NumPy and Matplotlib
Instructor: Jorge Mendez
What is NumPy?

• Python’s main scientific computing package
• Main object: N-dimensional array (ndarray)
• Lots of compiled operations (fast!)
  • Math
  • Logic
  • I/O
  • Linear algebra
  • Statistics
  • Randomization
  • ...
ndarray

• Basis for most scientific computing packages
• Homogeneous type (typically numbers)
• Fixed size: changing the size of an ndarray creates a new one
  • Memory footprint *may* change dynamically, since elements of an ndarray can be arbitrary Python objects
ndarray

• Supports advanced operations on large amounts of data with less code and more efficiently
• Element-by-element operations (+, −, *, . . . ) are default
• Each dimension is an axis
  • By convention, last two axes correspond to rows and columns of matrices
Creating arrays

- **np.array(sequence)** — copy elements of sequence to an array
  - Type of the array is deduced from element types in sequence
  - Optional argument `dtype` to specify the array type
  - Nested sequences of depth N are transformed into N-dimensional arrays

- **np.zeros(shape), np.ones(shape), np.full(shape, val)** — array of all-zeros/ones/val with fixed shape
  - Avoids growing size if elements are initially unknown but shape is known
  - `shape` is a tuple of axis sizes

- **np.empty(shape)** — array of arbitrary elements with fixed shape

- **np.zeros_like(a), np.ones_like(a), np.full_like(a), np.empty_like(a)** — copy shape from a
Creating arrays

- `np.arange(start, stop, step)` — analogous to `range()`
  - All arguments may be floating points
  - E.g., `np.arange(0, 1, 0.1)`
  - Floating point precision makes exact number of resulting elements hard to predict

- `np.linspace(start, stop, num_elements)` — like `np.arange()`, but with fixed number of elements
Vectorization

- Vectorized code avoids explicit indexing and looping over arrays
  - Handled behind-the-scenes by optimized, pre-compiled C code

```python
for i in range(len(a)):
    c.append(a[i] * b[i])  # c = a * b
```

- For large arrays, vectorized operations are often significantly faster than equivalent loop operations
- Vectorization is a different way of thinking about code, but well worth learning
- Vectorized code typically resembles mathematical notation
  - E.g., `a.dot(b) + c` is NumPy for `a \cdot b + c`
Basic operations

• Arithmetic operators (+, −, *, /) are applied element-wise

• Matrix product is performed with a @ b or a.dot(b)
  • Not with *
  • Nobody uses @ notation

• Some operations (+=, −=, *=, /=) act in place
Live Example
Unary and universal operations

• Many unary operations (sum, max, min, cumsum) are methods of ndarray
  • E.g., a.max()
  • Operate by default on “flattened” array
  • Optional argument axis indicates dimension along which to operate

• Universal operations (exp, sin, cos, sqrt) also operate element-wise
Live Example
Indexing, slicing, and iterating

• One-dimensional arrays are indexed and sliced like sequences
• Multi-dimensional arrays have one index per axis
  • Entire index is given as a tuple
  • Each index can itself be a slice
  • \(a[::10, 1]\) — first ten elements of the second column
  • \(a[-10:,:]\) — last ten elements of all columns
  • Missing slices are treated as complete slices (\(\ldots\))
  • Use dots (\(\ldots\)) to indicate all needed complete slices
    • E.g., \(a[::3, 1:6, \ldots, 5:]\)
• Iterating is done over first axis
  • Use \(a.flat\) to iterate element-wise
Live Example
Modifying an array’s shape

- `a.ravel()` — return flattened array into a single dimension
- `a.reshape(new_shape)` — return array with modified shape
- `a.T` — return transposed array
  - For N-D arrays, revert the order of the axes
- `a.resize(shape)` — same as reshape, but modify `a` directly
- If an axis is `−1`, number of elements is automatically computed
- Default order is C-style: last index changes fastest
  - Optional argument `order=‘F’` uses FORTRAN-style: first index changes fastest
Live Example
Broadcasting

• Many functions can operate on arrays that do not have the same shape
• Broadcasting prescribes the behavior of all functions in these cases
• Two simple rules
  1. If not all arrays have the same number of axes, 1s are prepended to smaller arrays until all arrays have the same number of dimensions
  2. Every axis of size 1 is treated as if it were the same size as the largest array along that axis, assuming all values are the same as the singleton element
• At this point, it must hold that all arrays have the same shape, or they can’t be broadcast together
Live Example
Indexing with arrays of indices

