Applied Numerical Methods for Civil Engineering

CGN 3405 - 0002

Week 3: Introduction to Python Programming: Part I

Xinyu Chen

Assistant Professor

University of Central Florida

How to understand

Applied Numerical Methods for Civil Engineering?

Numerical methods are techniques by which **mathematical problems** are formulated so that they can be solved with **arithmetic operations**.

Programming Environment

- **No prior programming experience required!**
- Setting up your **environment**
  - Free, no installation
  - Cloud-based Jupyter notebooks
  - Access anywhere with browser
  - Link: `https://colab.research.google.com`

Programming Environment

- **No prior programming experience required!**
- Setting up your **environment**
  - Free, no installation
  - Cloud-based Jupyter notebooks
  - Access anywhere with browser
  - Link: https://colab.research.google.com
- Try it now!

```python
print('Hello Civil Engineering!')
print('Welcome to Applied Numerical Methods')
```

- What is `print()`?
  - A **function** that displays text
  - Anything in quotes is text (string)

Quizzes Now!

- **Today's participation** (ungraded survey): Please check out

  "`Class Participation Quiz 5`"

  Time slot: **2:30PM – 3:00PM**

  on Canvas.
- Online engagement (graded quizzes)

  "`Quiz 5`" (11 questions)

  Deadline: **11:59PM, January 26, 2026**

  on Canvas.

Variables: Storing Data

Variables are containers for data

```python
# Assign values to variables
length = 10.5       # meters
width = 5.2         # meters
material = 'Steel'
```

Rules for variable names:

1. Start with letter or underscore
2. Can contain letters, numbers, underscores
3. Case-sensitive: `Length` $\neq$ `length`
4. Descriptive names recommended
5. Avoid Python keywords, e.g., `lambda`, `class`, `list`, `def`, etc.

Examples:

```python
length = 4
Length = 4.5
print('length = {}'.format(length))
print('Length = {}'.format(Length))
```

Basic Data Types

Four essential types

- Integers: Whole numbers $\ldots, -2, -1, 0, 1, 2, \ldots$

```
length = 4
```

- Floats: Decimal numbers

```
deflection = 0.025  # meters
```

- Strings: Text

```
material = 'Steel'
```

- Booleans: True/False

```
a = True
if a is True:
    print(1)
else:
    print(0)
```

Checking Data Types

- Use `type()` function:

```python
# Check types
length = 4
print(type(length))          # <class 'int'>

deflection = 0.025
print(type(deflection))      # <class 'float'>

material = 'Steel'
print(type(material))        # <class 'str'>

safe = True
print(type(safe))            # <class 'bool'>
```

- Why check types?
  - Different operations work with different types
  - Avoid errors like adding string to number
  - Understand what your code is doing

Basic Arithmetic Operations

Python programming example.

```
1 a = 2
2 b = 3
3 print(a + b)  # plus
4 print(a - b)  # minus
5 print(a * b)  # product
6 print(a / b)  # division
7 print(a ** 2) # quadratic function
8 print(a ** 3) # cubic function
```

Corresponding **arithmetic operations**:

Line **3**: $a + b$

Line **4**: $a - b$

Line **5**: $a \cdot b$

Line **6**: $\frac{a}{b}$

Line **7**: a^2

Line **8**: a^3

Note: `a ** n` refers to a to the power of n, or a^n (n is not only an integer).

Basic Arithmetic Operations

Engineering example.

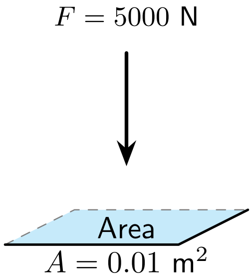
- Definition of normal stress:

$$\sigma = \frac{F}{A}$$

  where

  - $F = 5000$ N (force)
  - $A = 0.01$ m^2 (area)

```python
force = 5000   # N
area = 0.01    # m^2
stress = force / area   # Pa
print('stress = {}'.format(stress))
```



Order of Operations

Python follows PEMDAS:

1. Parentheses ()
2. Exponents
3. Multiplication
4. Division
5. Addition
6. Subtraction

```
# Different results!
a1 = 10 + 5 * 2       # (5*2 first)
a2 = (10 + 5) * 2     # (parentheses first)
```

$$a_1 = 10 + 5 \times 2 \qquad a_2 = (10 + 5) \times 2$$

Order of Operations

Python follows PEMDAS:

