NumPy Tools for Data Science

NumPy, vectorization

Upcoming Assignments

- Reading: 2.1 – 2.5
- DS Lab 2: Vectorization due Jan 17th 11:59 pm

Today

- NumPy Broadcasting
- Array Slicing
- Linear Algebra with np
- Vector vs Non-Vector Operations

Tim Kapp
tkapp@byu.edu
TMCB 2254 — By Appointment

Teaching Assistants:

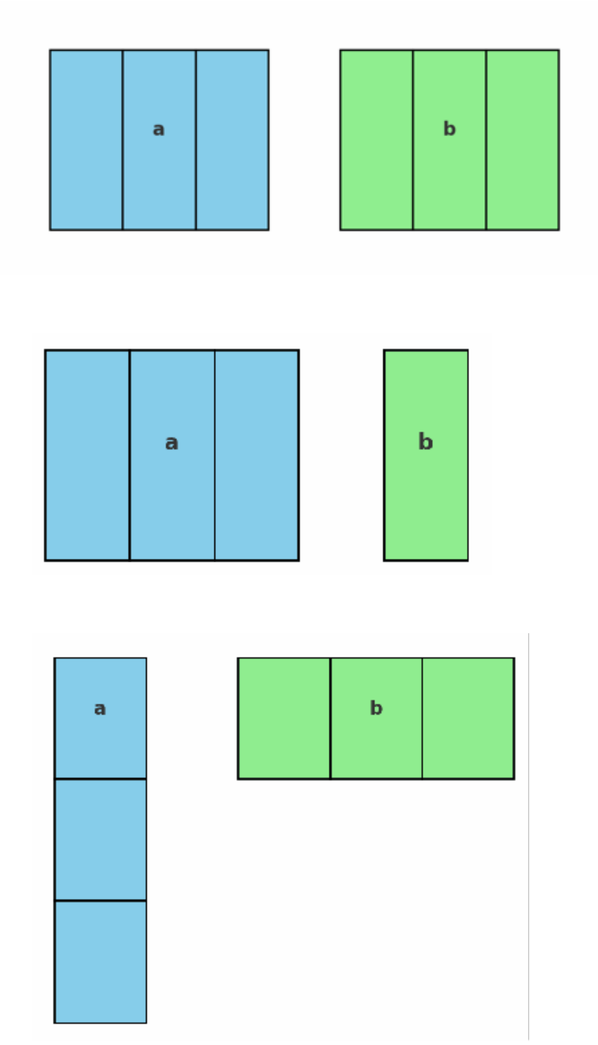
- Kayla Ou -- ouj22@byu.edu
- Spencer Hales — srhales@byu.edu
- Spencer Marshall — sm892@byu.edu
- Michael Jensen — mikelj01@student.byu.edu
- Toby Alley — talley0@byu.edu

West View Building (WVB 1151)
Office Hours: See Syllabus on Canvas

Class #	Week #	Month	Date	Topic	Reading	Labs
1	1	**Jan**	**8**	Welcome, Introduction, Course Objectives, DS Lifecycle	Chapter 1 Intro DS	Lab 1: Colab Set Up, GitHub
2	2	**Jan**	**13**	Python setup, Google colab, Github		
3	2	**Jan**	**15**	NumPy, Vectorization	Chapter 2.1-2.5 Python	Lab 2: Vectorization
4	3	**Jan**	**20**	Pandas, Matplotlib, Seaborn		
5	3	**Jan**	**22**	Data Cleaning and Preparation	Chapter 2.6-2.8 Python	Lab 3: NumPy, Pandas
6	4	**Jan**	**27**	Data Acquisition, ETL, Populations, Sampling	Chapter 3 Data Prep	
7	4	**Jan**	**29**	Descriptive Statistics	Chapter 4 Prob & Stat	Lab 4: Data Preparation
8	5	**Feb**	**3**	Exploratory Data Analysis (EDA)		
9	5	**Feb**	**5**	Principles of Data Visualization	Chapter 5 EDA	
10	6	**Feb**	**10**	Data management - databases, SQL queries		
11	6	**Feb**	**12**	More SQL Features, Joins	Chapter 7 DB an	
12	7	**Feb**	**17**	**MONDAY SCHEDULE, NO CLASS**		
13	7	**Feb**	**19**	SQLite		
14	8	**Feb**	**24**	MIDTERM REVIEW		
15	8	**Feb**	**26**	MIDTERM		
16	9	**Mar**	**3**	Overview of ML		
17	9	**Mar**	**5**	Unsupervised Learning- Kmeans	Chapter 8 Unsupervised Learning	
18	10	**Mar**	**10**	Unsupervised Learning- Hierarchical, DBSCAN		
19	10	**Mar**	**12**	Supervised Learning: Part 1	Chapter 9 Supervised Learn	Lab 8: Cluster Analysis
20	11	**Mar**	**17**	Supervised Learning: Part 2		
21	11	**Mar**	**19**	Evaluation of models, comparing performance	Chapter 10 Decision Trees	Lab 9: ML Classification/Regression
22	12	**Mar**	**24**	Feature Importance with RF and Logistic Regression		
23	12	**Mar**	**26**	ANN, Multi-Layer Perceptron, Backpropagation	Chapter 12 Eval	**New Lab?**
24	13	**Mar**	**31**	Deep Learning		
25	13	**Apr**	**2**	GenAI - Introduction	Chapter 13 ANN	Lab 10: MLP and Backpropagation

