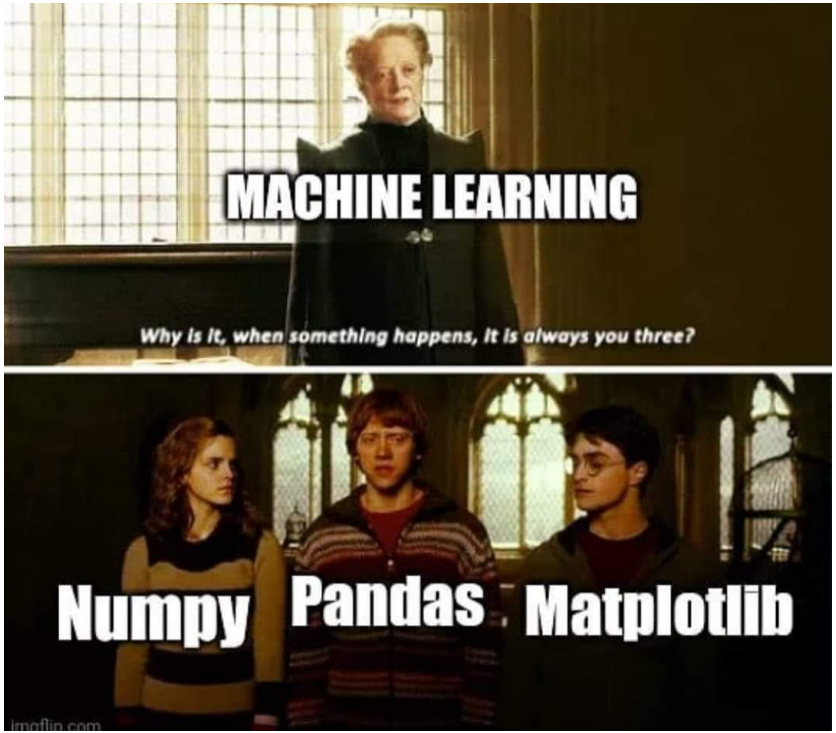
Python Libraries: numpy and matplotlib

Kameswari Chebrolu



Numpy (Numeric Python)

- Library used for high performance computing and data analysis
  - High-level math functions involving arrays/matrices
  - Fast numerical computations like matrix multiplication; linear algebra/fourier transform etc
- Very efficient for large arrays of data
  - Stores data contiguously ⟶ less memory, faster operation (10-100 times faster)
    - 1000 x 1000 matrix multiply
      - Python triple loop takes > 10 min.
      - Numpy takes ~0.03 seconds
  - Batch operations can be managed without writing loops (vectorization)

Ndarray (n-dimentional-array)

- ndarray used for storage of homogeneous data
  - All elements the same type
- Supports convenient slicing, indexing and efficient vectorized computation
- Every array must have a shape and a dtype (data type)
  - Vector: array in single dimension
  - Matrix: array in two dimensions
  - Tensor: 3-D or higher dimensional arrays

In NumPy, every array has:

Shape (.shape) → The number of elements along each axis (its a tuple)

Dimensions (.ndim) → The number of axes in the array

Array Type	Example	Shape (.shape)	Dimensions (.ndim)
1D Array	`[1, 2, 3, 4, 5]`	`(5,)`	`1`
2D Array	`[[1, 2, 3], [4, 5, 6]]`	`(2, 3)`	`2`
3D Array	`[[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]]`	`(2, 2, 3)`	`3`

Create Arrays

- Arrays in NumPy are created using np.array(), which can take lists, tuples or other array-like structures (e.g. range())
- Arrays can have different dimensions, from 0-D (scalars) to multi-dimensional (e.g., 2D matrices, 3D tensors, etc.)
- ndmin parameter allows specifying the minimum number of dimensions explicitly

- Different Data Types in Arrays
  - Integer (i) – Whole numbers (e.g., int8, int16, int32, int64)
  - Floating point (f) – Decimal numbers (e.g., float16, float32, float64)
  - Boolean (b) – True or False values
  - Unsigned integers (u) – Positive integers only
  - Complex (c) – Numbers with a real and imaginary part
  - String (S or U) – Fixed-size byte or Unicode string

