# Data Science

## Introduction to NumPy

Chris Jermaine, Risa Myers, Marmar Orooji

Rice University

# NumPy

- NumPy is a package for numerical computing with Python
- Supports super-fast math, statistics (implemented in C)
- Core of the package – multi-dimensional array object
  - Used to store vectors, matrices, tensors
- Load with

```
import numpy as np
```

# NumPy arrays

- Grid of values all of the same type
- Supports vectorized programming – great for Big Data!

# Vectorization programming motivation

- Writing control flow code in data science programming is BAD
- Python is interpreted
  - Time for each statement execution is generally large
  - In data science, you have a lot of data
  - So this code can take a long time:

```
for b in range(0, BIG):
    a[b] = b

sum = 0
for b in a:
    sum += a[b]
```

- Fewer statements executed, even if the work is the same...
  - ...means better performance!

# To reduce the number of statements

- Use NumPy arrays where possible
- Goal: use one line of Python to process an entire array!
- Some guidelines
  - Try to replace dictionaries with NumPy arrays
  - Try to replace loops with bulk array operations
    - Backed by efficient, low-level implementations
  - This is known as “vectorized” programming

# Vectorization in NumPy

- Numpy is able to apply computations on the entire array without the need to explicitly iterate through elements
- How?
  - NumPy arrays are homogenous (contain data of a single type)
  - This allows NumPy to delegate the implementation of array operations and looping internally to optimized compiled C code
  - This delivers the high performance of C in Python
  - Results in tremendous speedup in mathematical computation

# Create a 1D array

1. `np.array([])` – creates an array from a Python list

```
>>> np.array([0, 8, 5, 2])

array([0, 8, 5, 2])
```

2. `np.arange(start, stop, step)` – same as Python `range`

```
>>> np.arange(1,11,2)

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

3. `np.linspace(start, stop, num=50)` – returns evenly spaced numbers over the specified interval

```
>>> np.linspace(0., 20., 5)

array([ 0.,  5., 10., 15., 20.])
```

# 1D array indexing

- Similar to Python lists
- Starts at 0

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

index 0 1 2 3 4

# 1D array selection

```
>>> a = np.arange(1, 11, 2)
array([1, 3, 5, 7, 9])
```

- By index

```
>>> a[2]
5
```

- By slice

```
>>> a[1:-1]
array([3, 5, 7])
```

- By list of indices

```
>>> a[[2, 4]]
array([5, 9])
```

- By condition

```
>>> a[a<5]
array([1, 3])
```

# Create a 2D array

- Create a 2D array from a 2D Python list

```
>>> np.array([[0, 8, 5], [1, 4, 7]])

array([[0, 8, 5],
       [1, 4, 7]])
```

# 2D array indexing

- Indexing rows and columns

$$
\begin{array}{lccccc}
 & \multicolumn{3}{c}{\text{column indices}} & & \\
 & 0 & 1 & 2 & & \text{row indices} \\
\texttt{array([[} & \texttt{0,} & \texttt{8,} & \texttt{5],} & & 0 \\
\texttt{[} & \texttt{1,} & \texttt{4,} & \texttt{7]])} & & 1
\end{array}
$$

# 2D array selection

```
>>> a = np.array([[0, 8, 5], [1, 4, 7]])
array([[0, 8, 5],
       [1, 4, 7]])
```

- By index

```
>>> a[1, 2]
7
```

  - Row 1
  - Column 2

- By slice

```
>>> a[:, 1:]
array([[8, 5],
       [4, 7]])
```

  - All rows
  - Column 1 to the end

# 2D array selection by list of indices

```
>>> a = np.array([[0, 8, 5], [1, 4, 7]])
array([[0, 8, 5],
       [1, 4, 7]])
```

- Elements (0,0), (1,0), and (1,2)

```
>>> a[[0,1,1],[0,0,2]]
array([0, 1, 7])
```

- All rows, columns 0 and 2

```
>>> a[:,[0,2]]
array([[0, 5],
       [1, 7]])
```

- Row 1, columns 0 and 2

```
>>> a[1,[0,2]]
array([1, 7])
```

# Accessing subparts of arrays

- First we create a 2D array
- `row_stack` builds an array by vertically stacking the provided arrays

```
>>> a1 = np.arange(1, 6, 1)
>>> a2 = np.arange(2, 7, 1)
>>> a3 = np.arange(3, 8, 1)
>>> a = np.row_stack ((a1, a2, a3))
>>> a

