh!".format(self.first_name, self.last_name))

class American:
  def __init__(self, first_name, last_name):
    self.first_name = first_name
    self.last_name = last_name

  def print_name(self):
    print("I'm {} {}, y'all!".format(self.first_name, self.last_name))

class BritishColumbian(Canadian, American):
    pass
```

What Happens with Many Parents?

```
michael = BritishColumbian("Michael", "Cooper")

# What will the following print out? Will the greeting end with "eh", or "y'all"?
michael.print_name()
```

- Recall that we defined the `BritishColumbian` class as follows:

```
class BritishColumbian(Canadian, American):
    pass
```

- Python will first search the `Canadian` class, then the `American` class for methods to bring into the class namespace. So it will print the greeting using "eh".
- Officially called "C3 Superclass Linearization" – more detail here!

Classes: Magic Methods

Finally adding some unicorn magic to the course!

Magic Methods

- Magic methods allow your class to interact seamlessly with Python's builtin functionality.
  - E.g. Python uses `__init__` to build classes.
- What else can we do?
- Can we make classes that behave like…
  - Iterators?
  - Lists?
  - Sets?
  - Comparables?

Python Uses Magic Methods

```
x = MagicClass()
y = MagicClass()
str(x)                  # => x.__str__()
x == y                  # => x.__eq__(y)

x < y                   # => x.__lt__(y)
x + y                   # => x.__add__(y)
iter(x)                 # => x.__iter__()
next(x)                 # => x.__next__()
len(x)                  # => x.__len__()
el in x                 # => x.__contains__(el)
```

- By overriding these methods in our own classes we can add unique functionality.
- These are just a few examples, there are many, many more magic methods in Python! (Link 1, Link 2)

A Short Example

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def rotate_90_counterclockwise(self):
        self.x, self.y = -self.y, self.x

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    def __str__(self):
        return "Point({0}, {1})".format(self.x, self.y)
```

```
A = Point(3, 4)
B = Point(-1, 2)
str(A)                      # => Point(3, 4)
print(A + B)                # => Point(2, 6)
```

`numpy`: Python's Numerical Computing Package

`numpy` – Where Are We Going?

- **This lecture consists of building blocks.**
- You likely won't leave this lecture with a deep knowledge of data science and machine learning.
- *However*, after this lecture, `numpy` should no longer be a stumbling block as you develop expertise in data science and machine learning.
- `numpy` sets the stage for our further study:
  - Matplotlib
  - Pandas
  - Machine Learning

Introducing a New Friend

- This lecture contains statistics and linear algebra. Some content may be beyond the scope of this class.
- This unicorn (which looks suspiciously like a matrix) will appear to flag these sections!

What is a Matrix?

- In mathematics, a:
  - Scalar is 1-D (one number)
  - Matrix is 2-D arrangement of numbers
  - Tensor is n-D arrangement of numbers.
- This is a (2-dimensional) matrix:

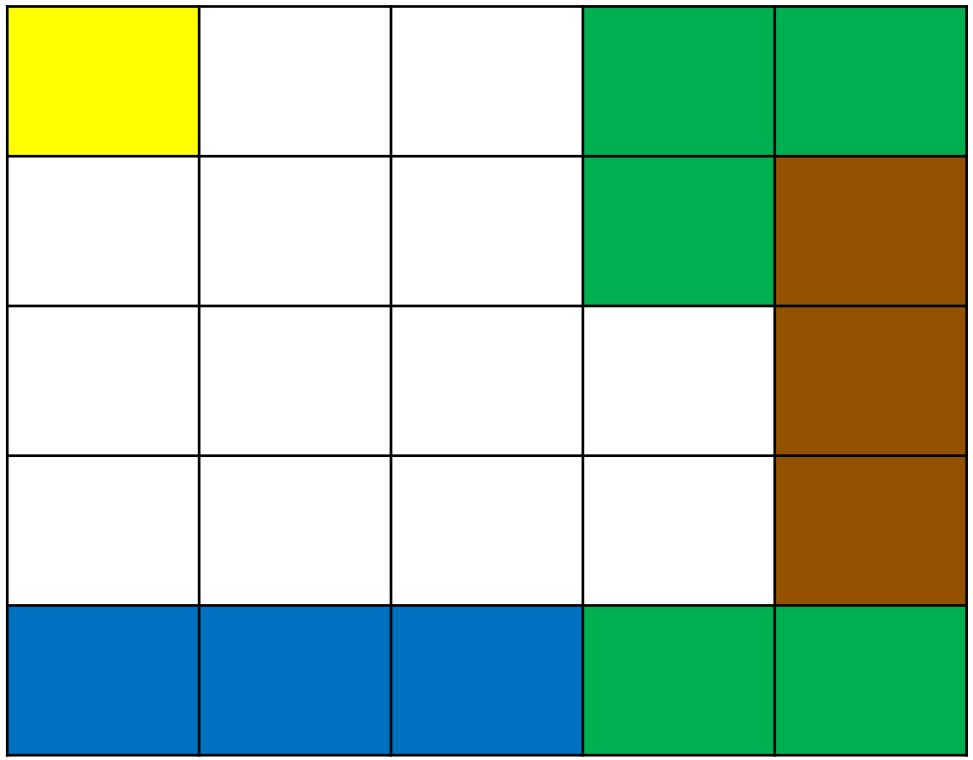
$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

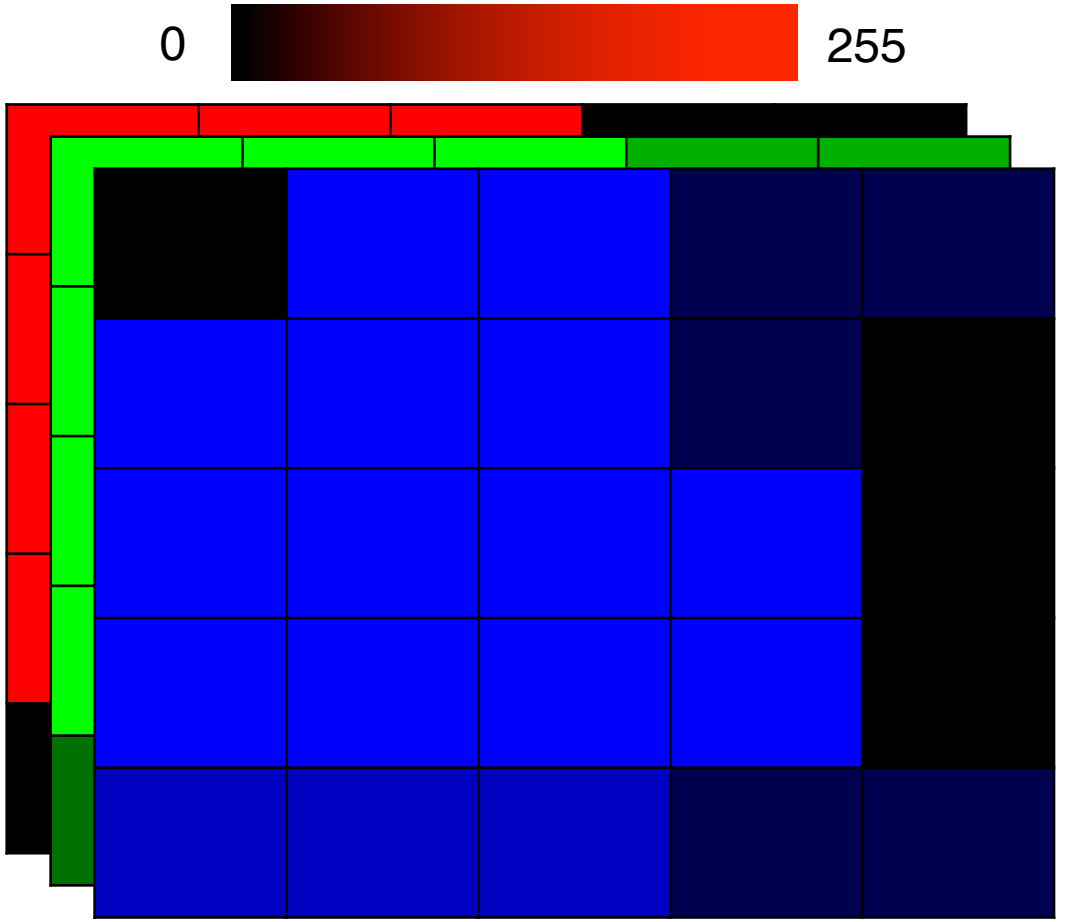
Why are Matrices Useful?

1. Simple atomic operations compose extremely complex, accurate models.
   - We're going to see one of these by the end of class!
2. Real-world data often fits cleanly into matrices.
   - Examples to follow this slide!

