Numerical Python

v0.4

Research Computing Services
IS & T

Python's strengths

- Python is a general purpose language.
  - In contrast with R or Matlab which started out as specialized languages
- Python lends itself to implementing complex or specialized algorithms for solving computational problems.
- It is a highly productive language to work with that's been applied to hundreds of subject areas.

Extending its Capabilities

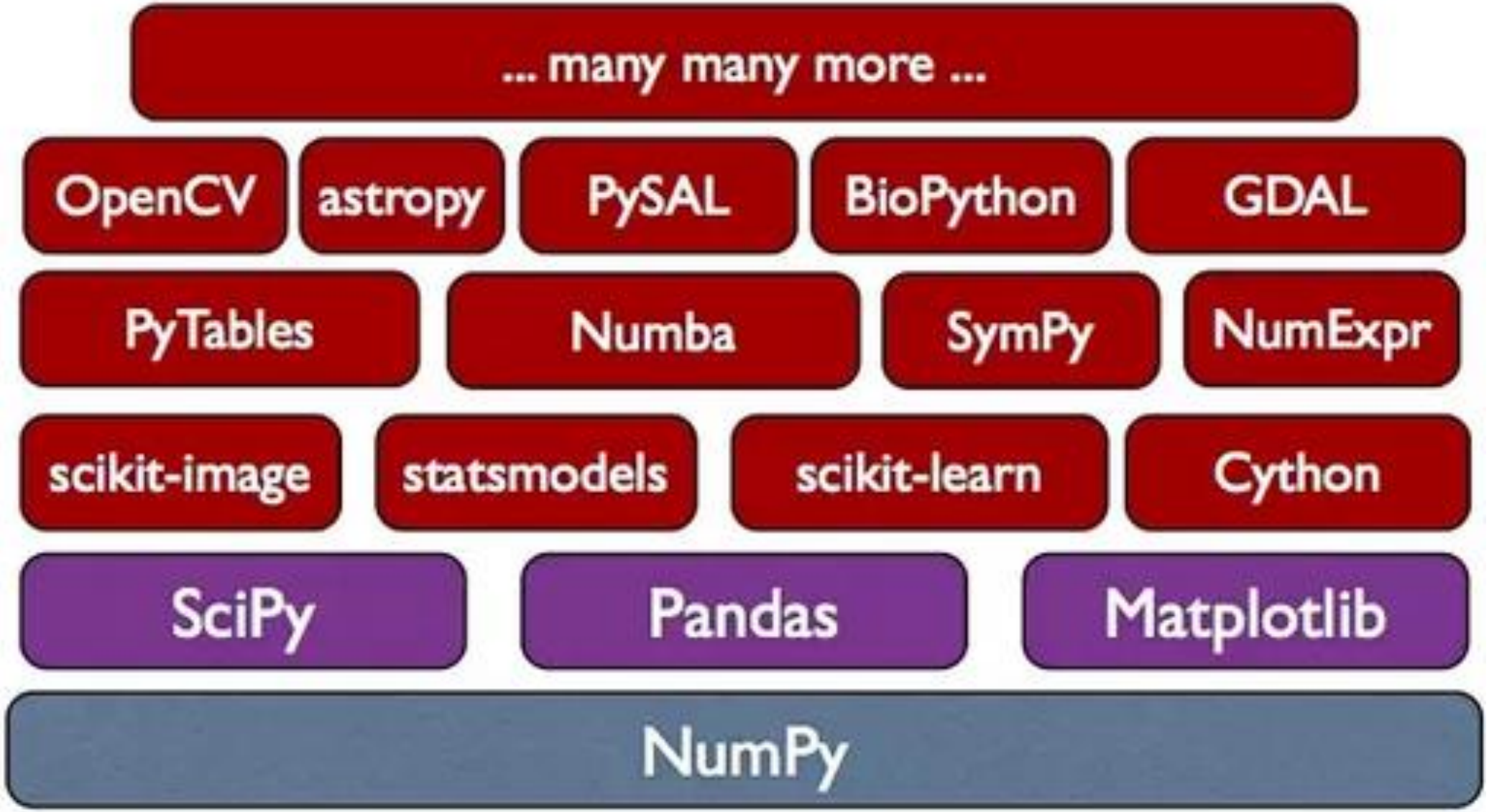
- However…for number crunching some aspects of the language are not optimal:
  - Runtime type checks
  - No compiler to analyze a whole program for optimizations
  - General purpose built-in data structures are not optimal for numeric calculations

- “regular” Python code is not competitive with compiled languages (C, C++, Fortran) for numeric computing.

