CSE 4510/5310
BIG DATA

Instructor: Fitzroy Nembhard, Ph.D.

Week 2

Numpy:
Numerical Python

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY®

Distribution

- All slides included in this class are for the exclusive use of students and instructors associated with Programming in a Second Language (CSE 2050) at the Florida Institute of Technology
- Redistribution of the slides is not permitted without the written consent of the author.

Goals

- To discuss and practice the basics of numerical tools (NumPy)
- To practice data processing/analysis concepts using real-world datasets

2.1 NUMPY: ARRAYS AND VECTORIZED COMPUTATION

NumPy and Pandas

- **NumPy**, short for Numerical Python, is one of the most important foundational packages for numerical computing in Python.
- Many functions that we use in Pandas build upon the NumPy array and its set of functions.
- Wes McKinney is the creator of the Python pandas project. He is also the author of a textbook named Python for Data Analysis textbook.

NumPy Components

- **ndarray** - an efficient multidimensional array providing fast array-oriented arithmetic operations and flexible broadcasting capabilities.
  - It is used for processing homogeneous data; that is, all of the elements must be the same type.
  - Every array has a ***shape***, a tuple indicating the size of each dimension, and a ***dtype***, an object describing the *data type* of the array
- **Mathematical functions** for fast operations on entire arrays of data without having to write loops.
- **Tools for reading/writing array data** to disk and working with memory-mapped files.
- Linear algebra, **random number generation**, and Fourier transform capabilities.
- A C API for connecting NumPy with libraries written in C, C++, or FORTRAN.

Major Features of NumPy

- Fast vectorized array operations for data munging and cleaning, subsetting and filtering, transformation, and any other kinds of computations
- Common array algorithms like sorting, unique, and set operations
- Efficient descriptive statistics and aggregating/summarizing data
- Data alignment and relational data manipulations for merging and joining heterogeneous datasets
- Expressing conditional logic as array expressions instead of loops with if-elif-else branches
- Group-wise data manipulations (aggregation, transformation, function application)

Batch computations with NumPy ndarray

- Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements

```
import numpy as np
data = np.random.randn(2, 3) # Creates random normally distributed data


# Output is a 2x3 array filled with random data

>>> array([[-0.2047, 0.4789, -0.5194],
           [-0.5557, 1.9658, 1.3934]])


>>> data * 10
# Output is the result of multiplying each element by 10

array([[-2.0471, 4.7894, -5.1944],
       [-5.5573, 19.6578, 13.9341]])
```

Batch computations with NumPy ndarray (Cont'd)

- Any arithmetic operation between equal-size arrays applies the operation element-wise

```
import numpy as np
data = np.random.randn(2, 3)

# Output is a 2x3 array filled with random data

>>> array([[-0.2047, 0.4789, -0.5194],
           [-0.5557, 1.9658, 1.3934]])


>>> data + data
# Output is the result of positional addition

array([[-0.4094, 0.9579, -1.0389],
       [-1.1115, 3.9316, 2.7868]])
```

Creating an array

- The array function accepts any sequence-like object (including other arrays) and produces a new NumPy array containing the passed data

```
my_list = [6, 7.5, 8, 0, 1] # A regular python list

>>> arr1 = np.array(my_list)

# Results in
array([6. , 7.5, 8. , 0. , 1. ]) # Notice all numbers are floats


my_list2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
arr2 = np.array(data2)

# Results in
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

Creating an array (Cont'd)

- **zeros** and **ones** create **arrays of 0s or 1s**, respectively, with a given length or shape.

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

# Results in
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])


>>> np.zeros((3, 6))

# Results in
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])
```

Creating an array (Cont'd)

- **empty** creates an array without initializing its values to any particular value

```
>>> np.empty((2, 3, 2)) # numpy.empty(shape, dtype=float, order='C', *, like=None)

# Results in the following:
# Do not assume that np.empty will return an array of all zeros.
# It creates an array of uninitialized (arbitrary) data of the given shape, dtype, and order

array([[[0., 0.],
        [0., 0.],
        [0., 0.]],

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

Creating an array (Cont'd)

- **arange** is an array-valued version of the built-in Python range function

```
>>> np.arange(15)


# Results in:
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

Creating an array with dtypes

- Since NumPy is focused on numerical computing, the data type, if not specified, will in many cases be float64 (floating point).
- We can specify the data-type to use when creating an array

```
>>> arr1 = np.array([1, 2, 3], dtype=np.float64)

>>> arr2 = np.array([1, 2, 3], dtype=np.int32)
```