Images!

Key insight: an image is a 3D array of numbers.





Netflix!

- How can we store users' movie preferences?

	Little Women	Parasite	Joker	Avengers: Endgame
User 1	5	3	5	2
User 2	4	5	1	5
User 3	2	5	1	4
User 4	5	1	4	4

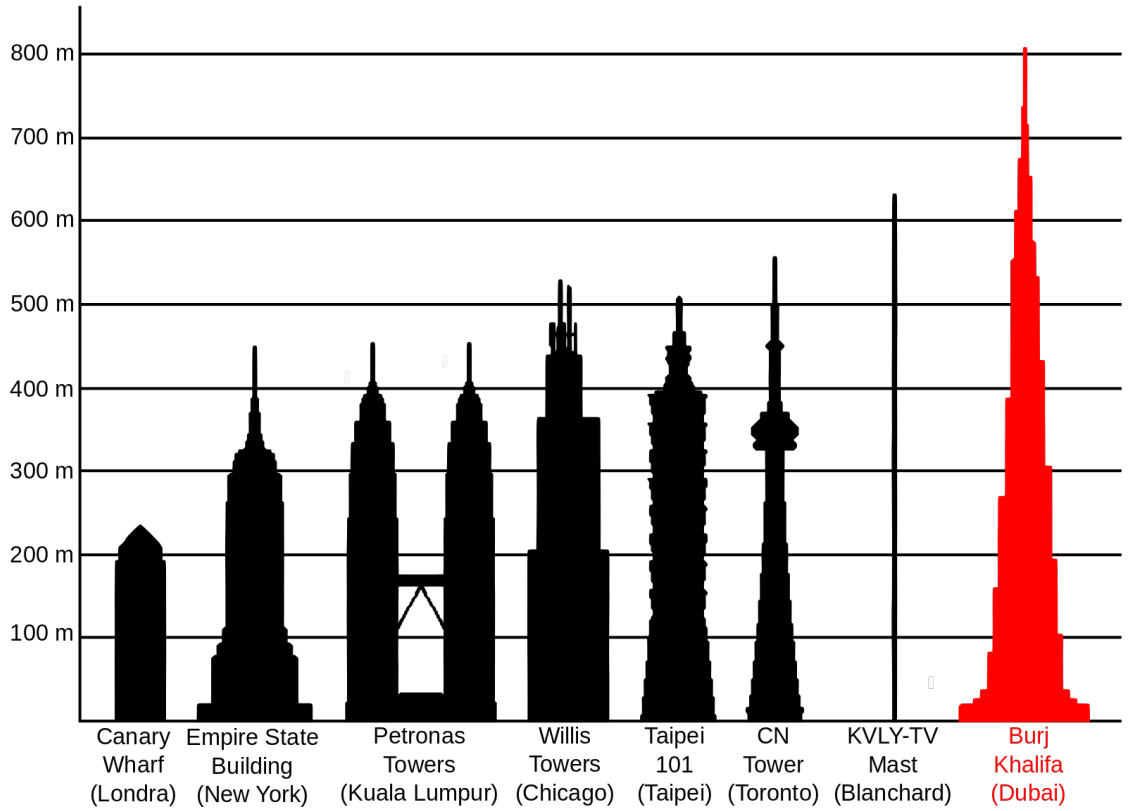
Netflix!

- How can we store users' movie preferences?
- How can we predict future preferences?

	Little Women	Parasite	Joker	Avengers: Endgame	1917
User 1	5	3	5	2	?
User 2	4	5	1	5	?
User 3	2	5	1	4	?
User 4	5	1	4	4	?

Dynamics!

- Modeling building sway in windstorms



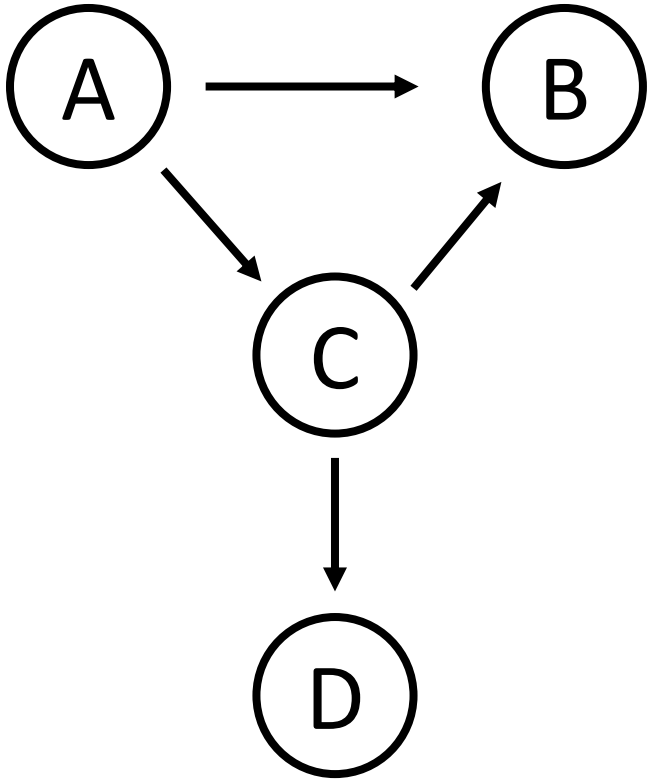
Shifts Building Along Axes… (m)

Wind Along Axes… (kph)

	x	y	z
x	0.05	0.001	0.001
y	0.001	0.05	0.001
z	0.001	0.001	0.05

Graphs!

- Storing and manipulating graphs!



Edges going to…

Edges going from…	A	B	C	D
A	0	1	1	0
B	0	0	0	0
C	0	1	0	1
D	0	0	0	0

`ndarray`

`numpy`'s core object!

ndarray

- Quick warning: how not to create an `ndarray`

```
# ndarrays do not accept generators! So be default, you can't do
# list comprehensions with them.
>>> np.array((x*x for x in range(5))
array(<generator object <genexpr> at 0x112b25900>, dtype=object)


# Instead, use np.fromiter to construct your array!
>>> np.fromiter((x*x for x in range(5))
array([0., 1., 4., 9., 16.])
```

`ndarray` (Continued!)

- All elements of an `ndarray` must be of homogenous type. (Even if that type is `object`).
  - This **does not** mean that you need to declare the type when constructing the array – Python will do this for you!

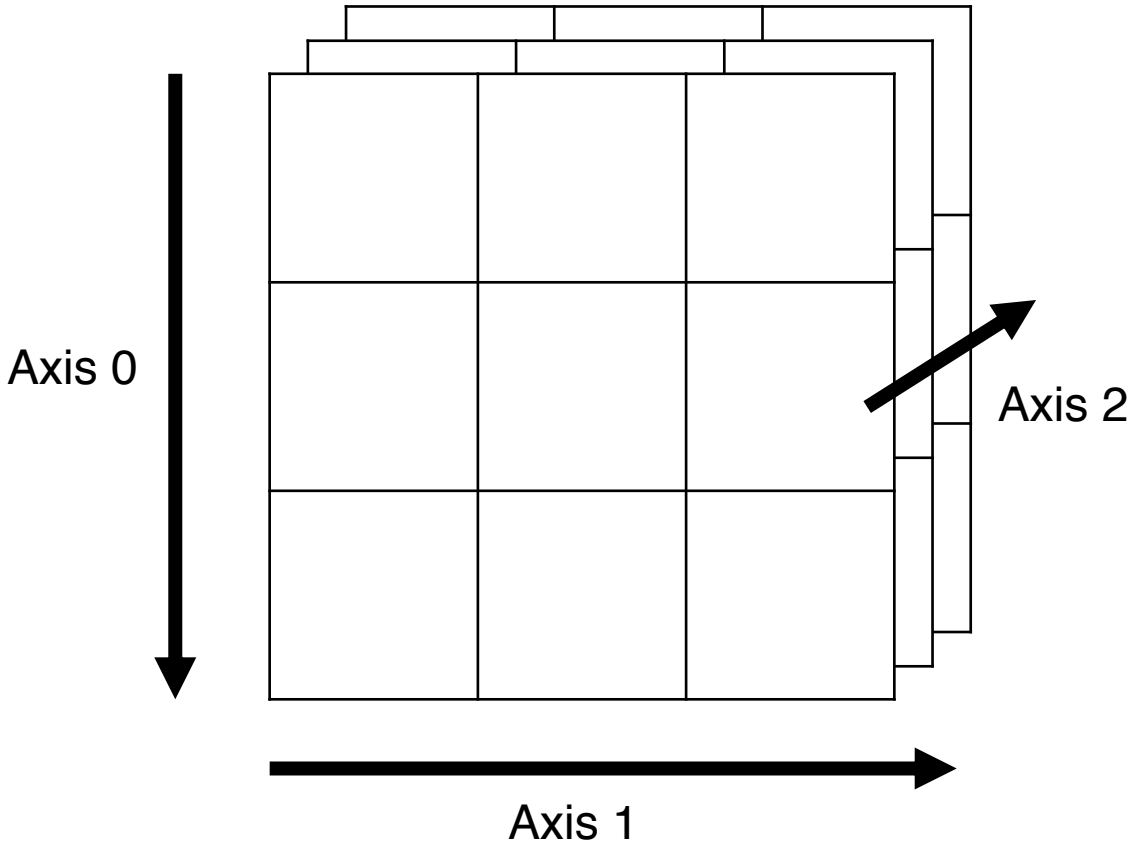
```
>>> # What happens if we try elements of hetrogenous types?
>>> arr = np.array([1.5, False, "Hello!"])
>>> arr
array(['1.5', 'False', 'Hello!'], dtype='<U32')
>>> # Catastrophe!
```

`ndarray` (Continued!)