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

row

a = [[ 1 , 2 , 3 , 4 , 5 ], 0
[ 2 , 3 , 4 , 5 , 6 ], 1
[ 3 , 4 , 5 , 6 , 7 ]] 2

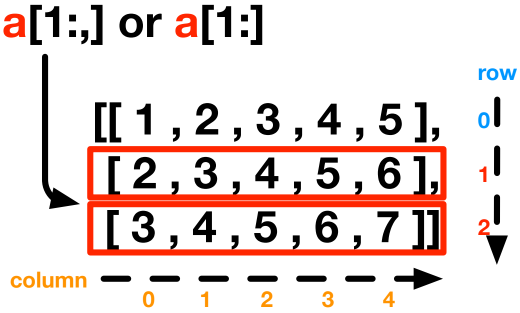
column 0 1 2 3 4

# Accessing subparts of arrays (cont)

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

>>> a[1:,]

array([[2, 3, 4, 5, 6],
       [3, 4, 5, 6, 7]])

>>> a[1:]
array([[2, 3, 4, 5, 6],
       [3, 4, 5, 6, 7]])
```

- Indices start with 0
- Gets rows 1, 2, and so on

**Elements of a from element 1 to the end**



# Accessing subparts of arrays (cont)

- Say we want the last row:

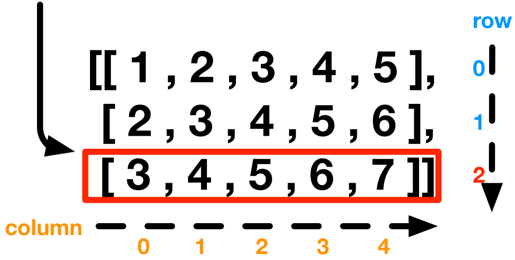
```
>>> a[2:3,]
array([[3, 4, 5, 6, 7]])

>>> a[2:3]
array([[3, 4, 5, 6, 7]])
```

- Note: returns a 2D array. Want a vector?

```
>>> a[2:3][0]

array([3, 4, 5, 6, 7])
```

**Elements of a that are >= 2 but < 3**

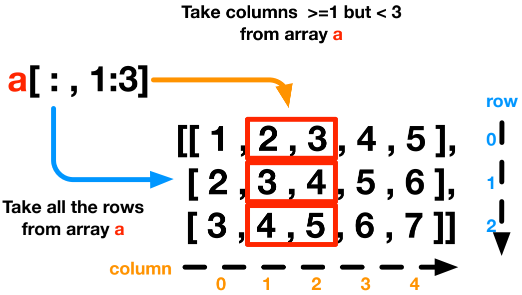
**a[2:3,] or a[2:3]**



# Accessing subparts of arrays (cont)

- Now we want the second, third columns, all rows:

```
>>> a[:,1:3]

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

>>> a[:,np.array((1,2))]

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



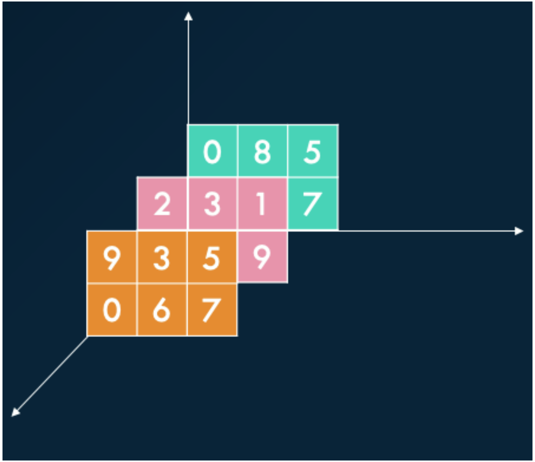
# Create a 3D array

- Create a 3D array from a 3D Python list

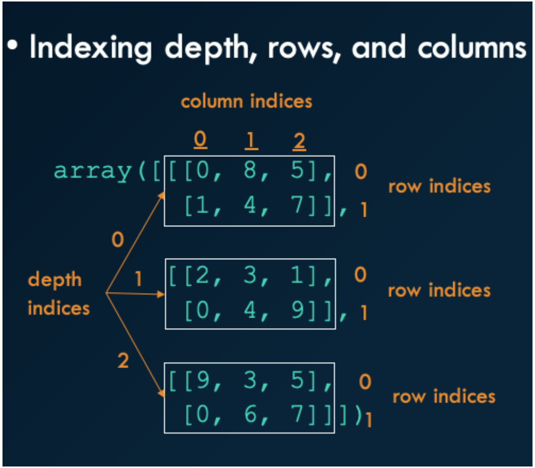
```
>>> np.array([[[0,8,5],[1,4,7]],[[2,3,1],[0,4,9]],[[9,3,5],[0,6,7]]])

array([[[0, 8, 5],
        [1, 4, 7]],

       [[2, 3, 1],
        [0, 4, 9]],

       [[9, 3, 5],
        [0, 6, 7]]])
```