1. Parentheses
2. Exponents
3. Multiplication
4. Division
5. Addition
6. Subtraction

Which one is correct?

$$c = \frac{w}{24 \cdot E \cdot I}$$

```python
w = 10 ** 4              # uniform load
E = 2 * 10 ** 11         # modulus
I = 3.25 * 10 ** (-4) # moment of inertia
c1 = w / 24 * E * I
c2 = w / (24 * E * I)
```

Lists: Storing Multiple Values

- Lists store collections of data

```python
# List of beam deflections (mm)
deflections = [12.3, 15.7, 18.2, 14.9, 16.5]
print(deflections)  # [12.3, 15.7, 18.2, 14.9, 16.5]

# List of materials
materials = ['Steel', 'Concrete', 'Timber', 'Aluminum']

# Access elements (0-indexed!)
print(deflections[0])   # First: 12.3
print(deflections[-1])  # Last: 16.5
```

Lists: Storing Multiple Values

- Lists store collections of data

```python
# List of beam deflections (mm)
deflections = [12.3, 15.7, 18.2, 14.9, 16.5]
print(deflections)  # [12.3, 15.7, 18.2, 14.9, 16.5]

# List of materials
materials = ['Steel', 'Concrete', 'Timber', 'Aluminum']

# Access elements (0-indexed!)
print(deflections[0])   # First: 12.3
print(deflections[-1])  # Last: 16.5
```

- List operations for engineering data

```python
print(len(deflections))       # Number of deflections
print(min(deflections))       # Minimum deflection
print(max(deflections))       # Maximum deflection
print(sum(deflections))       # Total
print(sum(deflections)/len(deflections))  # Average
```

Conditionals (if/elif/else)

- Make decisions in code:

```python
stress = 235  # MPa

if stress > 250:
    print('WARNING: Stress exceeds yield strength!')
elif stress > 200:
    print('Alert: Stress approaching limit')
else:
    print('Stress within safe limits')
```

Conditionals (`if/elif/else`)

- Make decisions in code:

```python
stress = 235  # MPa

if stress > 250:
    print('WARNING: Stress exceeds yield strength!')
elif stress > 200:
    print('Alert: Stress approaching limit')
else:
    print('Stress within safe limits')
```

- Comparison operators:
  - `>` greater than
  - `<` less than
  - `>=` greater or equal
  - `<=` less or equal
  - `==` equal to
  - `!=` not equal to

Logical Operators (and/or/not)

- Use the logical operator and:

```
stress = 235

if stress <= 250 and stress > 200:
    print('Alert!')
else:
    print('Others')
```

- Use the logical operator or:

```
stress = 235

if stress > 250 or stress > 200:
    print('At least alert!')
else:
    print('Safe!')
```

for Loop: Repeating Tasks

- Process each item in a sequence:

```python
# List of beam deflections
deflections = [12.3, 15.7, 18.2, 14.9, 16.5]  # mm

# Check each beam
for d in deflections:
    if d > 15:
        print('Deflection exceeds limit')
    else:
        print('Deflection is OK')
```

- Common pattern: Process each item in experimental data

range() Function for Numerical Loops

- Generate sequences of numbers:

```python
# Count from 0 to 4
for i in range(5):
    print(i)

# With start and end
for i in range(2, 6):
    print(i)

# With step
for i in range(0, 10, 2):
    print(i)
```

`range()` Function for Numerical Loops

- Generate sequences of numbers:

```python
1  # Count from 0 to 4
2  for i in range(5):
3      print(i)
4
5  # With start and end
6  for i in range(2, 6):
7      print(i)
8
9  # With step
10 for i in range(0, 10, 2):
11     print(i)
```

Line **2-3** Result: 0, 1, 2, 3, 4

Line **6-7** Result: 2, 3, 4, 5

Line **10-11** Result: 0, 2, 4, 6, 8

while Loop: Repeat Until Condition

- Repeat while condition is true:

```python
a = [1, 2, 3, 4, 5, 6, 7, 8]
i = 0
while a[i] < 6:
    print(a[i])
    i = i + 1
```