• \( a[\text{idx\_arr}] \) — the \( \text{idx\_arr} \) elements of \( a \)
  • Each element of \( \text{idx\_arr} \) is treated as an index into \( a \)
  • If \( a \) is multidimensional, elements index over the first axis
  • The first dimensions of the resulting array match \( \text{idx\_array} \)’s shape
  • The last dimensions of the resulting array match \( a \)’s shape after the first axis

• \( a[\text{idx\_i}, \text{idx\_j}] \) — index over two dimensions of \( a \)
  • \( \text{idx\_i} \) and \( \text{idx\_j} \) must be the same shape
  • Each \( \text{idx\_i}, \text{idx\_j} \) pair is treated as an index into the first two axes of \( a \)

• Both can also be used for assigning values to \( a \)
Indexing with Boolean arrays

- **a[bool_array]** — elements of a for which bool_array is True
  - bool_array is the same shape as a
  - Resulting array is 1-dimensional
  - Useful for getting elements that match a condition

- **a[bool_i, bool_j]** — elements of a for which both bool_i and bool_j are True
  - bool_i, bool_j must be 1-dimensional
  - Their size must be the same as a’s corresponding axis
  - Typically only one or the other: keep all rows/columns matching some condition, slice the other dimension
Live Example
The \texttt{ix()} function

- Indexing with arrays doesn’t directly allow us to get a given set of rows and a given set of columns
- Broadcasting combined with indexing does
- \texttt{ix()} gives us the required broadcastable arrays
Linear algebra

• `np.linalg.inv(a)` — matrix inverse
• `np.eye(n)` — identity matrix of size n
• `np.trace(a)` — trace of a (sum of diagonal elements)
• `np.linalg.solve(a, b)` — solve $ax = b$ for $x$
• `np.linalg.eig(a)` — eigenvalues and eigenvectors of a
There’s much more to NumPy

• This barely covers NumPy’s quickstart tutorial!
• It’s impossible to learn all of NumPy’s functionality
• So how do you know when NumPy has the function you need?
  • Usually, if you are looping through an array, you can vectorize your code
  • If fancy indexing is not enough, then there might be a NumPy function for what you need
• Where to look next?
  • I/O
  • np.random
  • Structured arrays
  • scipy
Matplotlib
What is Matplotlib?

• Python’s primary plotting package
• Widely used for data visualization
• Easy to use for simple visualizations, but allows for fine-grained control for experienced users
• We will look only into `matplotlib.pyplot`
  • Highest-level module
  • Create figures, add elements such as lines and text
Parts of a figure

• **Figure**: the whole figure. Contains **Axes**, artists (titles, legends). Should have at least one **Axes**

```python
import matplotlib.pyplot as plt
fig = plt.figure()  # no Axes
fig.suptitle('figure_title')
plt.show()
```
Parts of a figure

• **Axes**: a plot. There may be more than one per Figure. Contains Axis objects (2 for 2-D plots, 3 for 3-D). Has a title, an x-label, and a y-label.

```python
fig, axes_lst = plt.subplots(2, 2)  # 2x2 grid of Axes
fig.suptitle('Title')
axes_lst[0, 0].set_title('Axes 0')
axes_lst[0, 0].set_ylabel('y')
axes_lst[0, 0].set_xlabel('x')
plt.show()
```

• **Axis**: number-line objects. Set graph limits and ticks
Inputs to plotting functions

- `np.array` is the expected type by all plotting functions
- Array-like objects (lists, tuples, Pandas dataframes) should be converted to arrays by the user
“Current”

- All functions in pyplot can refer to a “current” Figure and Axes
- Both can be created automatically
- This avoids the need to manually create and refer to figures/axes
Live Example
LOTS more to pyplot

• What we covered doesn’t even make a dent into pyplot’s capabilities
  • Let alone matplotlib...

• Where to go from here?
  • Formatting: markers and colors for each point
  • Scatter plots: no curve generation
  • Bar charts: for categorical variables
  • Managing multiple figures and multiple axes within a figure
  • 3-D plotting

• The documentation is very complete, and there is a huge community using matplotlib
Takeaways

• NumPy provides a very strong basis for scientific computing
• ALWAYS vectorize your code when working with NumPy
  • Sometimes the speedups are quite surprising
• NumPy provides some fancy indexing beyond Python’s basic indexing
  • Really useful for vectorizing code
• Matplotlib gives us lots of basic plotting functionality