Casting an array to another dtype

- You can explicitly convert or cast an array from one dtype to another using ndarray's astype method

```
>>> arr = np.array([1, 2, 3, 4, 5])

>>> float_arr = arr.astype(np.float64)
```

```
>>> arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
>>> arr.astype(np.int32)

# Results in:
 array([ 3, -1, -2, 0, 12, 10], dtype=int32)
```

Casting an array to another dtype

- If you have an array of strings representing numbers, you can use astype to convert them to numeric form:
- numpy.string_ type may truncate input without warning since it creates fixed size strings

```
>>> my_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)
>>> my_strings.astype(float)


# Results in:
 array([ 1.25, -9.6 , 42. ])
```

Summary of Functions for Creating Arrays

Function	Description
`array`	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype; copies the input data by default
`asarray`	Convert input to ndarray, but do not copy if the input is already an ndarray
`arange`	Like the built-in `range` but returns an ndarray instead of a list
`ones`, `ones_like`	Produce an array of all 1s with the given shape and dtype; `ones_like` takes another array and produces a ones array of the same shape and dtype
`zeros`, `zeros_like`	Like `ones` and `ones_like` but producing arrays of 0s instead
`empty`, `empty_like`	Create new arrays by allocating new memory, but do not populate with any values like `ones` and `zeros`
`full`, `full_like`	Produce an array of the given shape and dtype with all values set to the indicated "fill value" `full_like` takes another array and produces a filled array of the same shape and dtype
`eye, identity`	Create a square N × N identity matrix (1s on the diagonal and 0s elsewhere)

Summary of dtypes for Creating Arrays

Type	Type code	Description
`int8, uint8`	`i1, u1`	Signed and unsigned 8-bit (1 byte) integer types
`int16, uint16`	`i2, u2`	Signed and unsigned 16-bit integer types
`int32, uint32`	`i4, u4`	Signed and unsigned 32-bit integer types
`int64, uint64`	`i8, u8`	Signed and unsigned 64-bit integer types
`float16`	`f2`	Half-precision floating point
`float32`	`f4 or f`	Standard single-precision floating point; compatible with C float
`float64`	`f8 or d`	Standard double-precision floating point; compatible with C double and Python `float` object
`float128`	`f16 or g`	Extended-precision floating point
`complex64, complex128, complex256`	`c8, c16, c32`	Complex numbers represented by two 32, 64, or 128 floats, respectively
`bool`	?	Boolean type storing `True` and `False` values
`object`	O	Python object type; a value can be any Python object
`string_`	S	Fixed-length ASCII string type (1 byte per character); for example, to create a string dtype with length 10, use `'S10'`
`unicode_`	U	Fixed-length Unicode type (number of bytes platform specific); same specification semantics as `string_` (e.g., `'U10'`)

Character	Description	Example
`'b'`	Byte	`np.dtype('b')`
`'i'`	Signed integer	`np.dtype('i4') == np.int32`
`'u'`	Unsigned integer	`np.dtype('u1') == np.uint8`
`'f'`	Floating point	`np.dtype('f8') == np.int64`
`'c'`	Complex floating point	`np.dtype('c16') == np.complex128`
`'S','a'`	string	`np.dtype('S5')`
`'U'`	Unicode string	`np.dtype('U') == np.str_`
`'V'`	Raw data (void)	`np.dtype('V') == np.void`

Array Indexing and Slicing

- There are many ways you may want to select a subset of your data or individual elements

```
>>> arr = np.arange(10)
>>> array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> arr[5]

# Results in:
5
```

```
>>> arr[5:8]

# Results in:
 array([5, 6, 7])
```

Array Indexing and Slicing (Cont'd)

- There are many ways you may want to select a subset of your data or individual elements

```
>>> arr = np.arange(10)
>>> array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> arr[5:8] = 12

# Results in:
array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])


arr_slice = arr[5:8]
arr_slice[1] = 12345

# Results in the following:
# The effect propagated or broadcasted to the original array
array([ 0, 1, 2, 3, 4, 12, 12345, 12, 8, 9])
```

Array Indexing and Slicing (Cont'd)

- There are many ways you may want to select a subset of your data or individual elements