- The solution: specialized libraries that extend Python with data structures and algorithms for numeric computing.
  - Keep the good stuff, speed up the parts that are slow!

NumPy

- NumPy provides optimized data structures and basic routines for manipulating multidimensional numerical data.
- Mostly implemented in compiled C code.
- NumPy underlies many other numeric and algorithm libraries used throughout the Python software ecosystem.



Source: Continuum Analytics

Ndarray – the basic NumPy data type

- NumPy ndarray's are:
  - Typed
  - Fixed in size
  - Fixed in dimensionality

- An ndarray can be constructed from:
  - Conversion from a Python list, set, tuple, or similar data structure
  - NumPy initialization routines
  - Copies or computations with other ndarray's
  - NumPy-based functions as a return value

ndarray vs list

- List:
  - General purpose
  - Untyped
  - 1 dimension
  - Resizable
    - Add/remove elements anywhere
  - Accessed with [ ] notation and integer indices
- Ndarray:
  - Intended to store and process (mostly) numeric data
  - Typed
  - N-dimensions
    - Chosen at creation time
  - Fixed size
    - Chosen at creation time
  - Accessed with [ ] notation and integer indices

Make some ndarrays

- Let's play around with ndarrays for a bit…

```
import numpy as np

# Make a list
nums = [1,2.0, -1.3]
# Convert it to a numpy ndarray
arr = np.array(nums)
print(f'A numpy array: {arr}')

# Make another list
nums2 = [-2.3, -4, 6]
# make a list of lists - get a
# 2D array
arr2 = np.array([nums,nums2])

# Print a list of lists
print(f'A list of lists:\n
{[nums,nums2]}')
# Print a 2D array
print(f'A numpy 2D array:\n {arr2}')
```

BOSTON UNIVERSITY

ndarray math

- By default operators work element-by-element

- These are executed in compiled C code.

BOSTON UNIVERSITY

```
import numpy as np

a = np.array([1,2,3,4])
b = np.array([4,5,6,7])

c = a / b
# c is an ndarray
print(type(c))
# <class 'numpy.ndarray'>

c = a * b
# c → array([ 4, 10, 18, 28])
a + b
# array([ 5,  7,  9, 11])
a - b
# array([-3, -3, -3, -3])
a / b
# array([0.25, 0.4, 0.5, 0.57142857])

# Broadcasting
-2 * a
# array([ 2,  4,  6, 8])

# Combinations...
-2 * a + b**2
# array([14, 21, 30, 41])
```

- Vectors are applied row-by-row to matrices
- The length of the vector must match the width of the row.

```
a = np.array([10,20,30,40])

c = np.array([[1,2,3,4],
              [4,5,6,7],
              [1,1,1,1],
              [2,2,2,2]])

a + c
# array([[3, 4, 5, 6],
#                [6, 7, 8, 9],
#                [3, 3, 3, 3],
#                [4, 4, 4, 4]])
```

BOSTON UNIVERSITY

ndarray sizes & reshaping

```
a = np.array([10,20,30,40])

c = np.array([[1,2,3,4],
              [4,5,6,7],
              [1,1,1,1],
              [2,2,2,2]])
# The size is a property, not a
# function call.
a.size
# 4
c.size
# 16
```

- How many elements?

```
a.shape
# (4,)
c.shape
# (4,4)
```

- What are the dimensions?

```
d = np.array([1,2,3,4,5,6,7,8,9])
d.shape
# (9,)
d.size
# 9

d_2d = d.reshape((3,3))
# Or: d_2d = np.reshape(d,(3,3))
d_2d.shape
# (3,3)

d_col = d.reshape((1,9))
d_col.shape
# (1,9)

# Transpose it
d_row = d_col.T
d_row.shape
# (9,1)

# Make it 3d
d_3d = d.reshape((1,3,3))
d_3d.size
# 9
d_3d.shape
# (1,3,3)
```

- *reshape()* lets us change the dimensions without changing the size.