Data types can be explicitly set while creating arrays or modified later

- Converting Data Types: Arrays can be converted from one type to another using .astype()
  - Float → Integer (truncating decimals)
  - Integer → Boolean (non-zero values become True)
  - Numeric → String (numbers stored as character representations)

Copy vs View

- Copy of an array creates a completely independent object with its own data
  - Changes in the original array do not affect the copy
- View is a reference to the same memory location as the original array
  - Changes in one affect the other
- .base attribute helps check if an array is a view
  - Will reference the original array or a standalone copy (it will return None)

See 01-arrays.py

Special Arrays

- Zeros Array (np.zeros()): Creates an array filled entirely with zeros
  - Useful when you need to initialize an array before performing operations on it
- Ones Array (np.ones()): Creates an array filled entirely with ones
- Identity Matrix (np.identity()): Creates a square matrix where the diagonal elements are 1, and all other elements are 0
- Eye Array (np.eye()): Similar to the identity matrix, but it can create non-square matrices as well
  - Diagonal is filled with ones, and all other elements are zeros

- Arange Array (np.arange()): Creates an array of evenly spaced values within a given range
  - Similar to Python's range() function but returns a NumPy array
- Linspace Array (np.linspace()): Creates an array of evenly spaced values between two endpoints
  - You specify the number of elements, rather than the step size, unlike arange()
- Random Arrays:
  - np.random.rand(): Generates random numbers from a uniform distribution between 0 and 1
  - np.random.randn(): Generates random numbers from a standard normal distribution (mean=0, standard deviation=1)
  - np.random.randint(): Creates an array of random integers within a specified range

- Random Choice (np.random.choice()): Randomly selects elements from a specified list or array
- Empty Array (np.empty()): Creates an array without initializing the entries
  - Values in the array can be random
- Diagonal Array (np.diag()): Creates an array with a given list of values as the diagonal
  - Can also be used to extract the diagonal of an existing array

See 02-special-arrays.py

Indexing

- Indexing allows one to access individual elements in arrays
  - 1D arrays use a single index, while 2D or higher-dimensional arrays use a combination of indices for each axis
- Slicing: extract a portion of an array using the start, stop, and step parameters
  - Can slice arrays in one or more dimensions
  - For 1D arrays, one can specify a range to get a subsequence
  - For 2D arrays, one can slice rows or columns independently
  - Can also slice both dimensions simultaneously, specifying the range for both rows and columns

- Stepping: allows one to select elements at regular intervals
  - In one-dimensional arrays, stepping can be applied to select every nth element
  - In multi-dimensional arrays, stepping can be applied to both rows and columns
- Negative Indexing: lets one access elements starting from the end of an array
  - an index of -1 refers to the last element, -2 to the second last, and so on

- Accessing rows: In a 2D array, rows can be accessed by using a single index for the row and a colon for all columns
- Accessing columns: Columns in a 2D array are accessed by fixing the column index and using a colon for all rows

- Ellipsis (...): Allow one to skip one or more dimensions when indexing making it easier to index large arrays without specifying each axi
  - Especially useful for working with arrays with many dimensions
- Boolean Indexing: Can index an array using a condition

See 03-indexing.py

Shaping

- Shaping refers to changing the structure of NumPy arrays
  - Can manipulate the number of dimensions and their size
- Reshaping: Can change the dimensions of an array without changing its data
  - Can convert a 1D array into a 2D matrix or change a 2D array into a 3D array
    - Total number of elements remains the same

- Flattening: Flatten converts a multi-dimensional array into a 1D array
- Transposing: Swaps the rows and columns of a 2D array
  - Changes the axis order
- Adding or Removing Dimensions
  - Expand_dims adds a new axis to the array, increasing its dimensions
  - Squeeze removes axes of “length 1” from the array, reducing its dimensions

Original Shape	Squeezed Shape	Why?
`(1, 1, 3)`	`(3,)`	Removed two `1` dimensions
`(1, 4, 1, 2)`	`(4, 2)`	Removed first and third dimensions (size `1`)

- Resizing: Can change the size of the array by adding or removing elements
  - Internally, it first flattens the array
  - For larger shapes, it repeats the flattened array until the new shape is filled
    - If there is still more space, it continues repeating the flattened array
  - For smaller shapes, it truncates the flattened array to fit the new shape
- Raveling: Ravel is similar to flattening, but it returns a view of the original array when possible (instead of a copy)
  - More memory efficient