Result: 0, 1, 2, 3, 4, 5

Functions: Reusable Code Blocks

- **Quadratic formula**. Given $ax^2 + bx + c = 0$ $(a \neq 0)$, the quadratic formula is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
1 import numpy as np
2 
3 def quad_formula(a, b, c):
4     t = np.sqrt(b**2 - 4*a*c)
5     x1 = (-b + t) / (2*a)
6     x2 = (-b - t) / (2*a)
7     return x1, x2
```

Line **4** Compute $t = \sqrt{b^2 - 4ac}$

Line **5** Compute $x_1 = \dfrac{-b + t}{2a}$

Line **6** Compute $x_2 = \dfrac{-b - t}{2a}$

Functions: Reusable Code Blocks

- Given **parameters**: uniform load $w = 1 \times 10^4$ kg/m, modulus $E = 2 \times 10^{11}$ Pa, and moment of inertia $I = 3.25 \times 10^{-4}$ m^4.
- Compute the **constant factor**:

$$c = \frac{w}{24 \cdot E \cdot I} = \frac{10^4}{24 \times (2 \times 10^{11}) \times (3.25 \times 10^{-4})} = 6.41 \times 10^{-6}$$

```python
import numpy as np

def const(w, E, I):
    return w / (24 * E * I)

w = 10 ** 4             # uniform load
E = 2 * 10 ** 11        # modulus
I = 3.25 * 10 ** (-4) # moment of inertia
c = const(w, E, I)      # constant factor
print(c)
```

Quick Summary

Monday's Class:

- Python environment (no installation with Colab)
- Introduction to Python: Variables, data types (integer, float, string, and Boolean).
- Arithmetic operations, order of operations.
- Storing multiple values with lists
- Logical operators (`for` and `while`)
- Defining functions by yourself

Assignment 1

- **Correction: Question 1b**.

  Euler's Method for a Simple ODE (Numerical Computing).

$$\frac{\mathrm{d}y}{\mathrm{d}x} = x + y, \quad y(0) = 1$$

  The analytical solution is

$$y(x) = 2e^{x} - x - 1$$

  because

$$\frac{\mathrm{d}y}{\mathrm{d}x} = 2e^{x} - 1 = x + (2e^{x} - x - 1) = x + y$$

- **Questions 2b, 3b**. Please use Python programming
  - Bungee jumping velocity model: Time step size $\Delta t = 0.1\,\mathrm{s}$
  - Cantilever beam deflection: Step size $\Delta x = 0.125\,\mathrm{m}$

Exam 1

- Exam Information
  - Date: February 20, 2026
  - Time: 2:30PM – 3:20PM
  - **Written Exam**
  - 15% in your final score
- Format
  - **20 quiz questions** (40 points in total): All selected from the quizzes sessions
  - **Numerical computing tests** ($\approx$ 45 points)
  - **Python programming tests** ($\approx$ 15 points): I will give you Python codes, please write down the results.
- How can I help?
  - Review classes on February 16/18, 2026
- **Maximum Tolerance**: Given the scores of Exam 1 and Exam 2 as a and b, respectively, only in the case of $b > a$, then your score for both exams will become b.

Quizzes Now!

- **Today's participation** (ungraded survey): Please check out

  "`Class Participation Quiz 6`"

  Time slot: **2:30PM – 3:00PM**

  on Canvas.

- Online engagement (graded quizzes)

  "`Quiz 6`" (13 questions)

  Deadline: **11:59PM, January 28, 2026**

  on Canvas.

Learning Objectives

You should be able to:

- Understand the difference among one-, two- and n-dimensional arrays in NumPy
- Understand how to apply some linear algebra operations to n-dimensional arrays without using `for`-loops
- Understand axis and shape properties for n-dimensional arrays

Basics

- Why NumPy for Civil Engineering?
  - Numerical Computing: Solve engineering equations efficiently
  - Matrix Operations: Structural analysis, stiffness matrices
  - Data Processing: Sensor data, experimental results
  - Performance: **50x faster** than Python lists for numerical computing
- What is NumPy?
  - Numerical Python library
  - n-dimensional arrays as core data structure
  - Mathematical functions optimized for arrays

Importing NumPy

- Import convention:

```python
import numpy as np
```

- Why `np`?
    - Standard convention in scientific Python
    - Shorter than typing `numpy` every time
    - Everyone uses this convention

NumPy Arrays vs. Python Lists

- Python Lists

```python
a = [2.2, 3.3, 4.1, 5.2, 6.1]
b = [1.5, 2.1, 3.8, 4.3, 5.2]
c = []
for i in range(5):
    c.append(a[i] * b[i]) # Inefficient!
```

NumPy Arrays vs. Python Lists

- Python Lists