BOSTON UNIVERSITY

Vectors + Matrices, again.

- A 1-D vector is always treated as a row.
- If you want to do operations by columns, make the vector a 2D array where one dimension is of size 1.

```
# make this 2D, 4 rows, 1 column
a = np.array([10,20,30,40]).reshape((4,1))

c = np.array([[1,2,3,4],
              [4,5,6,7],
              [1,1,1,1],
              [2,2,2,2]])

a + c
# array([[11, 12, 13, 14],
#        [24, 25, 26, 27],
#        [31, 31, 31, 31],
#        [42, 42, 42, 42]])



# Transpose A, now it'll add row-by-row
a.T + c
#array([[11, 22, 33, 44],
#       [14, 25, 36, 47],
#       [11, 21, 31, 41],
#       [12, 22, 32, 42]])
```

BOSTON UNIVERSITY

Linear algebra multiplication

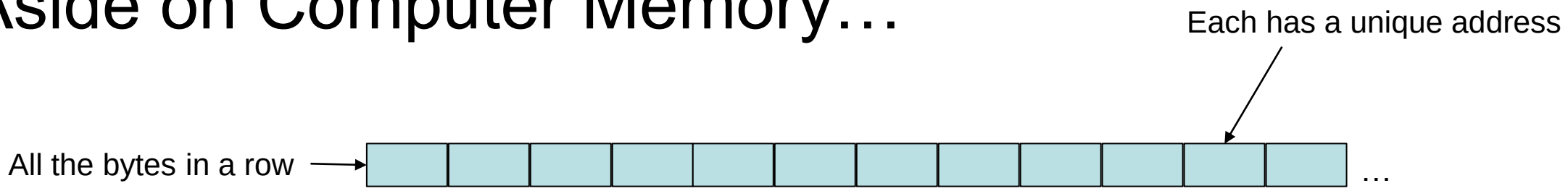
- Vector/matrix multiplication can be done using the *dot()*, *cross()* functions, or @ operator

- There are many other linear algebra routines!

```
a = np.array([[1, 0],
              [0, 1]])
b = np.array([[4, 1],
              [2, 2]])
np.dot(a, b)  # --> array([[4, 1],
              #              [2, 2]])
a @ b         # --> array([[4, 1],
              #              [2, 2]])
np.cross(a,b)# --> array([ 1, -2])
```