- Why?
  - Storage – `numpy` can densely pack data types into memory for quicker access (this is covered more in CS107).
  - Speed – to avoid the cost of Python loops, many `numpy` operations are implemented in C++.
    - Therefore, valid types for elements of ndarray elements are the same as the types in C++.

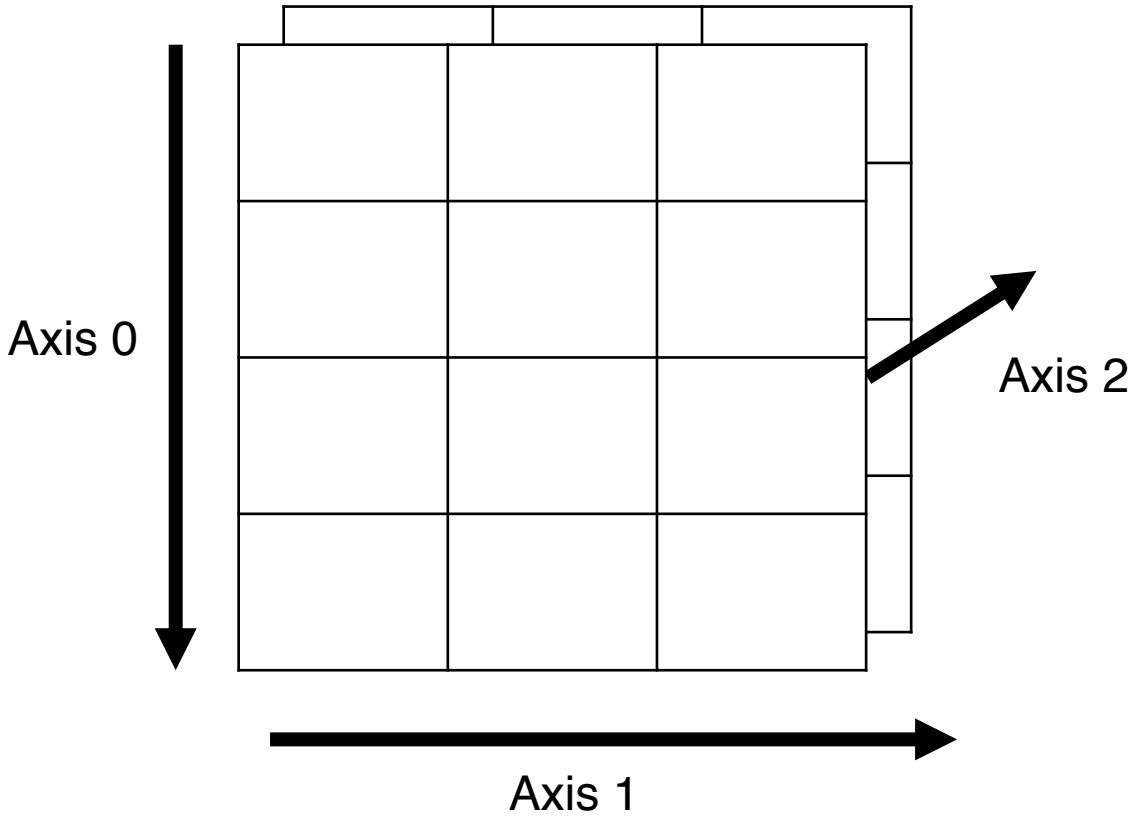
Axes and Shapes

- Each `numpy` array has axes numbered 1, 2, … N.



Axes and Shapes

- The *shape* of a `numpy` array is a tuple consisting of the number of elements along each axis.



What is the shape of this array?

`(4, 3, 2)`

```
# Return the shape of an array
# in tuple form
>>> arr_name.shape
```

Axes and Shapes

- The *shape* of a `numpy` array is a tuple consisting of the number of elements along each axis.
- Many `numpy` operations take place *along* an axis.

```
>>> arr = np.array([[1, 2, 3], [4, 5, 6]])
>>> arr
array([1, 2, 3],
      [4, 5, 6])
>>> np.sum(arr, axis=0)    # Along axis 0 - so sums vertically.
array([5, 7, 9])
>>> np.sum(arr, axis=1)    # Along axis 1 - so sums horizontally.
array([6, 15])
```

Array Indexing

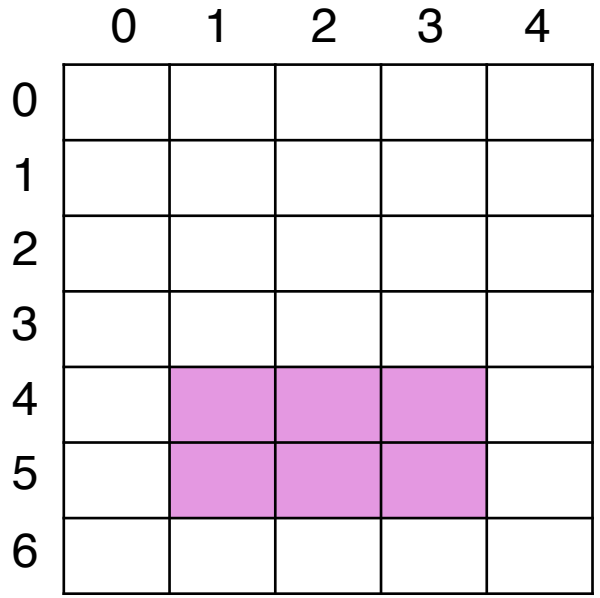
- Index along each axis with the same syntax as a Python list; comma-separate the indexing along each axis.

```
arr[index_axis_0, index_axis_1, ... index_axis_N]

# Example (with a 3D ndarray)
arr[:3, :, 2:5]
```

- The colon representing the indexing of axis 1 indicates that we take all elements from axis 1 in our segmentation.

Array Indexing

- Index along each axis with the same syntax as a Python list, comma-separate the indexing along each axis.
- E.g. how would we extract the highlighted values from the below array?