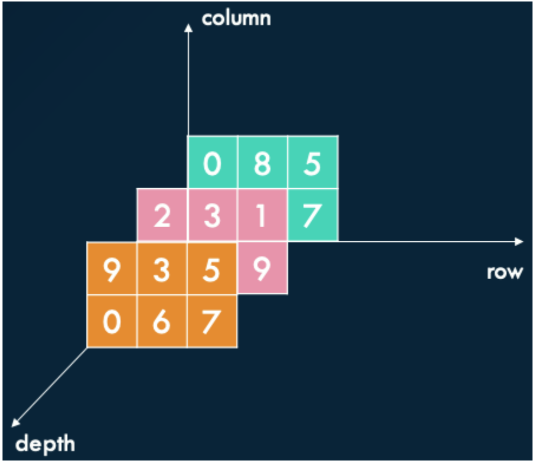
# Visualizing a 3D array

```
array([[[0, 8, 5],
        [1, 4, 7]],

       [[2, 3, 1],
        [0, 4, 9]],

       [[9, 3, 5],
        [0, 6, 7]]])
```



# 3D array indexing



# 3D array selection by indices

```
>>> a
array([[[0, 8, 5],
        [1, 4, 7]],

       [[2, 3, 1],
        [0, 4, 9]],

       [[9, 3, 5],
        [0, 6, 7]]])
```

```
>>> a[1, 1, 2]
9
```

- a[depth, row, column]

# 3D array selection by slice

```
>>> a
array([[[0, 8, 5],
        [1, 4, 7]],

       [[2, 3, 1],
        [0, 4, 9]],

       [[9, 3, 5],
        [0, 6, 7]]])
```

- a[depth 1+, all rows, columns < 2]

```
>>> a[1: ,: , :2]
array([[[2, 3],
        [0, 4]],

       [[9, 3],
        [0, 6]]])
```

# 3D array selection by list of indices

```
>>> a
array([[[0, 8, 5],
        [1, 4, 7]],

       [[2, 3, 1],
        [0, 4, 9]],

       [[9, 3, 5],
        [0, 6, 7]]])
```

```
>>> a[[0,2,1],[1,0,1],[2,0,1]]
array([7, 9, 4])
```

- This example selects elements

  (0,1,2) Depth 0, Row: 1, Column: 2

  (2,0,0) Depth 2, Row: 0, Column: 0

  (1,1,1) Depth 1, Row: 1, Column: 1

# More 3D array selection

```
>>> a
array([[[0, 8, 5],
        [1, 4, 7]],

       [[2, 3, 1],
        [0, 4, 9]],

       [[9, 3, 5],
        [0, 6, 7]]])
```

```
>>> a[[0,2],:,:]
array([[[0, 8, 5],
        [1, 4, 7]],

       [[9, 3, 5],
        [0, 6, 7]]])
```

- depths 0 and 2
- all rows and columns

```
>>> a[[0,2],:,[0,2]]
array([[0, 1],
       [5, 7]])
```

- depths 0 and 2
- all rows
- column 0 from depth 0
- column 2 from depth 2

```
>>> a[[0,2],1,[0,2]]
array([1, 7])
```

- depth 0 and depth 2
- row 1
- column 0 from depth 0
- column 2 from depth 2

- Get a tuple of array dimensions
- 1D

```
>>> arr1 = np.array([2, 3, 4, 5, 6])
>>> arr1.shape
(5,)
```

- 2D

```
>>> arr2 = np.array([[0, 8, 5], [1, 4, 7]])
>>> arr2.shape
(2,3)
```

# shape 3D

- 3D

```
>>> arr3
array([[[0, 8, 5, 3],
        [1, 4, 7, 2]],

       [[2, 3, 1, 0],
        [0, 4, 9, 9]],

       [[9, 3, 5, 2],
        [0, 6, 7, 5]]])

>>> arr3.shape
(3, 2, 4)
```

# Creating and filling multi-dimensional arrays

- To create a 2 by 5 array, filled with 3.14
- `np.full(shape, fill_value)`
- Example shape: (<axis 0, rows>, <axis 1, columns>)

```
>>> np.full((2, 5), 3.14)

array([[ 3.14, 3.14, 3.14, 3.14, 3.14],
       [ 3.14, 3.14, 3.14, 3.14, 3.14]])
```