https://docs.scipy.org/doc/numpy/reference/routines.linalg.html

An Aside on Computer Memory…



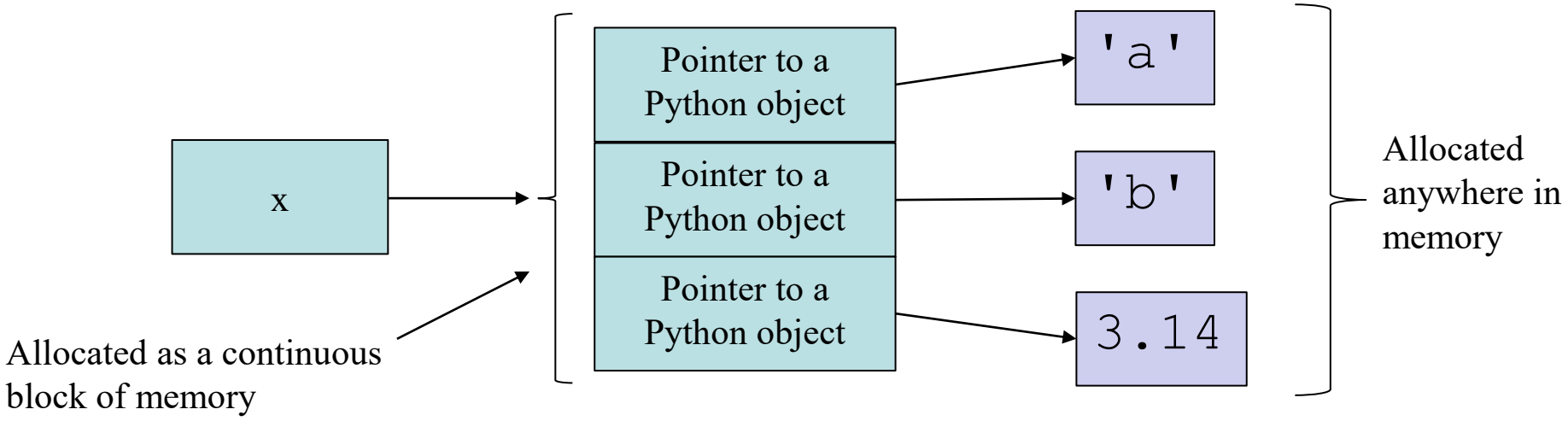
- All of the memory in a computer is accessed (and connected electronically) as a 1D array. Any byte can be read via its unique address.
- When a program reads data from memory the fastest way to do so is to read data from adjacent addresses.
  - Computer memory hardware is optimized for this access pattern.
- Let's contrast how Python lists and ndarrays organize themselves in memory…

BOSTON UNIVERSITY

List Implementation

```
x = ['a','b',3.14]
```

- A Python list mimics a *linked list* data structure when used
  - It's implemented as a resizable array of pointers to Python objects for performance reasons.

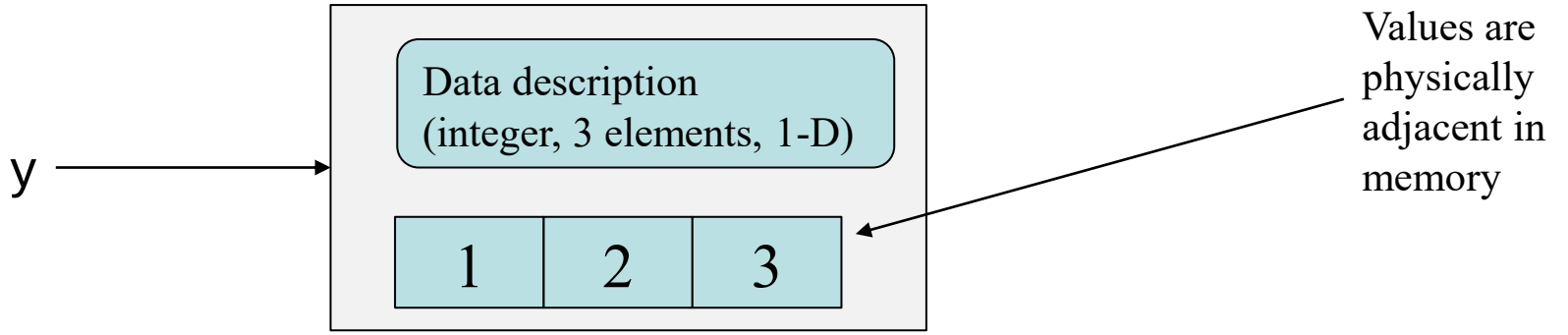


- `x[1]` → get the *pointer* (memory address) at index 1 → get the Python object in memory at that address → get the value from the object → return 'b'

NumPy ndarray

```
import numpy as np
# Initialize a NumPy array
# from a Python list
y = np.array([1,2,3])
```

- The basic data type is a class called *ndarray.*
- The object has:
  - a data that describes the array (data type, number of dimensions, number of elements, memory format, etc.)
  - A **contiguous** array in memory containing the data.