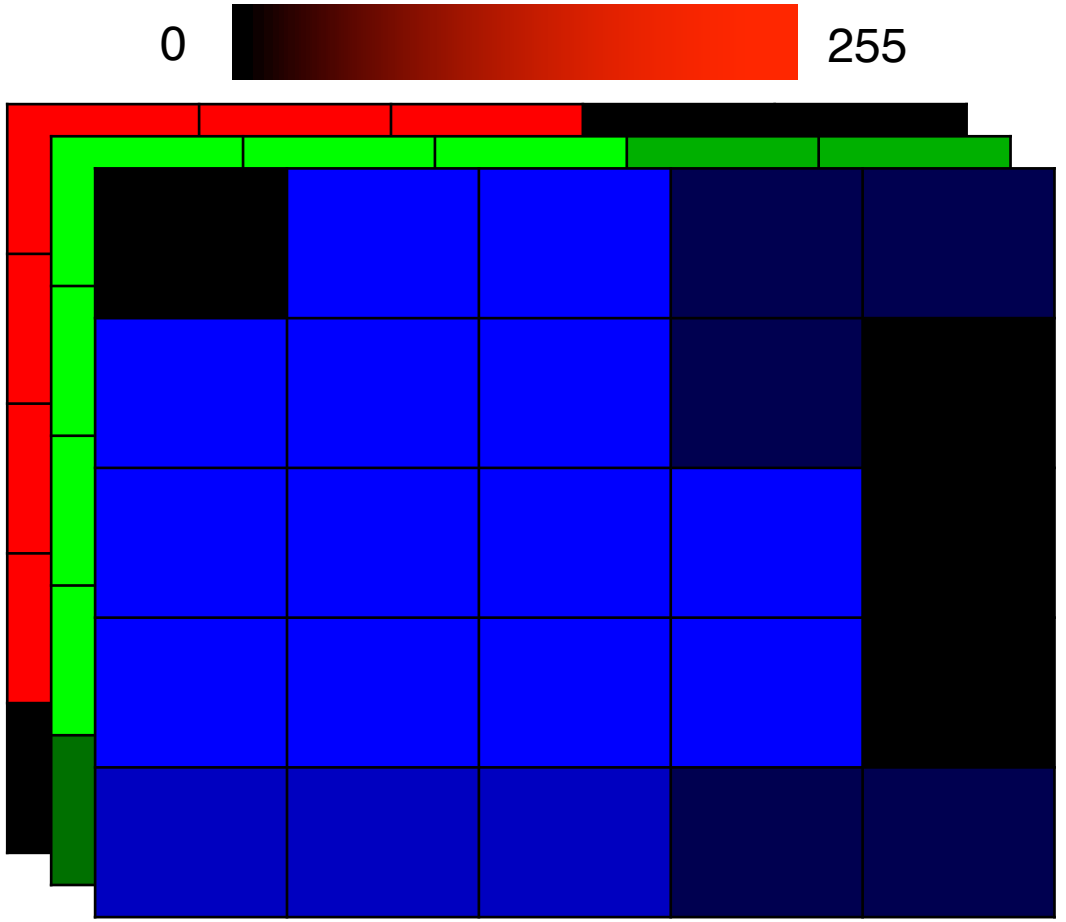



```
>>> segment = arr_name[4:6, 1:4]
```

(Remember that Python indexing is inclusive on the lower bound and exclusive on the upper bound!)

Example - Cropping an Image

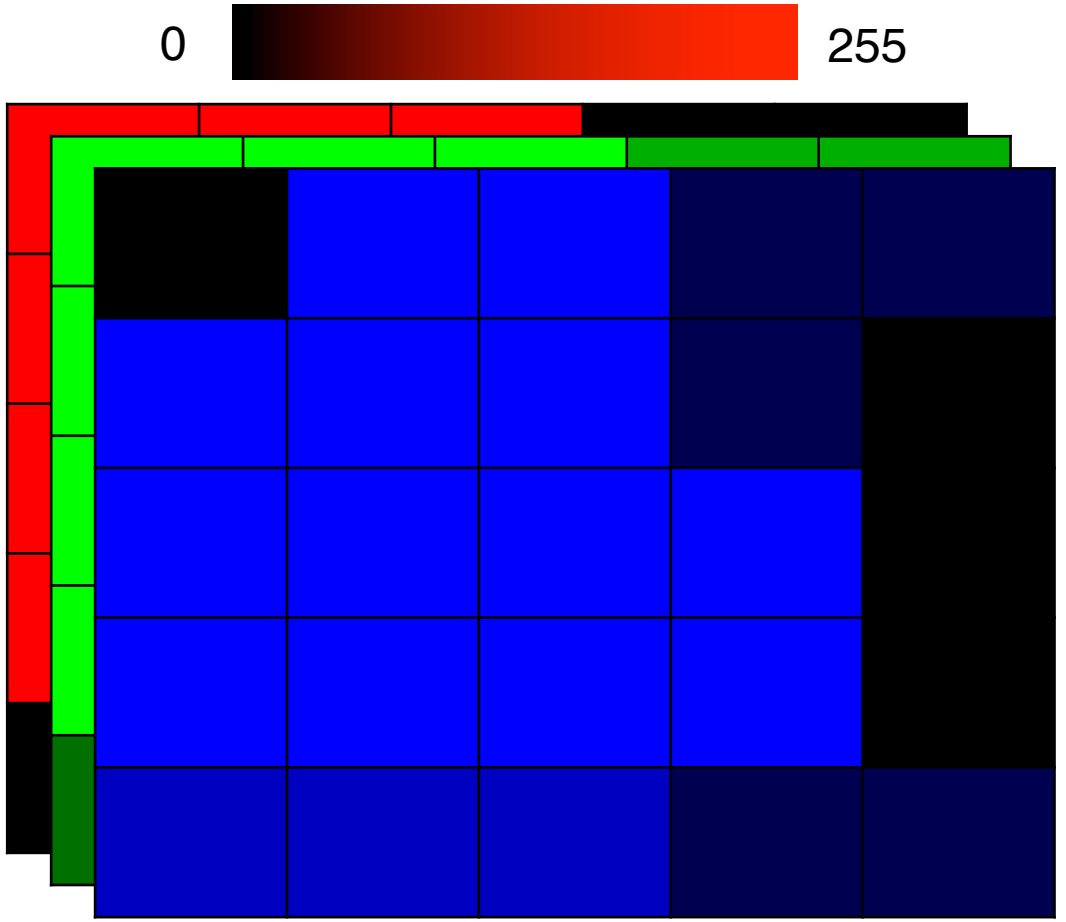
=



Along which axes would we need to index to obtain such a crop?

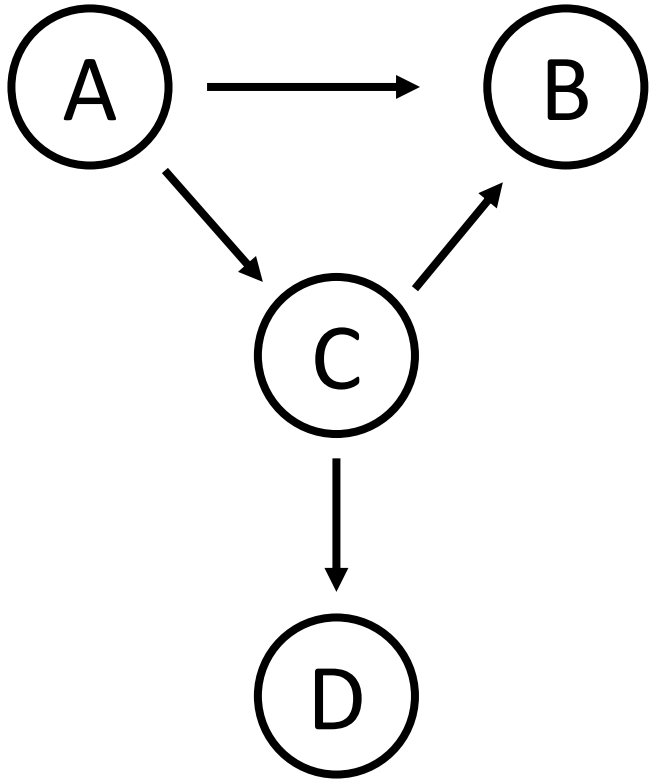
Example - Recolouring an Image

=



Along which axes would we need to index to obtain such a crop?

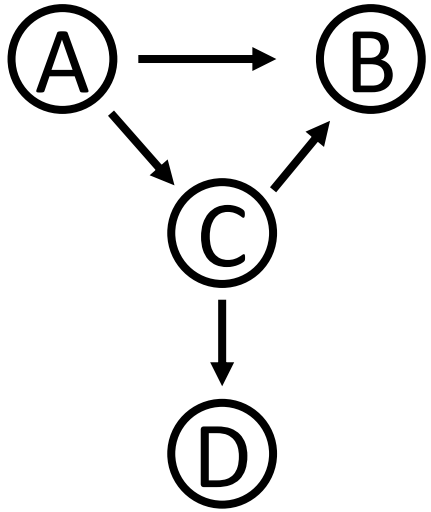
Graphs, Revisited!



Edges going to…

Edges going from…	A	B	C	D
A	0	1	1	0
B	0	0	0	0
C	0	1	0	1
D	0	0	0	0

Graphs, Revisited!



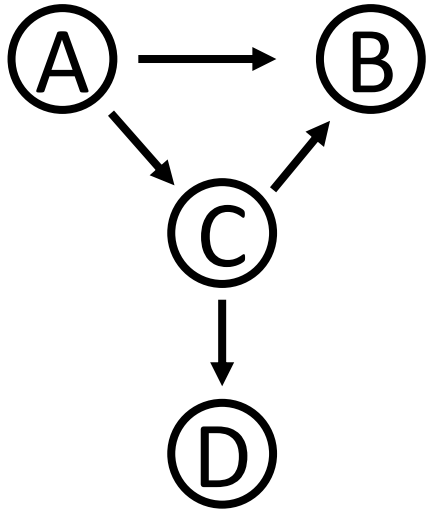
Edges going to…

Edges going from…	A	B	C	D	
A	0	1	1	0	2
B	0	0	0	0	0
C	0	1	0	1	2
D	0	0	0	0	0

How many outgoing edges does each node have?

```
outgoing_edges = np.sum(A, axis=1)
```

Graphs, Revisited!