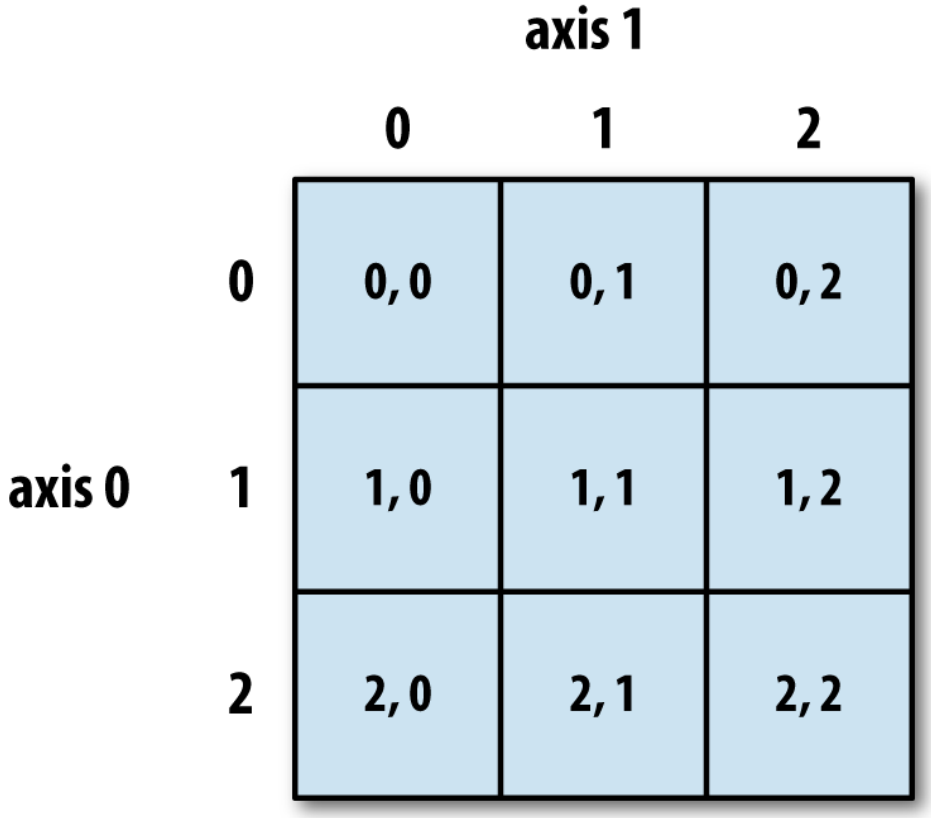
```
>>> arr = np.arange(10)
>>> array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> arr_slice = arr[5:8]
arr_slice[:] = 64

# Results in the following:
# The effect propagated or broadcasted to the original array
array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])


# To keep the data from broadcasting, use the copy function to create a new array
arr[5:8].copy()
```

Array Indexing and Slicing (Cont'd)

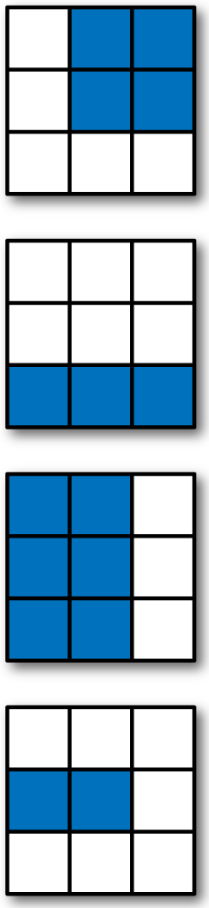
- In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays:

```
>>> arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> arr2d[2]

# Results in the following:
array([7, 8, 9])


>>> arr2d[0][2]          # or arr2d[0, 2]
# Results in the following:
3
```

Array Indexing and Slicing (Cont'd)



Array Indexing and Slicing (Cont'd)