- `y[1]` → check the ndarray data type → retrieve the value at offset 1 in the data array → return 2

https://docs.scipy.org/doc/numpy/reference/arrays.html

dtype

- Every ndarray has a *dtype*, the type of data that it holds.
  - Current list of data types.
- This is used to interpret the block of data stored in the ndarray.
- Can be assigned at creation time:
- Conversion from one type to another is done with the *astype()* method:

```
a = np.array([1,2,3])
a.dtype
# dtype('int32')
```

```
c = np.array([-1,4,124],dtype='int8')
c.dtype
# dtype('int8')
```

```
b = a.astype('float64')
b.dtype
# dtype('float64')
```

BOSTON UNIVERSITY

ndarray from numpy initialization

- There are a number of initialization routines. They are mostly copies of similar routines in Matlab.
- These share a similar syntax:

```
function_name([size of dimensions list], opt. dtype…)
```

  - zeros – everything initialized to zero.
  - ones – initialize elements to one.
  - empty – do not initialize elements
  - identity – create a 2D array with ones on the diagonal and zeros elsewhere
  - full – create an array and initialize all elements to a specified value
- Read the docs for a complete list and descriptions.

ndarray from a list

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

- The numpy function *array* creates a new array from any data structure with array like behavior (other ndarrays, lists, sets, etc.)
- Read the docs!

- Creating an ndarray from a list does not change the list.

- Often combined with a reshape() call to create a multi-dimensional array.

- Open the file *ndarray_basics.py* in Spyder so we can check out some examples.

BOSTON UNIVERSITY

ndarray indexing

- ndarray indexing is similar to Python lists, strings, tuples, etc.
- Index with integers, starting from zero.
- Indexing N-dimensional arrays, just use commas:

```
array[i,j,k,l] = 42
```

```
oneD = np.array([1,2,3,4])
twoD = oneD.reshape([2,2])

twoD →  array([[1, 2],
               [3, 4]])


# index from 0
oneD[0] →  1
oneD[3] →  4

# -index starts from the end
oneD[-1] → 4
oneD[-2] → 3

# For multiple dimensions use a comma
# matrix[row,column]
twoD[0,0] → 1
twoD[1,0] → 3
```

BOSTON UNIVERSITY

ndarray slicing

- Syntax for each dimension (same rules as lists):
  - start:end:step
  - start: → from starting index to end
  - :end → start from 0 to end (exclusive of end)
  - : → all elements.

- Slicing an ndarray does **not** make a copy, it creates a **view** to the original data.

- Slicing a Python list creates a copy.

```
y = np.arange(50,300,50)
# y --> array([ 50, 100, 150, 200, 250])


y[0:3] --> array([ 50, 100, 150])
y[-1:-3:-1] --> array([250, 200])

x = np.arange(10,130,10).reshape(4,3)
#    x --> array([[ 10,  20,  30],
                  [ 40,  50,  60],
                  [ 70,  80,  90],
                  [100, 110, 120]])

# 1-D returned!
x[:,0] --> array([ 10,  40,  70, 100])
# 2-D returned!
x[2:4,1:3] --> array([[ 80,  90],
                      [110, 120]])
```

BOSTON UNIVERSITY

Look at the file *slicing.py*

ndarray slicing assignment

- Slice notation on the left hand side of an = sign overwrites elements of an ndarray. Can be combined with right hand slicing.

```
y = np.arange(50,300,50)
#  y --> array([ 50, 100, 150, 200, 250])

y[0:3] = -1
#  y --> array([ -1, -1, -1, 200, 250])

y[0:8] = -1
# NO ERROR!
# y --> array([ -1, -1, -1, -1, -1])

x = np.array([10,20,30,40,50,60])

y[0:2] = x[3:5]
# y --> array([ 40,  50, -1, -1, -1])
```