```python
a = [2.2, 3.3, 4.1, 5.2, 6.1]
b = [1.5, 2.1, 3.8, 4.3, 5.2]
c = []
for i in range(5):
    c.append(a[i] * b[i]) # Inefficient!
```

- NumPy Arrays

```python
import numpy as np

a = np.array([2.2, 3.3, 4.1, 5.2, 6.1])
b = np.array([1.5, 2.1, 3.8, 4.3, 5.2])
c = a * b # Fast!
```

- Key Advantage: Vectorization → Faster computation, cleaner code

Algebraic Data $\rightarrow$ NumPy Arrays

- Scalar, e.g., $x = 1$

```python
import numpy as np

x = np.array(1)
```

- Vector, e.g., $\boldsymbol{x} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{bmatrix}$ of length 6

```python
x = np.array([1, 2, 3, 4, 5, 6])
```

- Matrix, e.g., $\boldsymbol{X} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ of 2 rows and 3 columns

```python
X = np.array([[1, 3, 5], [2, 4, 6]])
```

Algebraic Data → NumPy Arrays

- Scalar, e.g., $x = 1$

```python
import numpy as np

x = np.array(1)
```

- Vector, e.g., $\boldsymbol{x} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{bmatrix}$ of length 6

```python
x = np.array([1, 2, 3, 4, 5, 6])
```

- Matrix, e.g., $\boldsymbol{X} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ of 2 rows and 3 columns

```python
X = np.array([[1, 3, 5], [2, 4, 6]])
```

- Data type (integer, float, string, or boolean?)

```python
print(type(X))
```

Algebraic Data → NumPy Arrays

A system of linear equations.

- Let's solve:

$$\begin{cases} 3x + 2y = 5 \\ x - y = 0 \end{cases} \quad \Rightarrow \quad \begin{bmatrix} 3 & 2 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 5 \\ 0 \end{bmatrix}$$

- Try to solve by hand, and then check with Python.
- Define matrix $\boldsymbol{A}$ and vector $\boldsymbol{b}$:

Line **3**: $\boldsymbol{A} = \begin{bmatrix} 3 & 2 \\ 1 & -1 \end{bmatrix}$

Line **4**: $\boldsymbol{b} = \begin{bmatrix} 5 \\ 0 \end{bmatrix}$

```python
1 import numpy as np
2
3 A = np.array([[3, 2], [1, -1]])
4 b = np.array([5, 0])
5 solution = np.linalg.solve(A, b)
6 print('Solution (x, y):', solution)
```

Creating Arrays with Built-In Functions

- Line **3**: Matrix of ones (Fill with ones)

$$\boldsymbol{A} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

- Line **4**: Matrix of zeros (Filling with zeros)

$$\boldsymbol{B} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

- Line **5**: Identify matrix (1 on the diagonal and 0 otherwise)

$$\boldsymbol{C} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```python
1 import numpy as np
2
3 A = np.ones((2, 4))  # (number of rows, number of columns)
4 B = np.zeros((2, 4)) # (number of rows, number of columns)
5 C = np.eye(3)        # number of rows/columns
```

Creating Sequences with `np.arange()`

- `np.arange()`: Like Python's `range()`, but returns array

```
import numpy as np

# Bungee jumping velocity
delta_t = 0.1
t_start = 0
t_end = 20
time_step = np.arange(t_start, t_end, delta_t)
print(time_step)
```

  will not count `t_end = 20`.

- Toy examples:

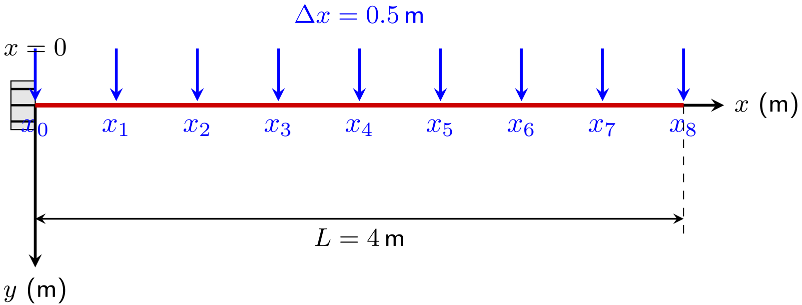
```
import numpy as np

a = np.arange(1, 10, 2) # step size: 2
b = np.arange(1, 10, 2.5) # step size: 2.5
```

$$\boldsymbol{a} = (1, 3, 5, 7, 9)^\top \qquad \boldsymbol{b} = (1, 3.5, 6, 8.5)^\top$$