How many incoming edges does each node have?

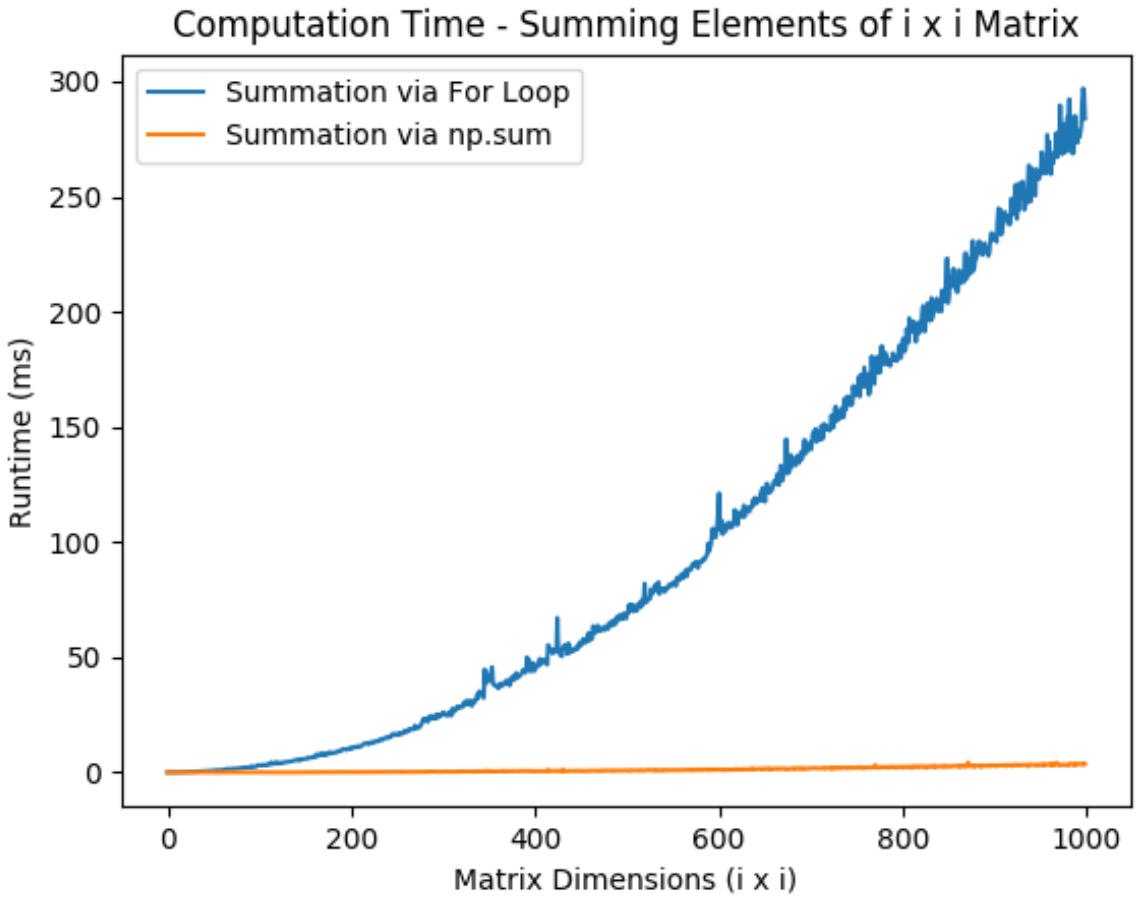
```
incoming_edges = np.sum(A, axis=0)
```

Edges going to…

Edges going from…

	A	B	C	D	
A	0	1	1	0	2
B	0	0	0	0	0
C	0	1	0	1	2
D	0	0	0	0	0
	0	2	1	1	

Iterative vs. Vectorized Runtimes



Note: though np.sum does grow exponentially with respect to matrix dimension, it just grows at an insignificant rate when compared to the iterative implementation (even `numpy` cannot perform O(1) matrix summation).

Elementwise Matrix Operations

- `numpy` supports four main elementwise operators on arrays.

```
# Compute the elementwise sum array1 + array2
np.add(array1, array2)

# Compute the elementwise difference array1 â€šÀÓ array2
np.subtract(array1, array2)

# Compute the elementwise product array1 * array 2
np.multiply(array1, array2)

# Compute the elementwise quotient array1 / array2
np.divide(array1, array2)
```

- Here, **`array1`** and **`array2`** must be the same shape (or *broadcastable* into the same shape) for the operations to work.

Matrix Broadcasting

- Let $\mathbf{A} \in \mathbb{R}^{3 \times 2}$ and $\mathbf{B} \in \mathbb{R}^{1 \times 2}$. Then, $\mathbf{A} + \mathbf{B}$ is

$$\begin{bmatrix} 5 & 1 \\ 2 & 0 \\ 4 & 3 \end{bmatrix} + \begin{bmatrix} 1 & -1 \end{bmatrix} = \begin{bmatrix} 5 & 1 \\ 2 & 0 \\ 4 & 3 \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ 1 & -1 \\ 1 & -1 \end{bmatrix}$$

$$= \begin{bmatrix} 6 & 0 \\ 3 & -1 \\ 5 & 2 \end{bmatrix}$$

- Here, we say that $\mathbf{B}$ has been *broadcast along axis 0* in order to make the dimensions align.

When are Matrices Broadcastable?

- The ith dimensions of matrices $\mathbf{A}$ and $\mathbf{B}$ must:
  - Be equal, or
  - One must be equal to one.
- Intuition: taking a slice of matrix per axis and propagating it.

- Quiz: Are these matrices broadcastable? What is the output size?
  - $\mathbf{A} \in \mathbb{R}^{3 \times 2}$ and $\mathbf{B} \in \mathbb{R}^{1 \times 2}$
    - Yes! Output size is $\mathbf{C} \in \mathbb{R}^{3 \times 2}$
  - $\mathbf{A} \in \mathbb{R}^{2 \times 5}$ and $\mathbf{B} \in \mathbb{R}^{3 \times 5}$
    - Not broadcastable!
  - $\mathbf{A} \in \mathbb{R}^{1 \times 3 \times 4 \times 2}$ and $\mathbf{B} \in \mathbb{R}^{5 \times 1 \times 4 \times 1}$
    - Yes! Output size is $\mathbf{C} \in \mathbb{R}^{5 \times 3 \times 4 \times 2}$

When are Matrices Broadcastable?

- The ith dimensions of matrices $\mathbf{A}$ and $\mathbf{B}$ must:
  - Be equal, or
  - One must be equal to one.
- Intuition: taking a slice of matrix per axis and propagating it.

- Quiz: Are these matrices broadcastable? What is the output size?
  - $\mathbf{A} \in \mathbb{R}^{3 \times 2}$ and $\mathbf{B} \in \mathbb{R}^{1 \times 2}$
    - Yes! Output size is $\mathbf{C} \in \mathbb{R}^{3 \times 2}$
  - $\mathbf{A} \in \mathbb{R}^{2 \times 5}$ and $\mathbf{B} \in \mathbb{R}^{3 \times 5}$
    - Not broadcastable!
  - $\mathbf{A} \in \mathbb{R}^{1 \times 3 \times 4 \times 2}$ and $\mathbf{B} \in \mathbb{R}^{5 \times 1 \times 4 \times 1}$
    - Yes! Output size is $\mathbf{C} \in \mathbb{R}^{5 \times 3 \times 4 \times 2}$
  - What about $\mathbf{A} \in \mathbb{R}^{4 \times 3}$ and $\mathbf{B} \in \mathbb{R}^{5 \times 4 \times 3}$? (This is tricky!)
    - Yes! $\mathbf{A} \in \mathbb{R}^{4 \times 3}$ is the same as $\mathbf{A} \in \mathbb{R}^{1 \times 4 \times 3}$ so `numpy` automatically adds the leading 1.