BOSTON UNIVERSITY

ndarray addressing with an ndarray

- Ndarray's can be used to address/index another ndarray.
- Use integer or Boolean values.
- Remember: this still returns a view.

```
a=np.linspace(-1,1,5)
# a --> [-1. , -0.5,  0. ,  0.5,  1. ]

b=np.array([0,1,2])

a[b] # --> array([-1. , -0.5,  0.])

c = np.array([True, False, True, True,
              False])

# Boolean indexing returns elements
# where True
a[c]  # --> array([-1. ,  0. ,  0.5])
```

numpy.where

- Similar to *find* in Matlab.
- Syntax:

```
numpy.where(condition, [x,y])
```

  - Condition: some Boolean condition applied to an ndarray
  - x, y: Optional variables to choose from. x is for `condition == True`, y is for `condition == False`.
  - All three arguments must apply to ndarray's.

```
a=np.linspace(-1,1,5)
# a --> [-1. , -0.5,  0. ,  0.5,  1. ]

# Returns a TUPLE containing the INDICES where
# the condition is True!
np.where(a <= 0)
# --> (array([0, 1, 2], dtype=int64),)

np.where(a <= 0, -a, 20*a)
# --> array([ 1. ,  0.5, -0. , 10. , 20. ])
```

BOSTON UNIVERSITY

Random Numbers

- First…the old school way as you see it all the time in existing code.

- Library: numpy.random
  - Library of functions for random number generation
  - Ex:
    - `numpy.random.random()` - generate random numbers from a uniform distribution
    - `numpy.random.shuffle()` – randomly re-order an ndarray

Random Numbers the right way

- Still part of *numpy.random*
- Create a random generator object
  - The default algorithm is called PCG64. There are others available.
- You can seed it to get a reproducible sequence
- Use the object to access functions involving randomness.
- This bit of code is in the file *sample_rng.py*

```
# Make an RNG object with a random seed
rng = np.random.default_rng()

# an ndarray of the integers 0-5
a = np.arange(6)
# shuffle a in place
rng.shuffle(a)
# a is now something like:
#     array([3, 0, 2, 4, 1, 5])

# Create an rng object from a specified seed
rng2 = np.random.default_rng(seed=100)
# the RNG sequence is deterministic, so
# everyone using the same seed and algorithm
# gets the same sequence. Get 4 numbers
# from a Gaussian distribution
rng2.normal(size=4)
# array([-1.15754965,  0.2897558 ,  0.78085407,  0.54397364])


# Simulate rolling dice
# Check the docs:
help(rng.integers)

# Get some random integers 1-5
dice_roll = rng.integers(1,high=6,size=10)
# no we want 1-6 for a 6-sided die,
# so add the endpoint argument
dice_roll = rng.integers(1,high=6,size=10,endpoint=True)
```

BOSTON UNIVERSITY

ndarray memory usage

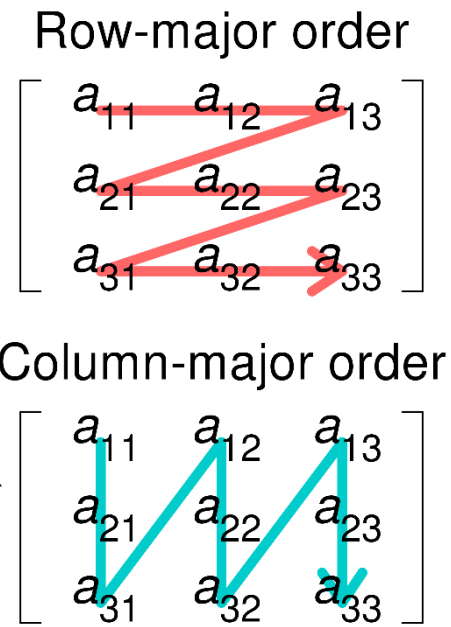
- The memory allocated by an ndarray:
  - Storage for the data: N elements * bytes-per-element
    - 4 bytes for 32-bit integers, 8 bytes for 64-bit floats (doubles), 1 byte for 8-bit characters etc.
  - A small amount of memory is used to store info about the ndarray (~few dozen bytes)
- Data storage is compatible with external libraries
  - C, C++, Fortran, or other external libraries can use the data allocated in an ndarray directly without any conversion or copying.

ndarray memory layout

- The memory layout (C or Fortran order) can be set:
  - This can be important when dealing with external libraries written in R, Matlab, etc.

```
X = np.ones([3,5],order='F')
# OR...
# Y is C-ordered by default
Y = np.ones([3,5])
# Z is a F-ordered copy of Y
Z = np.asfortranarray(Y)
```