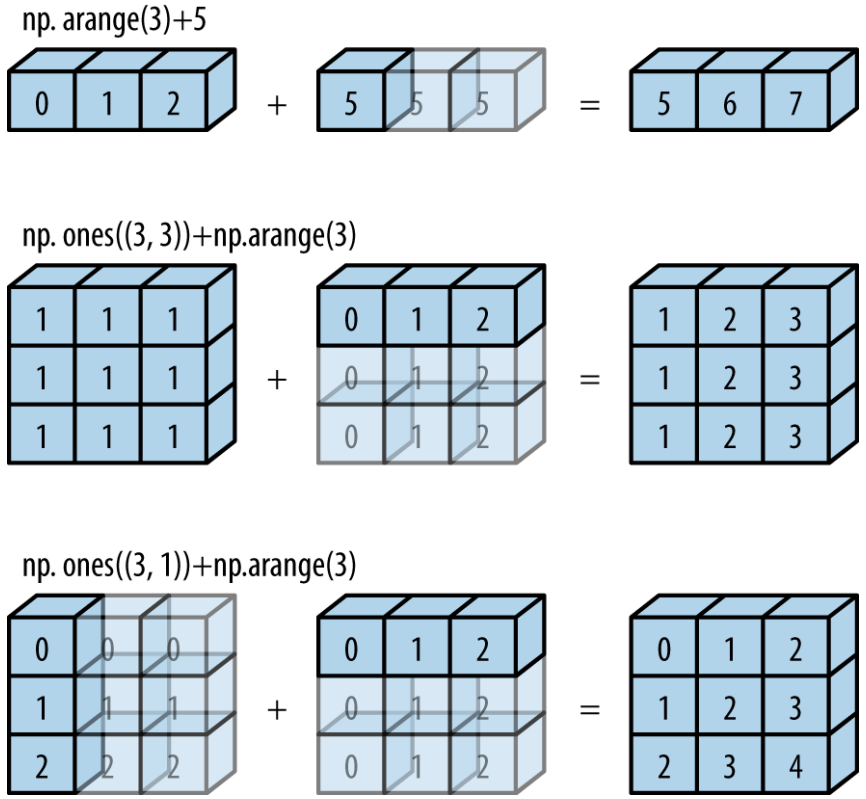


Expression	Shape
`arr[:2, 1:]`	`(2, 2)`
`arr[2]`	`(3,)`
`arr[2, :]`	`(3,)`
`arr[2:, :]`	`(1, 3)`
`arr[:, :2]`	`(3, 2)`
`arr[1, :2]`	`(2,)`
`arr[1:2, :2]`	`(1, 2)`

Broadcasting Explained

- Another means of vectorizing operations is to use NumPy's broadcasting functionality.
- Broadcasting is simply a set of rules for applying binary ufuncs (addition, subtraction, multiplication, etc.) on arrays of different sizes.

```
>>> arr = np.arange(5)
array([0, 1, 2, 3, 4])

>>> arr * 4

# Results in the following:
array([ 0, 4, 8, 12, 16])
```



Boolean Indexing

- Like arithmetic operations, comparisons (such as ==) with arrays are also vectorized.
- Thus, comparing names with the string 'Bob' yields a Boolean array

```
>>> names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
>>> data = np.random.randn(7, 4) # Creates random normally distributed data

# Results in the following:
array([[ 0.0929, 0.2817, 0.769 , 1.2464],
       [ 1.0072, -1.2962, 0.275 , 0.2289],
       [ 1.3529, 0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397, 0.477 ],
       [ 3.2489, -1.0212, -0.5771, 0.1241],
       [ 0.3026, 0.5238, 0.0009, 1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])

>>> names == 'Bob'
# Results in the following:
array([ True, False, False,  True, False, False, False])
```

Boolean Indexing

- A boolean array can be passed when indexing an array
- The boolean array must be of the same length as the array axis it's indexing

```
# Given
array([[ 0.0929, 0.2817, 0.769 , 1.2464],
       [ 1.0072, -1.2962, 0.275 , 0.2289],
       [ 1.3529, 0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397, 0.477 ],
       [ 3.2489, -1.0212, -0.5771, 0.1241],
       [ 0.3026, 0.5238, 0.0009, 1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])

>>> array([ True, False, False,  True, False, False, False])
>>> data[names == 'Bob']

# Results in the following:
array([[ 0.0929, 0.2817, 0.769 , 1.2464],
       [ 1.669 , -0.4386, -0.5397, 0.477 ]])
```

Fancy Indexing

- Fancy indexing is a term adopted by NumPy to describe indexing using integer arrays

```
>>> arr = np.empty((8, 4))
>>> for i in range(8):
        arr[i] = i # updates an entire row with data
>>> arr


# Results in the following:
array([[0., 0., 0., 0.],
       [1., 1., 1., 1.],
       [2., 2., 2., 2.],
       [3., 3., 3., 3.],
       [4., 4., 4., 4.],
       [5., 5., 5., 5.],
       [6., 6., 6., 6.],
       [7., 7., 7., 7.]])
```