# Creating and filling multi-dimensional arrays

- To create a 3 by 2 by 5 array, filled with 0
- `np.zeros(shape)`
- Example shape: (<axis 0, depth>, <axis 1, rows>, <axis 2, columns>)

```
>>> np.zeros((3, 2, 5))
array([[[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]],

       [[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]],

       [[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]]])
```

# Aggregations over arrays

- Can aggregate along dimensions of high-d arrays
- Syntax

  `<numpy_array>.<aggregate_operation>(axis)`

  - Where `aggregate_operation` is one of
    - `sum`
    - `max`
    - `min`
    - `argmax`
    - `argmin`
    - `...`
  - `axis` specifies the axis along which the aggregation is performed

# Aggregations over a 1D array

- Convert a 1D array into a scalar
- Compute the sum

```
>>> a
array([1, 2, 3, 4, 5])
>>> a.sum()
15
```

- Find the max value

```
>>> a
array([1, 2, 3, 4, 5])
>>> a.max()
5
```

- Find the position of the max value

```
>>> a
array([1, 2, 3, 4, 5])
>>> a.argmax()
4
```

# Sum over a 2D array

- Convert a 2D array into 1D

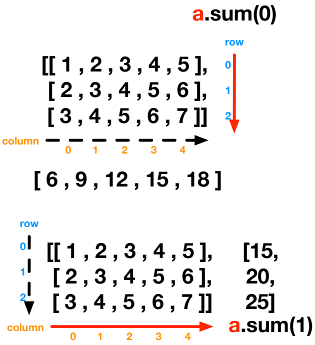
```
>>> a
array([[1, 2, 3, 4, 5],
       [2, 3, 4, 5, 6],
       [3, 4, 5, 6, 7]])
```

- Collapse the array along the rows (axis = 0) and sum the values (reduces the row dimension)

```
>>> a.sum(0)
array([6, 9, 12, 15, 18])
```

- Collapse the array along the columns (axis = 1) and sum the values (reduces the column dimension)

```
>>> a.sum(1)
array([15, 20, 25])
```



# Max over a 2D array

- Convert a 2D array into 1D

```
>>> a
array([[10, 2,  3, 4, 5],
       [ 2, 3, 13, 5, 6],
       [ 3, 4,  5, 6, 7]])
```

- Get the max value

```
>>> a.max()
13
```

- Collapse the array along the rows (axis = 0) and get the max values (reduces the row dimension)

```
>>> a.max (0)
array([10, 4, 13, 6, 7])
```

- Collapse the array along the columns (axis = 1) and get the max values (reduces the column dimension)

```
>>> a.max (1)
array([10, 13, 7])
```

# Max position over a 2D array

- Convert a 2D array into 1D
- Find the position of the max value(s)

```
>>> a

array([[10, 2,  3, 4, 5],
       [ 2, 3, 13, 5, 6],
       [ 3, 4,  5, 6, 7]])
```

- Get the position of the max value

```
>>> a.argmax()
7
```

# Max position over a 2D array by dimension

- Convert a 2D array into 1D
- Find the position of the max value(s)

```
>>> a

array([[10, 2,  3, 4, 5],
       [ 2, 3, 13, 5, 6],
       [ 3, 4,  5, 6, 7]])
```

- Collapse the array along the rows (axis = 0) and get the positions of the max values

```
>>> a.argmax(0)
array([0, 2, 1, 2, 2])
```

- Collapse the array along the columns (axis = 1) and get the positions of the max values

```
>>> a.argmax(1)
array([0, 2, 4])
```

# Sum over a 3D array

- Converts a 3D array into 2D
- Collapse the array along the depth (axis = 0) and sum the values (reduces the depth dimension)

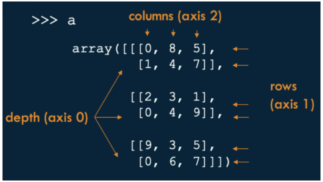
```
>>> a.sum(0)
array([[11, 14, 11],
       [ 1, 14, 23]])
```

- Collapse the array along the rows (axis = 1) and sum the values (reduces the row dimension)

```
>>> a.sum(1)
array([[ 1, 12, 12],
       [ 2,  7, 10],
       [ 9,  9, 12]])
```

- Collapse the array along the columns (axis = 2) and sum the values (reduces the column dimension)

```
>>> a.sum(2)
array([[13, 12],
       [ 6, 13],
       [17, 13]])
```



# Max over a 3D array

- Converts a 3D array into 2D
- Collapse the array along the depth (axis = 0) and get the max values (reduces the depth dimension)