`np.linspace()`: Specifying Number of Points

Given $\Delta x = 0.5$, the number of steps is $L/\Delta x = 8$.




```python
import numpy as np

# Equally spaced points between 0 and 4
x = np.linspace(0, 4, 5)  # 4 / 1 + 1 = 5
x = np.linspace(0, 4, 9)  # 4 / 0.5 + 1 = 9
```

Basic Operations: Element-Wise Product

- Vectors of the same length, e.g.,

$$\boldsymbol{a} = (20, 30, 40, 50)^\top \qquad \boldsymbol{b} = (0, 1, 2, 3)^\top$$

```python
import numpy as np

a = np.array([20, 30, 40, 50])
b = np.array([0, 1, 2, 3])
# b = np.arange(4)
c = a * b # new array
print(c)
```

Basic Operations: Element-Wise Product

- Vectors of the same length, e.g.,

$$\boldsymbol{a} = (20, 30, 40, 50)^\top \qquad \boldsymbol{b} = (0, 1, 2, 3)^\top$$

```python
import numpy as np

a = np.array([20, 30, 40, 50])
b = np.array([0, 1, 2, 3])
# b = np.arange(4)
c = a * b # new array
print(c)
```

- Matrices of the same size, e.g.,

$$\boldsymbol{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \qquad \boldsymbol{B} = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

```python
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
C = A * B
print(c)
```

Matrix-Vector Multiplication

A system of linear equations.

- Let's solve:

$$\begin{cases} 3x + 2y = 5 \\ x - y = 0 \end{cases} \Rightarrow \begin{bmatrix} 3 & 2 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 5 \\ 0 \end{bmatrix} \Rightarrow \begin{cases} x = 1 \\ y = 1 \end{cases}$$

```python
import numpy as np

A = np.array([[3, 2], [1, -1]])
xy = np.array([1, 1])
b = A @ x # multiplication with the symbol @
print(b)
```

np.random.rand(): Generating Random Values

`np.random.rand()` creates an array of the given shape and populate it with random samples from a **uniform distribution** over `[0, 1)`.

- `np.random.seed()` function is used to initialize the pseudo-random number generator in NumPy
- Generate a **vector**:

```python
import numpy as np
np.random.seed(0)

a = np.random.rand(4)
```

$$\boldsymbol{a} = (0.5488135, 0.71518937, 0.60276338, 0.54488318)^{\top}$$

- Generate a **matrix**:

```python
import numpy as np
np.random.seed(0)

A = np.random.rand(2, 3)
```

$$\boldsymbol{A} = \begin{bmatrix} 0.5488135 & 0.71518937 & 0.60276338 \\ 0.54488318 & 0.4236548 & 0.64589411 \end{bmatrix}$$

`np.reshape()`: Reshaping Arrays

- **Converting matrix into vector**

  Given a matrix $\boldsymbol{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$, there are two strategies:

```python
import numpy as np

A = np.array([[1, 2, 3], [4, 5, 6]])
a1 = np.reshape(A, (6)) # C-like index ordering
print(a1)
a2 = np.reshape(A, (6), order = 'F') # Fortran-like
    index ordering
print(a2)
```

$$\boldsymbol{a}_1 = (1, 2, 3, 4, 5, 6)^\top \qquad \boldsymbol{a}_2 = (1, 4, 2, 5, 3, 6)^\top$$

np.reshape(): Reshaping Arrays

- **Converting vector into matrix**
  How about this?

$$\boldsymbol{a}_1 = (1, 2, 3, 4, 5, 6)^\top$$

```
A1 = np.reshape(a1, (2, 3)) # C-like index ordering
print(A1)
A2 = np.reshape(a1, (2, 3), order = 'F') # Fortran-like
    index ordering
print(A2)
```

$$\boldsymbol{A}_1 = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad \boldsymbol{A}_2 = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Indexing

- Given a vector

```
import numpy as np
np.random.seed(0)

a = np.random.rand(10)
print(a)
```

Result:

```
[0.5488135  0.71518937 0.60276338 0.54488318 0.4236548
     0.64589411 0.43758721 0.891773   0.96366276
    0.38344152]
```