Reshaping an Array

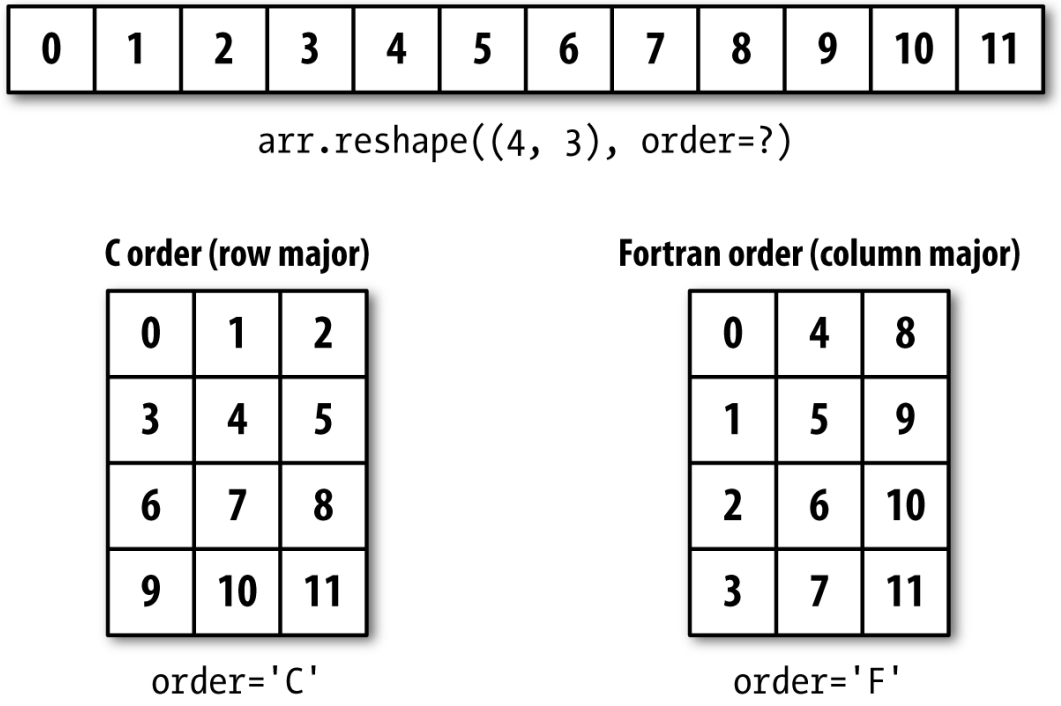
- Fancy indexing is a term adopted by NumPy to describe indexing using integer arrays

```
>>> arr = np.arange(32).reshape((8, 4))
>>> for i in range(8):      # or for i in range(arr.shape[0])
        arr[i] = i
>>> arr

# Results in the following:
array([[0,  1,  2,  3],
       [4,  5,  6,  7],
       [8,  9, 10,  11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])
```

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

Reshaping an Array (Cont'd)

- C/row major order: Traverse higher dimensions first (e.g., axis 1 before advancing on axis 0).
- Fortran/column major order: Traverse higher dimensions last (e.g., axis 0 before advancing on axis 1) . In the FORTRAN 77 language, matrices are all column major



Summing the Values in an Array

- Because it executes the operation in compiled code, NumPy's version of the sum operation is computed much more quickly

```
import numpy as np
>>> L = np.random.random(100)
>>> sum(L)

>>> big_array = np.random.rand(1000000)
%timeit sum(big_array) # Python's built-in sum
%timeit np.sum(big_array) # numpy's sum

10 loops, best of 3: 104 ms per loop
1000 loops, best of 3: 442 µs per loop
```

IPython's special commands (which are not built into Python itself) are known as "magic" commands.

These are designed to facilitate common tasks and enable you to easily control the behavior of the IPython system.

A magic command is any command prefixed by the percent symbol %

Examples:

%pwd # prints the working directory. You could store this in a variable like so my_var = %pwd

Other available shell-like magic functions are
%cat, %cp, %env, %ls, %man, %mkdir, %more, %mv, %pwd, %rm, and %rmdir

Minimum and Maximum

- NumPy's corresponding functions have similar syntax, and again operate much more quickly

```
%timeit min(big_array) # Python's built-in min
%timeit np.min(big_array) # numpy's min

10 loops, best of 3: 82.3 ms per loop
1000 loops, best of 3: 497 µs per loop
```

Frequently used IPython magic commands

Command	Description
`%quickref`	Display the IPython Quick Reference Card
`%magic`	Display detailed documentation for all of the available magic commands
`%debug`	Enter the interactive debugger at the bottom of the last exception traceback
`%hist`	Print command input (and optionally output) history
`%pdb`	Automatically enter debugger after any exception
`%paste`	Execute preformatted Python code from clipboard
`%cpaste`	Open a special prompt for manually pasting Python code to be executed
`%reset`	Delete all variables/names defined in interactive namespace
`%page` *`OBJECT`*	Pretty-print the object and display it through a pager
`%run` *`script.py`*	Run a Python script inside IPython
`%prun` *`statement`*	Execute *`statement`* with `cProfile` and report the profiler output
`%time` *`statement`*	Report the execution time of a single statement
`%timeit` *`statement`*	Run a statement multiple times to compute an ensemble average execution time; useful for timing code with very short execution time
`%who, %who_ls, %whos`	Display variables defined in interactive namespace, with varying levels of information/ verbosity
`%xdel` *`variable`*	Delete a variable and attempt to clear any references to the object in the IPython internals