Reshaping Matrices

- To transpose a matrix (convert columns into rows and vice versa), `numpy` provides a transpose function.

```
# Takes in A, an (m, n) matrix, returns A^T, an (n, m) matrix.
>>> np.transpose(A)

# Equivalent shortcut (yes, numpy will parse this correctly).
>>> A.T
```

Reshaping Matrices

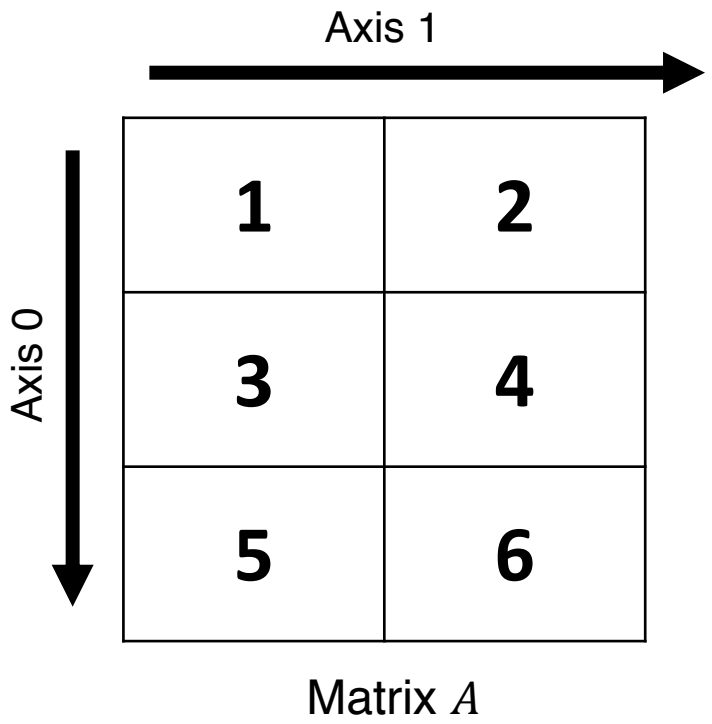
- If a matrix A contains $r = mn$ elements then the matrix A can be reshaped into an (m, n) matrix.

```
# Reshapes a matrix A into an (m, n) matrix
np.reshape(A, (m, n))
```

- *How are elements assigned to the reshaped matrix?*
  - E.g. $\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$ and $\begin{bmatrix} 2 & 1 \\ 4 & 3 \\ 6 & 5 \end{bmatrix}$ are both valid (3, 2) matrices containing the elements of the matrix $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$. But they're clearly not the same matrix!
  - This is mostly a piece of history, but it's neat to know!

C-Style vs. FORTRAN-Style

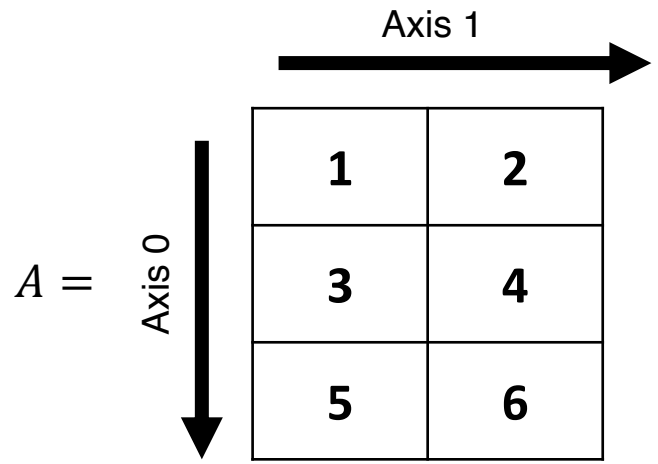
```
axis_0, axis_1 = A.shape

# C-Style
for i in axis_0:
    for j in axis_1:
        print(A[i, j])          # Prints 1, 2, 3, 4, 5, 6

# FORTRAN-Style
for i in axis_1:
    for j in axis_0:
        print(A[j, i])          # Prints 1, 3, 5, 2, 4, 6
```



(Un)raveling a Matrix

- The function `np.ravel` unrolls an array into a 1-D array, using either C-Style or FORTRAN-Style iteration.

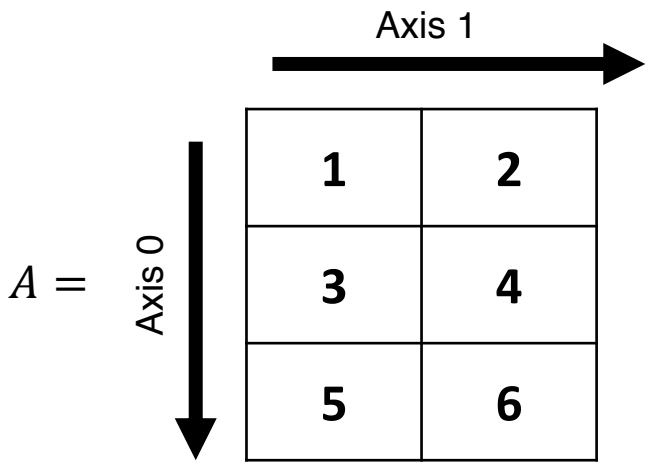
```
# Using the A from the previous slide, call np.ravel

>>> np.ravel(A, order="C")
array([1, 2, 3, 4, 5, 6])

>>> np.ravel(A, order="F")
array([1, 3, 5, 2, 4, 6])
```



Back to Reshaping

- The function `np.reshape` can be thought of as the composition of `np.reshape` and `np.ravel`.

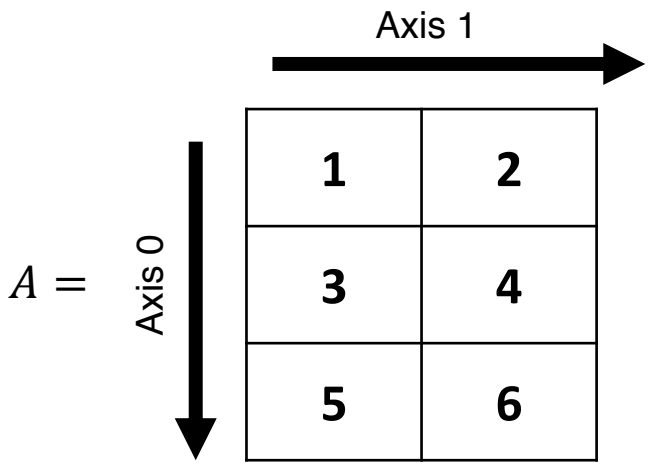
```
# Using the FORTRAN-Style ordering
>>> np.reshape(A, (2, 3), order="F")

# This is equivalent to:
>>> unraveled = np.ravel(A, order="F")
array([1, 3, 5, 2, 4, 6])

>>> np.reshape(unraveled, (2, 3), order="F")
array([1, 5, 4],
      [3, 2, 6])

# Note the... silly ordering - the elements of unraveled are first added
along axis 0, then along axis 1, as is consistent with FORTRAN-Style
ordering.
```



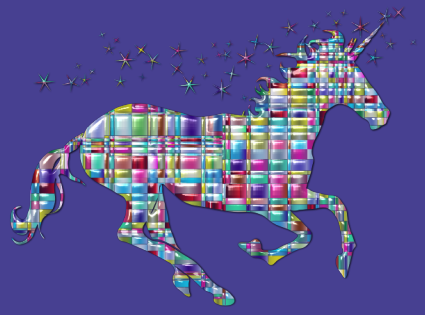
Back to Reshaping

- The function `np.reshape` can be thought of as the composition of `np.reshape` and `np.ravel`.

```
# Using the C-Style ordering (default)
>>> np.reshape(A, (2, 3))

# This is equivalent to:
>>> unraveled = np.ravel(A, order="C")
array([1, 2, 3, 4, 5, 6])

>>> np.reshape(unraveled, (2, 3))
array([1, 2, 3],
      [4, 5, 6])

# First add along axis 1, then along axis 0.
```



Matrix Multiplication

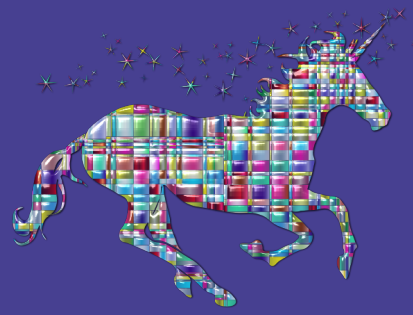


- Several different types of matrix multiplication provided by `numpy` – it's important not to get these mixed up!

```
# Compute the inner product of two arrays
np.dot(array1, array2)

# Compute the outer product of two arrays
np.outer(array1, array2)

# Compute the elementwise product array1 * array 2
np.matmul(array1, array2)
# (Equivalent shortcut: array1 @ array2)
```

- In 2-D space, `np.matmul` and `np.dot` are equivalent operations.
  - In higher dimensions, `np.matmul` performs tensor multiplication with broadcasting; `np.dot(A, B)` takes the inner product over the last axis of `A` and the second last axis of `B`.
  - In English: use `np.matmul` for multiplying tensors.