- Indexing

```
i = 1
j = 7
print(a[i])      # 2nd
print(a[j])      # 8th
print(a[i :])    # 2nd to the last
print(a[: j])    # 1st to 7th
print(a[i : j]) # 2nd to 7th
```

Indexing

- Given a matrix

```python
import numpy as np
np.random.seed(0)

A = np.random.rand(7, 5)
print(A)
print(A[2 : 4, 3 : 5])
```

$$\boldsymbol{A} = \begin{bmatrix} 0.5488135 & 0.71518937 & 0.60276338 & 0.54488318 & 0.4236548 \\ 0.64589411 & 0.43758721 & 0.891773 & 0.96366276 & 0.38344152 \\ 0.79172504 & 0.52889492 & 0.56804456 & 0.92559664 & 0.07103606 \\ 0.0871293 & 0.0202184 & 0.83261985 & 0.77815675 & 0.87001215 \\ 0.97861834 & 0.79915856 & 0.46147936 & 0.78052918 & 0.11827443 \\ 0.63992102 & 0.14335329 & 0.94466892 & 0.52184832 & 0.41466194 \\ 0.26455561 & 0.77423369 & 0.45615033 & 0.56843395 & 0.0187898 \end{bmatrix}$$

Quick Summary

Wednesday's Class:

- Difference between NumPy array and Python list
- Writing of algebraic data with NumPy arrays
- Built-in functions, e.g., `np.ones()`, `np.zeros()`, and `np.eye()`
- NumPy sequences with `np.arange()` (set step size)
- NumPy sequences with `np.linspace()` (set the number of steps)
- Basic operations: Element-wise product `*` and matrix-vector multiplication `@` ("at" symbol)
- Random value generation with `np.random.rand()`
- Reshaping arrays (matrix to vector, or vector to matrix): `np.reshape()`
- Indexing

Quizzes Now!

- **Today's participation** (ungraded survey): Please check out

  "`Class Participation Quiz 7`"

  Time slot: **2:30PM – 3:00PM**

  on Canvas.
- Online engagement (graded quizzes)

  "`Quiz 7`"

  Deadline: **11:59PM, January 30, 2026**

  on Canvas.

Indexing

- Given a matrix

```python
import numpy as np
np.random.seed(0)

A = np.random.rand(7, 5)
print(A)
print(A[2, 4]) # 3rd row, 5th column
```

$$\boldsymbol{A} = \begin{bmatrix} 0.5488135 & 0.71518937 & 0.60276338 & 0.54488318 & 0.4236548 \\ 0.64589411 & 0.43758721 & 0.891773 & 0.96366276 & 0.38344152 \\ 0.79172504 & 0.52889492 & 0.56804456 & 0.92559664 & \boxed{0.07103606} \\ 0.0871293 & 0.0202184 & 0.83261985 & 0.77815675 & 0.87001215 \\ 0.97861834 & 0.79915856 & 0.46147936 & 0.78052918 & 0.11827443 \\ 0.63992102 & 0.14335329 & 0.94466892 & 0.52184832 & 0.41466194 \\ 0.26455561 & 0.77423369 & 0.45615033 & 0.56843395 & 0.0187898 \end{bmatrix}$$

Indexing

- Given a matrix

```python
import numpy as np
np.random.seed(0)

A = np.random.rand(7, 5)
print(A)
print(A[:, 4]) # 5th column
```

$$
\boldsymbol{A} = \begin{bmatrix} 0.5488135 & 0.71518937 & 0.60276338 & 0.54488318 & 0.4236548 \\ 0.64589411 & 0.43758721 & 0.891773 & 0.96366276 & 0.38344152 \\ 0.79172504 & 0.52889492 & 0.56804456 & 0.92559664 & 0.07103606 \\ 0.0871293 & 0.0202184 & 0.83261985 & 0.77815675 & 0.87001215 \\ 0.97861834 & 0.79915856 & 0.46147936 & 0.78052918 & 0.11827443 \\ 0.63992102 & 0.14335329 & 0.94466892 & 0.52184832 & 0.41466194 \\ 0.26455561 & 0.77423369 & 0.45615033 & 0.56843395 & 0.0187898 \end{bmatrix}
$$