NumPy Array Broadcasting

- **Broadcasting allows NumPy to perform operations on arrays with different shapes by “virtually” expanding dimensions**
- Compatibility Rules:
  1. Dimensions have the same size, or
  2. One of the dimensions has a size of 1.
- If compatible, replicate the smaller dimension array to match the larger array.
- If one array has fewer dimensions, prepend 1s to its shape to match
- If, after alignment, any mismatch remains, then broadcasting fails.



NumPy Array Broadcasting

1. Same Shape → Elementwise Operations
If two arrays have the same shape, operations are applied element by element.

2. Scalar with Array
A scalar is treated as an array of the same shape as the other operand.

3. Different Shapes with Compatible Dimensions
Broadcasting works when dimensions are **equal** or **one of them is 1**.

4. Automatic Expansion of Size-1 Dimensions
If one array has a dimension of size 1, NumPy stretches it to match the other.

5. Broadcasting Fails if Incompatible
If shapes don't align according to the rules, NumPy raises an error.

```python
import numpy as np

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

print(a + b)   # [5 7 9]
print(a * b)   # [ 4 10 18]

a = np.array([1, 2, 3])

print(a + 5)   # [6 7 8]
print(a * 2)   # [2 4 6]


a = np.array([[1], [2], [3]])  # shape (3,1)
b = np.array([10, 20, 30])     # shape (3,)

# a is broadcast across columns, b across rows
print(a + b)
# [[11 21 31]
#  [12 22 32]
#  [13 23 33]]


a = np.array([[1, 2, 3]])   # shape (1,3)
b = np.array([[10], [20]])  # shape (2,1)

print(a + b)
# [[11 12 13]
#  [21 22 23]]
```

NumPy Array Broadcasting

6. Broadcasting with Higher Dimensions
NumPy compares dimensions **from right to left**. Missing dimensions are assumed to be 1.

```
a = np.array([1, 2, 3])           # shape (3,)
b = np.array([[10], [20], [30]]) # shape (3,1)

# Expand a → shape (1,3), then match b → shape (3,3)
print(a + b)
# [[11 12 13]
#  [21 22 23]
#  [31 32 33]]
```

✅ **Summary of Broadcasting Rules**

1. NumPy compares array shapes from **right to left**.
2. Two dimensions are compatible if they are **equal** or one is **1**.
3. If one array has fewer dimensions, prepend 1s to its shape.
4. If, after alignment, any dimension mismatch remains, broadcasting fails.

NumPy Array Slicing

The Array slicing in **NumPy** is a way to extract a **portion (subset) of an array** by specifying start, stop, and step positions along one or more dimensions.

basics: start: stop: step

- Slices use **half-open intervals**: start inclusive, stop exclusive.
- Any part can be omitted; defaults are start=0, stop=len, step=1.
- Negative indices count from the end; negative step reverses direction.
- Slicing returns a **view** (no data copy) when possible.