More Matrix Operations!



```
# numpy is full of functions to perform matrix operations.

# (If you don't know all these concepts, don't worry! But those
# familiar with linear algebra may be interested).

np.linalg.norm(A)               # Compute the norm of matrix A

np.linalg.eig(A)                # Tuple: (eigenvalues, eigenvectors) of A

np.linalg.det(A)                # Determinant of matrix A

np.linalg.matrix_rank(A)        # Rank of A

np.linalg.qr(A)                 # Tuple: (Q, R) factorization of A

np.linalg.svd(A)                # Tuple: (U, S, V) SVD of A.

np.linalg.matrix_power(A, k)    # Returns the matrix exponential A^k
```

Statistical Methods

- If we interpret `numpy` arrays as collections of data point, `numpy` provides functions for statistical analyses on the data.

```
np.mean(A, axis=i)                          # Mean of elements of A along axis i

np.average(A, axis=i, weights=None)         # Weighted average of A along axis i
                                            # (weights argument is optional).

np.std(A, axis=i)                           # Std. dev. of A along axis i

np.median(A, axis=i)                        # Median of A along axis i

np.var(A, axis=i)                           # Variance of A along axis i
```

- Remember! These functions work *along* their axis (as with the earlier sum example).

Example – Mean Reversion in Markets

	03-Feb	04-Feb	05-Feb	06-Feb	07-Feb	10-Feb
AMZN	2010.60	2029.88	2071.02	2041.02	2041.99	2,038.90
GS	238.36	242.88	244.99	245.35	239.75	242.27
TSLA	673.69	882.96	823.26	699.92	730.55	762.08
BTC	9617.82	9726.00	9793.07	9863.89	10131.58	9,826.47

```
predictions = np.mean(A, axis=1)
```

Halftime

Logistics

Assignment 1	Grades will be released later today – great job everyone!
Assignment 2	Due February 21, 2020, at 11:59PM.
Labs	Lab on Wednesday – bring a charged computer!

Attendance Form

Link: http://iamhere.stanfordpython.com

Code: **MAGICALUNICORNMETHODS**

If you're not here, fill out this alternative form:

https://bit.ly/2O4Gpyr

The Parameter Fitting Problem

Our first machine learning task!

Let's Play a Game!

X	Y
1	2
2	4
3	6
4	?
-1	?

Let's Play a Game!

X	Y
1	2
2	4
3	6
4	8
-1	-2

How Did You Do That?

- Slight mathematical formalization:
- "Well, there's some θ (here, it's 2) and I've noticed that $y_{\mathrm{i}} = x_i\theta$. So to predict each y_{i}, I just multiplied it by this value that I determined."

X	Y
1	2
2	4
3	6
4	8
-1	-2

Let's Play a Game!

X	Y
1	2.001
2	3.998
3	6.012
4	?
-1	?

Let's Play a Game!

X	Y
1	2.001
2	3.998
3	6.012
4	8
-1	-2

Maybe in reality, these values are 7.994 and -2.0131. Does it matter?

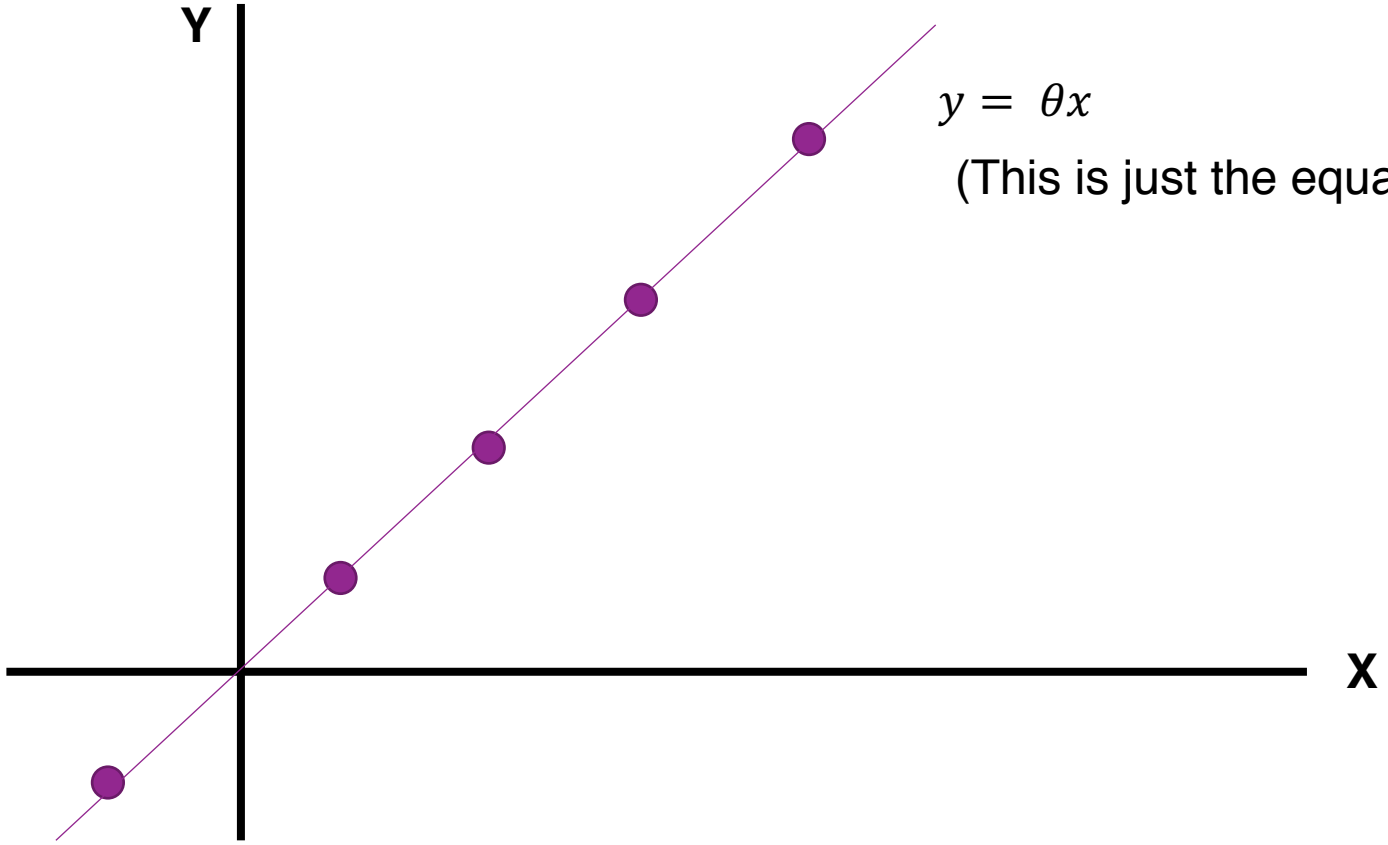
How Did You Do That?

- Slight mathematical formalization:
- "Well, there's some θ (here, it's 2) and I've noticed that $y_{\mathrm{i}} \approx x_i\theta$. So to predict each y_{i}, I just multiplied it by this value that I determined."
- You accepted a little bit of error in exchange for a relatively simple model.
- **This is machine learning:** finding patterns in data.

X	Y
1	2.001
2	3.998
3	6.012
4	8
-1	-2

Another Perspective

X	Y
1	2.001
2	3.998
3	6.012
4	8
-1	-2



$y = \theta x$

(This is just the equation for a line!)

Let's Play a Higher-Dimensional Game!

X_1	X_2	Y
1	1	0.5
1	2	0
2	1	1.5
2	2	1
1	-1	?

Let's Play a Higher-Dimensional Game!

X_1	X_2	Y
1	1	0.5
1	2	0
2	1	1.5
2	2	1
1	-1	1.5

This One Was Trickier!