Indexing

- Given a matrix

```python
import numpy as np
np.random.seed(0)

A = np.random.rand(7, 5)
print(A)
print(A[:, :4]) # 1st to 4th columns
```

$$
\boldsymbol{A} = \begin{bmatrix} 0.5488135 & 0.71518937 & 0.60276338 & 0.54488318 & 0.4236548 \\ 0.64589411 & 0.43758721 & 0.891773 & 0.96366276 & 0.38344152 \\ 0.79172504 & 0.52889492 & 0.56804456 & 0.92559664 & 0.07103606 \\ 0.0871293 & 0.0202184 & 0.83261985 & 0.77815675 & 0.87001215 \\ 0.97861834 & 0.79915856 & 0.46147936 & 0.78052918 & 0.11827443 \\ 0.63992102 & 0.14335329 & 0.94466892 & 0.52184832 & 0.41466194 \\ 0.26455561 & 0.77423369 & 0.45615033 & 0.56843395 & 0.0187898 \end{bmatrix}
$$

np.where(): Performing Conditional Operations

- It acts as a vectorized alternative to standard `if-else` loops
- It allows for data filtering, conditional value replacement, and index retrieval based on specific conditions
- The basic syntax for `np.where()`:

```python
import numpy as np
np.random.seed(0)

a = np.random.rand(5)
print(a)
index = np.where(a > 0.5)
print(index)
```

- Condition: $\boldsymbol{a} > 0.5$

$$\boldsymbol{a} = (0.5488135, 0.71518937, 0.60276338, 0.54488318, 0.4236548)^\top$$

- Output: `print(index)`

```
(array([0, 1, 2, 3]),)
```

np.where(): Performing Conditional Operations

- It acts as a vectorized alternative to standard `if-else` loops
- It allows for data filtering, conditional value replacement, and index retrieval based on specific conditions
- The basic syntax for `np.where()`:

```python
import numpy as np
np.random.seed(0)

a = np.random.rand(5)
print(a)
index = np.where((a > 0.5) & (a < 0.7))
print(index)
```

- Condition: $\boldsymbol{a} > 0.5$ and $\boldsymbol{a} < 0.7$

$$\boldsymbol{a} = (0.5488135, 0.71518937, 0.60276338, 0.54488318, 0.4236548)^\top$$

- Output: `print(index)`

```
(array([0, 2, 3]),)
```

np.where(): Performing Conditional Operations

- The basic syntax for `np.where()`:

```
import numpy as np

A = np.array([[1, 2, 3], [4, 5, 6]])
index = np.where((A > 1) & ( < 5))
print(index)
```

- Condition: $\boldsymbol{A} > 1$ and $\boldsymbol{A} < 5$

$$\boldsymbol{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

- The index set: $\{(1,2),(1,3),(2,1)\}$

np.where(): Performing Conditional Operations

- The basic syntax for `np.where()`:

```python
import numpy as np

A = np.array([[1, 2, 3], [4, 5, 6]])
index = np.where((A > 1) & ( < 5))
print(index)
```

- Condition: $\boldsymbol{A} > 1$ and $\boldsymbol{A} < 5$

$$\boldsymbol{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

- The index set: $\{(1,2),(1,3),(2,1)\}$
- How about the index set in `np.where()`?

`print(index)`

```
(array([0, 0, 1]), array([1, 2, 0]))
```

row index + column index!!!

Transposing Arrays

- Transpose of matrix:

$$\boldsymbol{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad \Rightarrow \quad \boldsymbol{A}^\top = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

```python
import numpy as np

A = np.array([[1, 2, 3], [4, 5, 6]])
print(A.T)
```

- An alternative:

```python
print(A.transpose())
```

`np.flip()`: Flipping Arrays

- Flip or reverse a vector

```python
import numpy as np

a = np.array([1, 2, 3, 4, 5, 6, 7, 8])
# a = np.arange(1, 9)
a_flip = np.flip(a)
print(a_flip)
```

$$\boldsymbol{a} = (1, 2, 3, 4, 5, 6, 7, 8)^\top \quad \Rightarrow \quad \boldsymbol{a}_{\text{flip}} = (8, 7, 6, 5, 4, 3, 2, 1)^\top$$

`np.flip()`: Flipping Arrays

- Flip **only the rows** of a matrix:

$$\boldsymbol{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad \Rightarrow \qquad \boldsymbol{A}_{\text{row}} = \begin{bmatrix} 4 & 5 & 6 \\ 1 & 2 & 3 \end{bmatrix}$$

```python
import numpy as np

A = np.array([[1, 2, 3], [4, 5, 6]])
A_row = np.flip(A, axis = 0)
print(A_row)
```