- Row-major order: C, C++, Java, C#, and others
- Column-major order: Fortran, R, Matlab, and others
- See here for more detail



https://en.wikipedia.org/wiki/Row-_and_column-major_order

ndarray memory layout

- For row-major ordering the rightmost index accesses values in adjacent memory.
- The opposite is true for column-major ordering.
- This can have a significant impact on performance when accessing ndarrays with *for* loops or built-in functions like *numpy.sum()*

```
# Y is C-ordered by default
Y = np.ones([2,3,4])
# For loop indexing:
total=0.0
for i in range(Y.shape[0]):
    for j in range(Y.shape[1]):
        for k in range(Y.shape[2]):
            total += Y[i,j,k]

# X is Fortan-ordered
X = np.ones([2,3,4], order='F')
# For loop indexing:
total=0.0
for i in range(X.shape[2]):
    for j in range(X.shape[1]):
        for k in range(X.shape[0]):
            total += X[k,j,i]
```

Look at the file *row_vs_col_timing.py*

NumPy I/O

- When reading files you can use standard Python, use lists, allocate ndarrays and fill them.
- Or use any of NumPy's I/O routines that will directly generate ndarrays.
- The best way depends on the structure of your data.
- If dealing with structured numeric data (tables of numbers, etc.) NumPy is easier and faster.
- Docs: https://docs.scipy.org/doc/numpy/reference/routines.io.html

Numpy docs

- As numpy is a large library we can only cover the basic usage here

- Let's look that the official docs:

https://docs.scipy.org/doc/numpy/reference/index.html

- As an example, computing an average:

https://docs.scipy.org/doc/numpy/reference/generated/numpy.mean.html#numpy.mean

Some numpy file reading options

- .npz and .npy file formats (cross-platform compatible) :
  - .npy files store a single NumPY variable in a binary format.
  - .npz files store multiple NumPy Variables in a file.

- h5py is a library that reads HDF5 files into ndarrays

- The I/O routines allow for flexible reading from a variety of text file formats

```
numpy.save  # save .npy
numpy.savez # save .npz
# ditto, with compression
numpy.savez_compressed

numpy.load  # load .npy
numpy.loadz # load .npz
```

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

BOSTON UNIVERSITY

NumPy I/O example

- Read in a data file from a set of ocean weather buoys.
  - File: *buoy_data.csv*
  - 18 columns. 1st column are dates, the rest are numeric data for different buoys.
  - Some rows have dates but are missing data points in some columns.
- Use the most flexible NumPy file reader, *genfromtxt*.
- Task: read temperature data into a 2D ndarray, clean up missing data, plot the data.

BOSTON UNIVERSITY

Look at the file *numpy_io.py*

SciPy

- SciPy builds on top of NumPy.
- Ndarrays are the basic data structure used.
- Libraries are provided for: →

- physical constants and conversion factors
- hierarchical clustering, vector quantization, K-means
- Discrete Fourier Transform algorithms
- numerical integration routines
- interpolation tools
- data input and output
- Python wrappers to external libraries
- linear algebra routines
- miscellaneous utilities (e.g. image reading/writing)
- various functions for multi-dimensional image processing
- optimization algorithms including linear programming
- signal processing tools
- sparse matrix and related algorithms
- KD-trees, nearest neighbors, distance functions
- special functions
- statistical functions

- Comparable to Matlab toolboxes.

scipy.io

- I/O routines support a wide variety of file formats:

Software	Format name	Read?	Write?
Matlab	.mat	Yes	Yes
IDL	.sav	Yes	No
Matrix Market	.mm	Yes	Yes
Netcdf	.nc	Yes	Yes
Harwell-Boeing (sparse matrices)	.hb	Yes	Yes
Unformatted Fortran files	.anything	Yes	Yes
Wav (sound)	.wav	Yes	Yes
Arff (Attribute-Relation File Format)	.arff	Yes	No