- What did we do this time? We can formalize this as follows:

$$X_{i1} - 0.5X_{i2} = Y_i$$

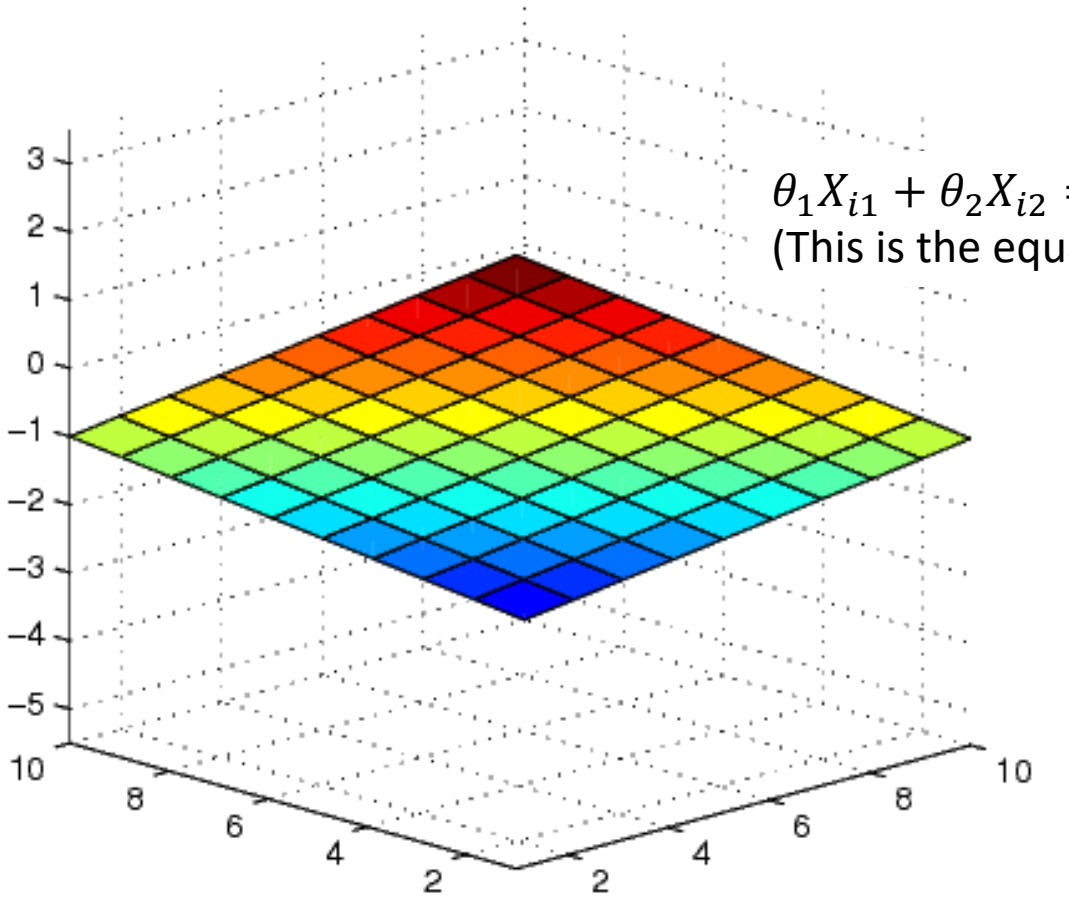
$$\theta_1 X_{i1} + \theta_2 X_{i2} = Y_i$$

- We can say that Y_1 is a *linear combination* of the X_{i1}'s.
- Can we extend this thinking to even higher dimensions? (Yes!)

X_1	X_2	Y
1	1	0.5
1	2	0
2	1	1.5
2	2	1
1	-1	1.5

Another Perspective

X_1	X_2	Y
1	1	0.5
1	2	0
2	1	1.5
2	2	1
1	-1	1.5



$\theta_1 X_{i1} + \theta_2 X_{i2} = Y_i$
(This is the equation for a plane!)

A Concrete Example

- Assume that we want to build a model to predict a student's grade in a course given a dataset of past students' performances.
- $A \in \mathbb{R}^{m \times n}$ is the *feature matrix*. The rows represent past students, the columns represent features.
- $y \in \mathbb{R}^{n}$ is the *labels vector*. This is the vector in which we list out the past students' grades in the class (we've expressed the percentage in decimal).

	Grade in prerequisite class	Hours per week spent on this class	...	Number of times per week they attend OH
Student 0	0.88	8	...	2
Student 1	0.95	15	...	1
⋮	⋮	⋮	⋱	⋮
Student $m-1$	0.78	5	...	0

	Grade in current class
Student 0	0.9
Student 1	0.85
⋮	⋮
Student $m-1$	0.69

Our Goal

- We want to learn some set of $\theta_1, \theta_2, \ldots \theta_n$ such that:

(Row 1): $\theta_1(0.88) + \theta_2(8) + \cdots + \theta_n(2) \approx 0.9$

(Row 2): $\theta_1(0.95) + \theta_2(15) + \cdots + \theta_n(1) \approx 0.85$

(Row 3): $\theta_1(0.78) + \theta_2(5) + \cdots + \theta_n(0) \approx 0.69$

Where did these come from?

	Grade in prerequisite class	Hours per week spent on this class	...	Number of times per week they attend OH
Student 0	0.88	8	...	2
Student 1	0.95	15	...	1
⋮	⋮	⋮	⋱	⋮
Student $m-1$	0.78	5	...	0

	Grade in current class
Student 0	0.9
Student 1	0.85
⋮	⋮
Student $m-1$	0.69

This One Was Trickier!

- What did we do this time? We can formalize this as follows:

$$X_{i1} - 0.5X_{i2} = Y_i$$

$$\theta_1 X_{i1} + \theta_2 X_{i2} = Y_i$$

- We can say that Y_1 is a *linear combination* of the X_{i1}'s.
- Can we extend this thinking to even higher dimensions? (Yes!)

X_1	X_2	Y
1	1	0.5
1	2	0
2	1	1.5
2	2	1
1	-1	1.5

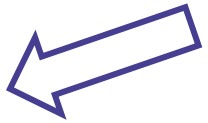
Our Goal

- We want to learn some set of $\theta_1, \theta_2, \ldots \theta_n$ such that:

(Row 1): $\theta_1(0.88) + \theta_2(8) + \cdots + \theta_n(2) \approx 0.9$

(Row 2): $\theta_1(0.95) + \theta_2(15) + \cdots + \theta_n(1) \approx 0.85$

(Row 3): $\theta_1(0.78) + \theta_2(5) + \cdots + \theta_n(0) \approx 0.69$



Why do the θ's need to be the same across all our data points?

	Grade in prerequisite class	Hours per week spent on this class	...	Number of times per week they attend OH
Student 0	0.88	8	...	2
Student 1	0.95	15	...	1
⋮	⋮	⋮	⋱	⋮
Student $m-1$	0.78	5	...	0

	Grade in current class
Student 0	0.9
Student 1	0.85
⋮	⋮
Student $m-1$	0.69

What's a "Good Fit"?

- Assume that we've already learned our θ's.

- What is our "prediction" (for, say, Student 1)?

$$P_1 = \theta_1(0.88) + \theta_2(8) + \cdots + \theta_n(2)$$

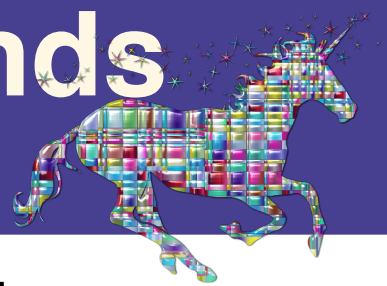
(We just named it P_1, this isn't a prior naming convention).

- What is the "true value"?

$$Y_1 = 0.9$$

- So "error for this student" is $P_1 - Y_1$…sort of.

The Euclidean Norm in 30 Seconds



- We measure the "badness of fit" as follows (if you don't know why, come see me after class!):

$$\sqrt{(Y_1 - P_1)^2 + (Y_2 - P_2)^2 + \cdots + (Y_m - P_m)^2}$$

- If this term is small, it means we've attained a vector of predictions that is very close to the actual values in our dataset.

What Does This All Mean Again?

- We want to learn some set of $\theta_1, \theta_2, \ldots \theta_n$ such that:

(Row 1): $P_1 = \theta_1(0.88) + \theta_2(8) + \cdots + \theta_n(2) \approx 0.9$

(Row 2): $P_2 = \theta_1(0.95) + \theta_2(15) + \cdots + \theta_n(1) \approx 0.85$

(Row 3): $P_3 = \theta_1(0.78) + \theta_2(5) + \cdots + \theta_n(0) \approx 0.69$

	Grade in prerequisite class	Hours per week spent on this class	...	Number of times per week they attend OH
Student 0	0.88	8	...	2
Student 1	0.95	15	...	1
⋮	⋮	⋮	⋱	⋮
Student $m-1$	0.78	5	...	0

	Grade in current class
Student 0	0.9
Student 1	0.85
⋮	⋮
Student $m-1$	0.69

This Is – Wait For It! – One `numpy` Command

```
# Returns solution vector, θ
>>> theta = np.linalg.lstsq(A, y)
array([4.3155, 2.1284, ... -1.9321])
```

Summary

- Classes, Continued!
  - Inheritance
  - Magic Methods
- NumPy
  - N-dimensional arrays, constituent axes, and shapes.
  - Array indexing
  - Matrix Operations
  - Broadcasting
  - Reshaping
  - Parameter Fitting