`np.flip()`: Flipping Arrays

- Flip **only the columns** of a matrix:

$$\boldsymbol{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad \Rightarrow \qquad \boldsymbol{A}_{\text{column}} = \begin{bmatrix} 3 & 2 & 1 \\ 6 & 5 & 4 \end{bmatrix}$$

```python
import numpy as np

A = np.array([[1, 2, 3], [4, 5, 6]])
A_column = np.flip(A, axis = 1)
print(A_column)
```

`np.flip()`: Flipping Arrays

- Flip **rows and columns** of a matrix simultaneously:

$$\boldsymbol{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad \Rightarrow \qquad \boldsymbol{A}_{\text{flip}} = \begin{bmatrix} 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

```python
import numpy as np

A = np.array([[1, 2, 3], [4, 5, 6]])
A_flip = np.flip(A)
print(A_flip)
```

Stacking Together Different Arrays

`np.vstack()`

- Stacking two arrays (same number of columns) **vertically**:

```python
import numpy as np

A = np.array([[1, 2, 3], [4, 5, 6]])
B = np.array([[10, 11, 12], [13, 14, 15], [16, 17,
    18]])
C = np.vstack((A, B))
print(C)
```

- Stacking $\boldsymbol{A}$ and $\boldsymbol{B}$ vertically

$$\boldsymbol{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad \boldsymbol{B} = \begin{bmatrix} 10 & 11 & 12 \\ 13 & 14 & 15 \\ 16 & 17 & 18 \end{bmatrix} \quad \Rightarrow \quad \boldsymbol{C} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 10 & 11 & 12 \\ 13 & 14 & 15 \\ 16 & 17 & 18 \end{bmatrix}$$

with **5 rows** and **3 columns**.

Stacking Together Different Arrays

`np.hstack()`

- Stacking two arrays (same number of rows) **horizontally**:

```python
import numpy as np

A = np.array([[1, 2], [3, 4]])
B = np.array([[10, 11, 12], [13, 14, 15]])
C = np.hstack(A, B)
print(C)
```

- Stacking $\boldsymbol{A}$ and $\boldsymbol{B}$ horizontally

$$\boldsymbol{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad \boldsymbol{B} = \begin{bmatrix} 10 & 11 & 12 \\ 13 & 14 & 15 \end{bmatrix} \quad \Rightarrow \quad \boldsymbol{C} = \begin{bmatrix} 1 & 2 & 10 & 11 & 12 \\ 3 & 4 & 13 & 14 & 15 \end{bmatrix}$$

with **2 rows** and **5 columns**.

Basic Statistics

Given a vector $(x_1, x_2, \ldots, x_n)^\top$ of length n:

- Sum

$$\sum_{i=1}^{n} x_i$$

```python
import numpy as np

x = np.arange(10)
print(np.sum(x))
```

- Mean

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i$$

```python
print(np.mean(x))
```

- Minimum and maximum values

```python
print(np.min(x))
print(np.max(x))
```

Basic Statistics

Given a vector $(x_1, x_2, \ldots, x_n)^\top$ of length n:

- Mean

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

- Variance

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$

  with σ denoting standard deviation.

- Define variance by yourself in Python:

```
import numpy as np

def variance(x): # use np.mean() and np.sum()
    mu = np.mean(x)
    var = np.sum((x - mu) ** 2) / x.shape[0]
    return var

x = np.arange(10)
print(variance(x))
```

- Compare the result with `np.var(x)`

Saving Arrays

NumPy arrays can be saved to `.csv`:

- `np.savetxt()`

```
import numpy as np
np.random.seed(0)

a = np.random.rand(100)
np.savetxt('sample.csv', a)
```

Saving Arrays

NumPy arrays can be saved to `.csv`:

- `np.savetxt()`

```python
import numpy as np
np.random.seed(0)

a = np.random.rand(100)
np.savetxt('sample.csv', a)
```

Loading arrays:

- `np.loadtxt()`

```python
import numpy as np

b = np.loadtxt('sample.csv')
print(b)
```

- Verify the saved data file:

```python
print(np.abs(a - b))
```

Quick Summary

Friday's Class:

- Indexing
- `np.where()` (perform conditional operations)
- Transpose arrays
- `np.flip()` (flip or reverse arrays)
- `np.vstack()` (stack vertically, same number of columns)
- `np.hstack()` (stack horizontally, same number of rows)
- Basic statistics
- Save and load arrays