Go to short_numpy_array_slicing_guide_exercises.ipynb

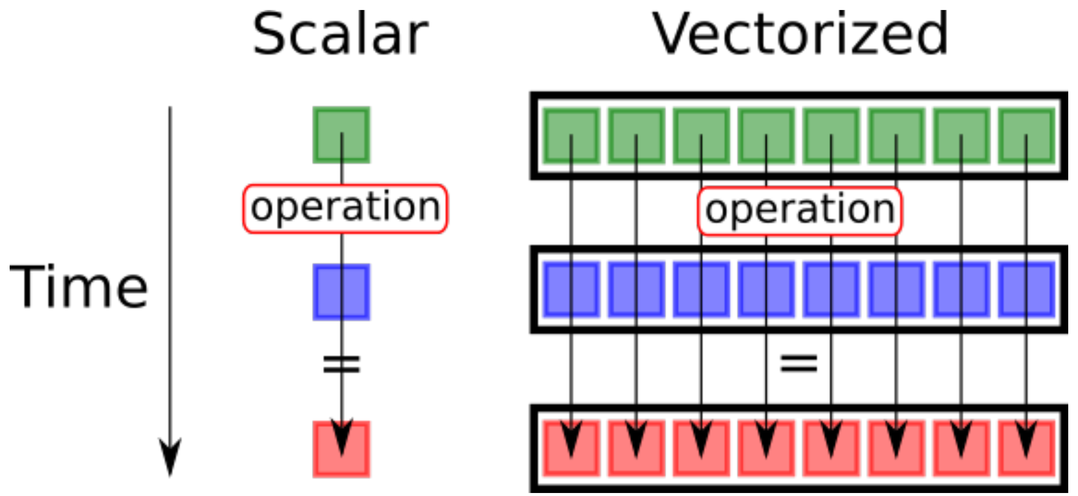
non-vectorization vs Vectorization in Python/NumPy

Non-vectorized approach:

- You write **explicit loops** in Python (e.g., for loops) to process elements one by one.
- This is straightforward but **slow**, because each iteration runs in Python's interpreter, which has a lot of overhead.

Vectorized approach:

- You use **NumPy**, **Pandas**, or similar libraries that implement operations in optimized C/Fortran code under the hood.
- The operation applies to the **whole array at once**, avoiding explicit Python loops.
- Much **faster** and more concise.



non-vectorization vs Vectorization in Python/NumPy

```python
# (a) Means without vectorization
def means_without_vectorization(X: List[List[float]]) ->
    means = []
    for row in X:
        row_sum = 0
        for element in row:
            row_sum += element
        means.append(row_sum / len(row))
    return means
```

`1.np_array = np.array(input_array)`: It first converts the input list of lists (`input_array`) into a NumPy array. This is crucial for utilizing NumPy's vectorized functions.

`2.means = np.mean(np_array, axis=1)`: This is the core of the vectorization. `np.mean()` is a NumPy function that calculates the mean of array elements. The `axis=1` argument specifies that the mean should be computed along the rows (axis 0 is for columns).

`3.return means`: The function returns the resulting NumPy array containing the mean of each row.

```python
# (b) Means with vectorization
def means_with_vectorization(input_array: List[List]) -> np.ndarray:
    # Convert the input list of lists to a NumPy array
    np_array = np.array(input_array)
    # Compute the mean over the rows (axis=1)
    means = np.mean(np_array, axis=1)
    return means
```

When you perform an operation with axis=1, **you are telling NumPy to perform the operation *along* the columns**

non-vectorization vs Vectorization in Python/NumPy

$$A + B = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \cdots & a_{1n} + b_{1n} \\ a_{21} + b_{21} & a_{22} + b_{22} & \cdots & a_{2n} + b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & a_{m2} + b_{m2} & \cdots & a_{mn} + b_{mn} \end{bmatrix}.$$

```python
def element_wise_addition_without_vectorization(A: List[List], B:  List[List]) -> List[List]:
    result = []
    for i in range(len(A)):
        row = []
        for j in range(len(A[0])):
            row.append(A[i][j] + B[i][j])
        result.append(row)
    return result
```

```python
def element_wise_addition_with_vectorization(A: List[List], B:  List[List]) -> np.ndarray:
    # Convert the input lists of lists to NumPy arrays
    np_A = np.array(A)
    np_B = np.array(B)
    # Perform element-wise addition using NumPy
    result = np_A + np_B
    return result
```

Key Linear Algebra Operations with NumPy

Matrix operations

- np.dot() — Dot product
- np.matmul() — Matrix multiplication
- np.transpose() — Transpose of an array
- np.reshape() — Reshape an array

Group Challenge: Mean Squared Error

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2.$$

- Write a function using NumPy to implement mean squared error between two arrays, each of size n
- Use the np.square function and the .mean() method.
- Break down the problem into calculating the squared error first, then find the mean of that squared error.

Python Code Examples

- Introduction to NumPy: https://colab.research.google.com/drive/1F0SNYtuqqcJHzrjYXX62KGGoc-v-VePl#scrollTo=H5Gk682T7FQI
- Mean Squared Error Details: https://colab.research.google.com/drive/1nzF3azRRvtFONjigEQ1O2qzr5AGCMX_U#scrollTo=Cq9_xFg9zgIP
- DS Lab 2: Vectorization Hints: https://colab.research.google.com/drive/1m3T5CZOsrUdI5b4rnDDm4vekRrTL_hmg