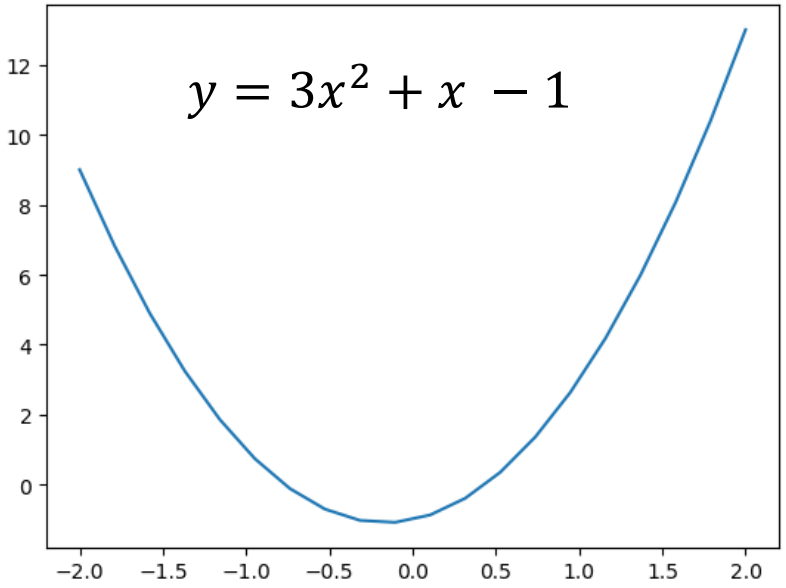
Using SciPy

- Think about your code and what sort of algorithms you're using:
  - Integration, statistical sampling, linear algebra, image processing, etc.
- See if an appropriate algorithm exists in SciPy before trying to write your own.
- Read the docs – many functions have large numbers of optional arguments.
- Understand the algorithms!

Example: Fit a line with SciPy

- There are many ways to fit equation parameters to data in NumPy and SciPy
- scipy.stats.linregress: Calculate a regression line
- Open the example *linregress.py*
- This demonstrates calling the function and extracting all the info it returns.

Example: scipy.optimize.minimize



- Finds the minimum value of a function.
- You provide the function as an argument to *minimize.*
- Using functions as arguments is a common pattern in scipy.

- Open *scipy_minimize.py*

BOSTON UNIVERSITY

https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html

OpenCV

- The *Open Source Computer Vision Library*
  - They have a Python tutorial
  - When added to your Python setup:
    - *import cv2*
- Highly optimized and mature C++ library usable from C++, Java, and Python.
- Cross platform: Windows, Linux, Mac OSX, iOS, Android
- OpenCV data in Python is implemented with ndarrays

- Image Processing
- Image file reading and writing
- Video I/O
- High-level GUI
- Video Analysis
- Camera Calibration and 3D Reconstruction
- 2D Features Framework
- Object Detection
- Deep Neural Network module
- Machine Learning
- Clustering and Search in Multi-Dimensional Spaces
- Computational Photography
- Image stitching

BOSTON UNIVERSITY

OpenCV vs SciPy

- For imaging-related operations and many linear algebra functions there is a lot of overlap between these two libraries.
- OpenCV functions are frequently faster, sometimes significantly so.
- The OpenCV Python API uses NumPy ndarrays, making OpenCV algorithms compatible with SciPy and other libraries.

OpenCV vs SciPy

- A simple benchmark: Gaussian and median filtering a 1024x671 pixel image of the CAS building.
- Gaussian: radius 5, median: radius 9.
- Timing: 2.4 GHz Xeon E5-2680 (Sandybridge)

See: *image_bench.py*

Operation	Function	Time (msec)	OpenCV speedup
Gaussian	scipy.ndimage.gaussian_filter	85.7	3.7x
	cv2.GaussianBlur	23.2	
Median	scipy.ndimage.median_filter	1,780	22.5x
	cv2.medianBlur	79.2	

BOSTON UNIVERSITY

End-of-course Evaluation Form

- Please visit this page and fill in the evaluation form for this course.
- Your feedback is highly valuable to the RCS team for the improvement and development of tutorials.
- If you visit this link later please make sure to select the correct tutorial – name, time, and location.

http://rcs.bu.edu/eval

BOSTON UNIVERSITY