Minimum and Maximum (Cont'd)

- By default, each NumPy aggregation function will return the aggregate over the entire array
- Aggregation functions take an additional argument specifying the axis along which the aggregate is computed

```
>>> M = np.random.random((3, 4))

# Results in the following:
array([[ 0.8967576 0.03783739 0.75952519 0.06682827]
       [ 0.8354065 0.99196818 0.19544769 0.43447084]
       [ 0.66859307 0.15038721 0.37911423 0.6687194 ]])

>>>  M.min(axis=0) # finds the minimum value within each column
array([ 0.66859307, 0.03783739, 0.19544769, 0.06682827])
```

Fast Sorting in NumPy: np.sort and np.argsort

- **np.sort** function is much more efficient than Python's built-in sort function especially on multi-dimensional arrays

```
>>> x = np.array([2, 1, 4, 3, 5])
>>> np.sort(x)

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

>>>  x = np.array([2, 1, 4, 3, 5]) # returns the indices of the sorted elements
>>>  i = np.argsort(x)

# Results in the following:
[1 0 3 2 4]
```

Fast Sorting in NumPy: lexsort

- **lexsort** is similar to argsort, but it performs an indirect lexicographical sort on multiple key arrays
- The order in which the keys are used to order the data starts with the last array passed

```
>>> surnames =    ('Hertz',    'Galilei', 'Hertz')
>>> first_names = ('Heinrich', 'Galileo', 'Gustav')
>>> ind = np.lexsort((first_names, surnames))

>>> ind
array([1, 2, 0])

>>> [surnames[i] + ", " + first_names[i] for i in ind]
['Galilei, Galileo', 'Hertz, Gustav', 'Hertz, Heinrich']
```

Fast Sorting in NumPy: lexsort

- **lexsort** Example 2

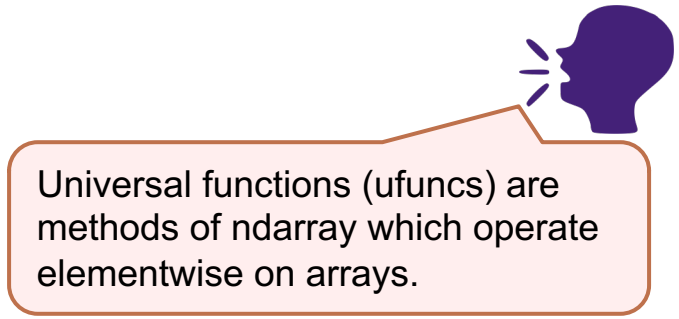
```
>>> first_name = np.array(['Bob', 'Jane', 'Steve', 'Bill', 'Barbara'])
>>> last_name = np.array(['Jones', 'Arnold', 'Arnold', 'Jones', 'Walters'])
>>> sorter = np.lexsort((first_name, last_name))
>>> list(zip(last_name[sorter], first_name[sorter]))

# Results in the following:
[('Arnold', 'Jane'),
 ('Arnold', 'Steve'),
 ('Jones', 'Bill'),
 ('Jones', 'Bob'),
 ('Walters', 'Barbara')]
```

Summary of operations in NumPy

- The key to making NumPy fast is to use vectorized operations, generally implemented through NumPy's universal functions (ufuncs)

Operator	Equivalent ufunc	Description
`+`	`np.add`	Addition (e.g., `1 + 1 = 2`)
`-`	`np.subtract`	Subtraction (e.g., `3 - 2 = 1`)
`-`	`np.negative`	Unary negation (e.g., `-2`)
`*`	`np.multiply`	Multiplication (e.g., `2 * 3 = 6`)
`/`	`np.divide`	Division (e.g., `3 / 2 = 1.5`)
`//`	`np.floor_divide`	Floor division (e.g., `3 // 2 = 1`)
`**`	`np.power`	Exponentiation (e.g., `2 ** 3 = 8`)
`%`	`np.mod`	Modulus/remainder (e.g., `9 % 4 = 1`)

Operator	Equivalent ufunc
`&`	`np.bitwise_and`
`|`	`np.bitwise_or`
`^`	`np.bitwise_xor`
`~`	`np.bitwise_not`