See 04-shaping.py

Arithmetic

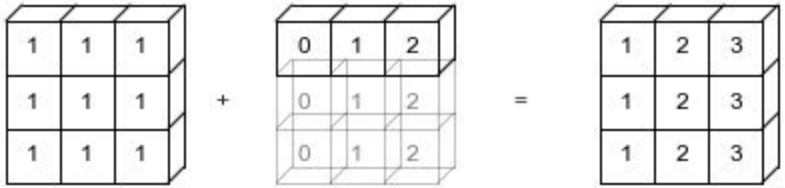
- NumPy provides powerful arithmetic operations that work element-wise on arrays
  - No need for loops; Optimized for performance
- Basic Arithmetic Operations
  - Supports addition, subtraction, multiplication, division, exponentiation, modulus, and floor division
  - When two arrays are involved, operations happen element by element
  - If a scalar (single number) is used with an array, NumPy broadcasts it

- Broadcasting: Allows operations between arrays of different shapes without explicitly resizing them
  - If an operation involves an array and a single value, NumPy automatically applies the operation to all elements of the array
  - If two arrays of different sizes are involved, NumPy aligns their shapes when possible

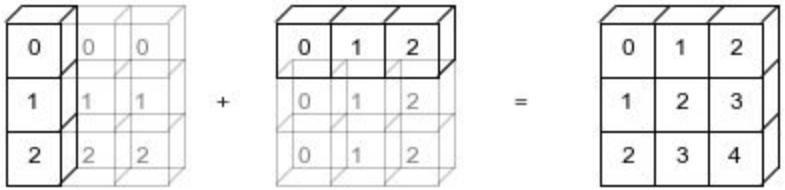
Broadcasting

`np.arange(3)+5`

`np.ones((3,3))+np.arange(3)`



`np.arange(3).reshape((3,1))+np.arange(3)`



- Universal Functions (ufuncs): special functions for mathematical operations like square root, logarithm, trigonometric functions, absolute value, and exponentiation
  - Work efficiently on large datasets and avoid the need for writing loops.

- Dot Product and Matrix Multiplication:
  - NumPy includes optimized functions for matrix operations like dot product and matrix multiplication
  - Matrix multiplication follows the rules of linear algebra

See 05-arithmetic.py

Miscellaneous Functions

- concatenate(): combines two or more arrays along a specified axis
- stack(): Joins multiple arrays along a new axis, effectively adding an additional dimension
  - Different from concatenate(), which works along an existing axis
- split(): splits an array into multiple sub-arrays along a specified axis
  - Number of splits depends on the number of sections you want to divide the array into

- array_split(): Similar to split(), but it allows for uneven splits
  - If the array doesn’t divide evenly, the remaining elements will be distributed as evenly as possible across the sub-arrays
- where(): returns elements from an array where a specified condition is true
  - Optionally returns a different set of values where the condition is false
- sort(): sorts the elements of an array in ascending or descending order, either along a specific axis or for the entire array

See 06-func.py

Matplotlib

- Used for drawing charts and for general visualization
  - Inspired by MATLAB (lot of common terms like axis, plots etc)
- Matplotlib is a package that provides a wide variety of plotting functions.
  - pyplot is a module within Matplotlib that offers a simplified interface for creating and managing plots
  - Can import it via the matplotlib.pyplot namespace
  - Each pyplot function makes some change to a figure
    - E.g. create a plotting area, draw some lines, decorates the plot with labels etc

Basic Flow of Pyplot Operations:

Create a plot – Use plot(), scatter(), bar(), etc.

Customize the plot – Add titles, labels, grid, legends, etc.

Show the plot – Use show() to display it.

Function	Description
`plot()`	Creates a basic line plot.
`scatter()`	Generates a scatter plot.
`bar()`	Creates a bar chart.
`hist()`	Plots a histogram.
`pie()`	Generates a pie chart.
`xlabel()`, `ylabel()`	Labels the X and Y axes.
`title()`	Adds a title to the plot.
`legend()`	Displays a legend.
`grid()`	Adds a grid to the plot.
`show()`	Displays the figure.

plot

- Used to create line plots
  - Most commonly used functions for visualizing data trends
  - Connects data points with straight lines by default
  - Supports multiple lines in a single plot
  - Allows customization of lines, markers, colors, labels, and more

plot(x, y, format_string, **kwargs)

x – Data for the x-axis (optional; if not provided, uses index positions)

y – Data for the y-axis

format_string (optional) – Specifies color, marker, and line style in a short format.