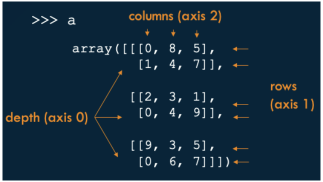
```
>>> a.max(0)
array([[9, 8, 5],
       [1, 6, 9]])
```

- Collapse the array along the rows (axis = 1) and get the max values (reduces the row dimension)

```
>>> a.max(1)
array([[1, 8, 7],
       [2, 4, 9],
       [9, 6, 7]])
```

- Collapse the array along the columns (axis = 2) and get the max values (reduces the column dimension)

```
>>> a.max(2)
array([[8, 7],
       [3, 9],
       [9, 7]])
```



# Max position over a 3D array

- Converts a 3D array into 2D
- Collapse the array along the depth (axis = 0) and get the positions of the max values (reduces the depth dimension)

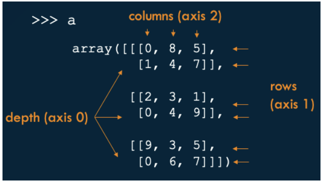
```
>>> a.argmax(0)
array([[2, 0, 0],
       [0, 2, 1]])
```

- Collapse the array along the rows (axis = 1) and get the positions of the max values (reduces the row dimension)

```
>>> a.argmax(1)
array([[1, 0, 1],
       [0, 1, 1],
       [0, 1, 1]])
```

- Collapse the array along the columns (axis = 2) and get the positions of the max values (reduces the column dimension)

```
>>> a.argmax(2)
array([[1, 2],
       [1, 2],
       [0, 2]])
```



# More examples of creating arrays

- Create an array with odd numbers thru 10

```
>>> np.arange(1, 11, 2)
array([1, 3, 5, 7, 9])
```

- “Tile” an array

```
>>> np.tile (np.arange(1, 11, 2), (1, 2))
array([[1, 3, 5, 7, 9, 1, 3, 5, 7, 9]])
>>> np.tile (np.arange(1, 11, 2), (2, 1))
array([[1, 3, 5, 7, 9],
       [1, 3, 5, 7, 9]])
```

# NumPy single array vectorized operations

- Return a single number
  - `np.min()` / `np.max()`
  - `np.argmin()` / `np.argmax()`
  - `np.mean()`
  - `np.median()`
  - `np.std()` / `np.var()`
  - `np.sum()`
  - `np.cumsum()` - return the cumulative sum of elements along the specified axis
  - `np.cumprod()` - return the cumulative product of elements along the specified axis

# NumPy single element vectorized operations

- Return the same-sized array
  - `np.abs()`
  - `np.sqrt() / np.square()`
  - `np.exp()`
  - `np.log() / np.log10 / np.log2()`
  - `np.sign()`
  - `np.ceil() / np.floor() / np.rint()` (round to nearest integer)
  - `np.isnan()`

# NumPy pair of same shape arrays vectorized operations

- Arrays must have the same shape
- Element-wise operations
- Return the same-sized array
  - `np.add() / (+)`
  - `np.subtract() / (-)`
  - `np.multiply() / (*)`
  - `np.divide() / np.floor_divide / (/)`
  - `np.power()`
  - `np.mod()`
  - `np.greater() / np.greater_equal()`
  - `np.less() / np.less_equal()`
  - `np.equal() / np.not_equal()`
  - `np.logical_and() / np.logical_or() / np.logical_xor()`

# Customized vectorized operations

- `np.vectorize()`
- Evaluate the specified function on the input array
- E.g., put 0 for even numbers in the array and 1 for odd numbers in the array

```
def even_odd (x):
    if x % 2 == 0:
        return 0
    else:
        return 1
>>> a
array([[1, 2],
       [3, 4]])
>>> vec_even_odd = np.vectorize(even_odd)
>>> vec_even_odd(a_array)

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

# Other useful vectorized operations: `where`

- `np.where(condition, positive, negative)`
- Returns elements from positive or negative, depending on the outcome of condition
- E.g., if an element of the array is less than a value, return that value, otherwise, return the negation of the value

```
>>> a = np.arange(10)
>>> np.where(a < 5, a, a * -1)
array([ 0,  1,  2,  3,  4, -5, -6, -7, -8, -9])
```

# Other useful vectorized operations: `nonzero`

- `np.nonzero(a)`
- Returns the indexes of the nonzero elements

```
a = np.array([0, 1, 4, 0, 5, 7, 0])

a.nonzero()
array([1, 2, 4, 5])
```

# Wrap up

1 Introduction to numpy

? How can we use what we learned today?

? What do we know now that we didn't know before?