Summary of ufuncs in NumPy

- The key to making NumPy fast is to use vectorized operations, generally implemented through NumPy's universal functions (ufuncs)

Operator	Equivalent ufunc
`==`	`np.equal`
`!=`	`np.not_equal`
`<`	`np.less`
`<=`	`np.less_equal`
`>`	`np.greater`
`>=`	`np.greater_equal`

Aggregation functions available in NumPy

Function Name	NaN-safe Version	Description
`np.sum`	`np.nansum`	Compute sum of elements
`np.prod`	`np.nanprod`	Compute product of elements
`np.mean`	`np.nanmean`	Compute median of elements
`np.std`	`np.nanstd`	Compute standard deviation
`np.var`	`np.nanvar`	Compute variance
`np.min`	`np.nanmin`	Find minimum value
`np.max`	`np.nanmax`	Find maximum value
`np.argmin`	`np.nanargmin`	Find index of minimum value
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements
`np.any`	N/A	Evaluate whether any elements are true
`np.all`	N/A	Evaluate whether all elements are true

2.2 APPLICATIONS OF NUMPY

1. Application of Boolean Indexing

- How can we use Boolean indexing to extract a county's records based on its name?

```
# Gold, Silver, Bronze
medals = np.array([          ..... Canada
   [0, 3, 0],    ..... Italy
   [0, 0, 1],    ..... Germany
   [0, 0, 1],    ..... Japan
   [1, 0, 0],    ..... Kazakhstan
   [0, 0, 1],    ..... Russia
   [3, 1, 1],    ..... South Korea
   [0, 1, 0],    ..... United States
   [1, 0, 1]
])
```

	Gold	Silver	Bronze
Canada	0	3	0
Italy	0	0	1
Germany	0	0	1
Japan	1	0	0
Kazakhstan	0	0	1
Russia	3	1	1
South Korea	0	1	0
United States	1	0	1

2. What Is the Average Height of US Presidents?

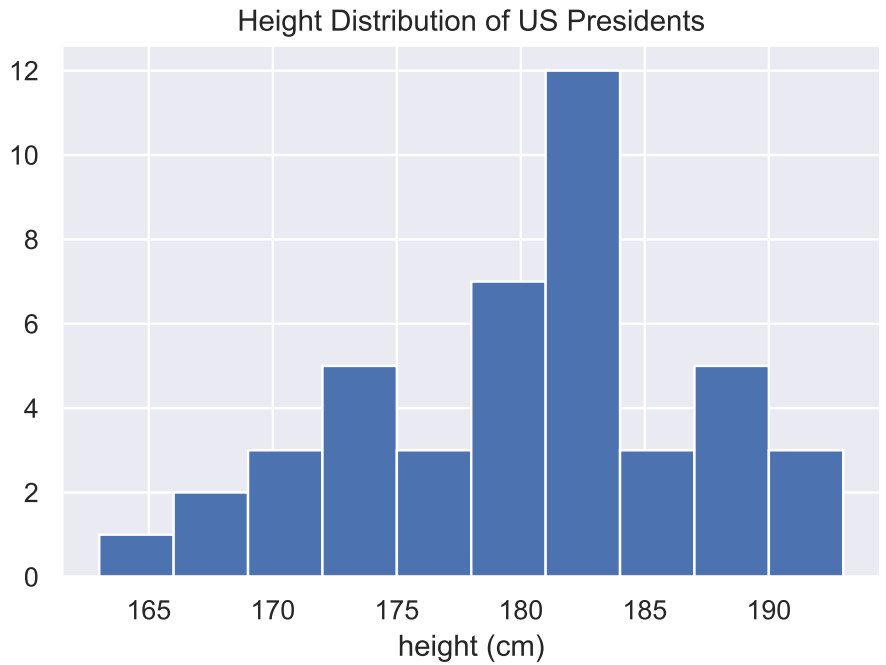
- Locate the file `presheights.csv` on Canvas
- Here, we use pandas to read the CSV file

```
>>> import pandas as pd
data = pd.read_csv('data/president_heights.csv')
heights = np.array(data['height(cm)']) # select the height column

print("Mean height:", heights.mean())
print("Standard deviation:", heights.std())
print("Minimum height:", heights.min())
print("Maximum height:", heights.max())
print("25th percentile:", np.percentile(heights, 25))
print("Median:", np.median(heights))
print("75th", np.percentile(heights, 75))
```