**kwargs – Additional keyword arguments for customization

Parameter	Description	Example
`color` (`c`)	Line color	`plot(x, y, color='red')`
`linestyle` (`ls`)	Type of line (`'-', '--', '-.', ':'`)	`plot(x, y, linestyle='--')`
`linewidth` (`lw`)	Line thickness	`plot(x, y, linewidth=2.5)`
`marker`	Marker style (`'o', 's', '*', '+', 'x'`)	`plot(x, y, marker='o')`
`markersize` (`ms`)	Size of the marker	`plot(x, y, marker='o', markersize=8)`
`label`	Legend label	`plot(x, y, label='My Line')`

show

- Used to display all the figures and plots that have been created up to that point in the script
- show() is a blocking function
  - Will halt the execution of code until the plot window is closed
  - Can interact with the plot (e.g., zoom in/out, save the plot) before the script continues or finishes

See 01-plot.py

Labels, Title, Grid and Legend

- Labels, title, and grid are used to enhance the readability and structure of plots
- Labels are used to annotate the axes of the plot
  - Helps viewer understand what the x-axis and y-axis represent

- Title: A short description or label at the top of the plot that gives context about the plot as a whole
  - Helps viewers quickly understand the purpose of the plot
- Grid: adds horizontal and vertical lines to the background of the plot
  - Makes it easier to read the values at any given point
  - Particularly useful in scatter plots, line plots, and other types of data visualizations that require precise reading of values

- Legend: used to provide an explanation for the different plot elements
  - Especially useful with multiple data series or lines in the same plot
  - Matplotlib can automatically generate a legend based on the labels
    - Label is typically shown in the legend corresponding to the plot element it is associated with
  - Can specify the location of the legend, customize its font properties, and adjust its appearance etc
  - Can include the label directly in the plot() function or use the plt.legend() method to control the labels after plotting

See 02-title-labels-grid-legend.py

subplot

- Used to create multiple plots (subplots) within a single figure
  - Helps to organize multiple graphs within one window, allowing one to compare different datasets side by side or in a grid layout
- Divides the figure area into a grid of rows and columns
  - Can specify the number of rows and columns, and then can select which subplot (cell) to plot in

- Generally called with three arguments: nrows, ncols, and index
  - nrows specifies the number of rows in the grid
  - ncols specifies the number of columns in the grid
  - index specifies which subplot to activate (by numbering from left to right, top to bottom)

See 03-subplot.py

Different plots

Part of syllabus

Plot Type	Function	Use Case Example
Scatter Plot	`plt.scatter(x, y)`	Relationship between height and weight
Line Plot	`plt.plot(x, y)`	Stock market trend
Bar Plot	`plt.bar(x, y)`	Product sales comparison
Pie Chart	`plt.pie(sizes, labels)`	Vote distribution
Histogram	`plt.hist(data, bins=10)`	Test score distribution
Box Plot	`plt.boxplot(data)`	Comparing class test scores
Area Plot	`plt.fill_between(x, y)`	Cumulative sales over time
Stacked Bar	`plt.bar(x, y1, bottom=y2)`	Sales by product in regions
Heatmap	`plt.imshow(data)`	Visualizing correlations or images
Contour Plot	`plt.contour(X, Y, Z)`	Temperature distribution

See 04-plot-types.py

Save plots

- savefig() function is used to save the current figure (plot) to a file
  - plt.savefig('filename') saves the current plot to the file specified by the filename argument
  - Can save plots in various formats, such as png, pdf, svg, jpg etc
    - Format is automatically determined by the file extension you provide

- DPI (Dots per Inch): DPI controls the resolution of the saved image
  - Higher DPI means higher resolution
  - Default DPI is 100, but you can adjust it for better quality
- Transparent Background: Can save the plot with a transparent background by setting the transparent parameter to True
- Bounding Box (Bbox): bbox_inches parameter determines which part of the figure to save
  - Use 'tight' to save only the area occupied by the plot and remove any extra whitespace

See 05-save-file.py

References

- Numpy: https://www.w3schools.com/python/numpy/default.asp
- Matplotlib: https://www.w3schools.com/python/matplotlib_pyplot.asp