2. What Is the Average Height of US Presidents?

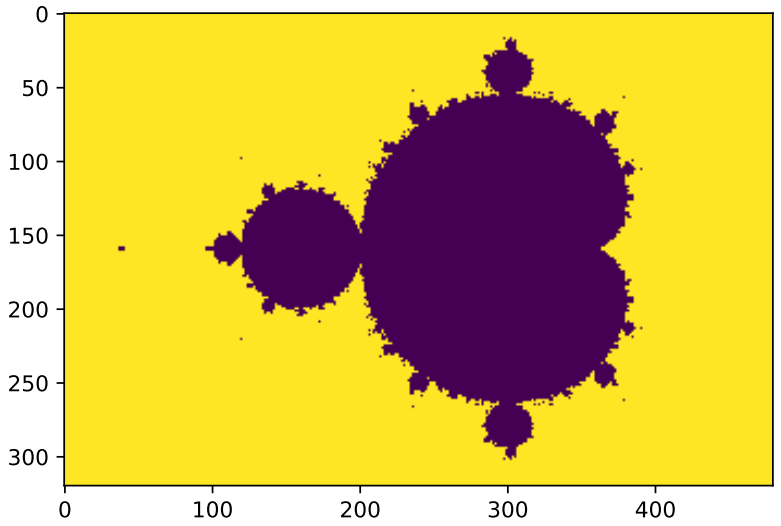
- Plotting a chart of the data

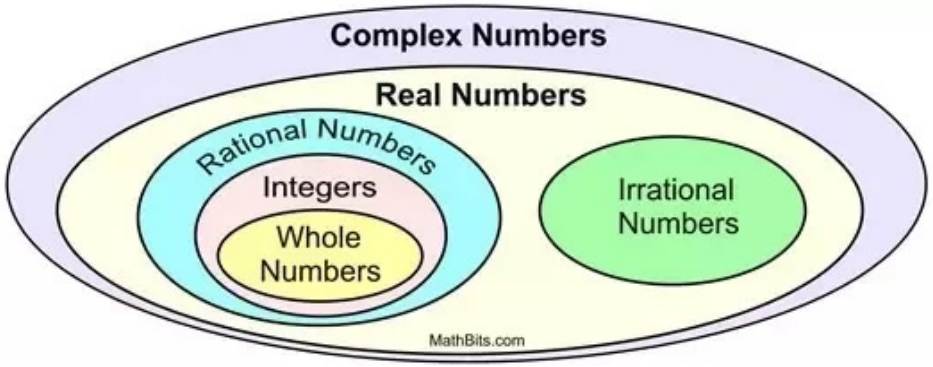
```
%matplotlib inline
import matplotlib.pyplot as plt

plt.hist(heights)
plt.title('Height Distribution of US Presidents')
plt.xlabel('height (cm)')
plt.ylabel('number');
```



3. Advanced* Numpy Example: Mandelbrot

- The Mandelbrot fractal is computed by defining, for each $c \in \mathbb{C}$, a polynomial $fc(z) = z^2 + c$. This polynomial is then *iterated* starting with z=0, that is, $f_c^n(0)$ is computed, and any point c for which $f_c^n(0)$ does *not* "escape to infinity" for $n \rightarrow \infty$ is part of the Mandelbrot set.
- $Z^{(t+1)} = Z^{(t)} * Z^{(t)} + C$





* For demonstration purposes only

Advanced* Numpy Example: Mandelbrot (Cont'd)

$$Mandelbrot\ Fractal: Z^{(t+1)} = Z^{(t)} * Z^{(t)} + C$$

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

def mandelbrot(m, n):
    x = np.linspace(-2, 1, num=m).reshape((1, m))
    y = np.linspace(-1, 1, num=n).reshape((n, 1))
    C = np.tile(x, (n, 1)) + 1j * np.tile(y, (1, m))

    Z = np.zeros((n, m), dtype=complex)
    M = np.full((n, m), True, dtype=bool)
    for i in range(100):
        Z[M] = Z[M] * Z[M] + C[M]
        M[np.abs(Z) > 2] = False # Boolean indexing to remove values that "escape to infinity"

    plt.imshow(np.uint8(np.flipud(1 - M) * 255))
```

How does linspace work?
It creates sequences of evenly spaced values within a defined interval.

```
np.linspace(start = 0, stop = 100, num = 5)
```

How does np.tile work?
`numpy.tile(A, reps)`
Construct an array by repeating A the number of times given by reps

---

How does np.full work?
`numpy.full(shape, fill_value)`
Return a new array of given shape and type, filled with fill_value.

* For demonstration purposes only

Source: https://tomroelandts.com/articles/how-to-compute-the-mandelbrot-set-using-numpy-array-operations