

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!

```
1 print('Hello Civil Engineering!')
2 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

```
1 # Assign values to variables
2 length = 10.5      # meters
3 width = 5.2        # meters
4 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:

```
1 length = 4
2 Length = 4.5
3 print('length = {}'.format(length))
4 print('Length = {}'.format(Length))
```

Four essential types

- **Integers:** Whole numbers $\dots, -2, -1, 0, 1, 2, \dots$

```
1 length = 4
```

- **Floats:** Decimal numbers

```
1 deflection = 0.025 # meters
```

- **Strings:** Text

```
1 material = 'Steel'
```

- **Booleans:** True/False

```
1 a = True
2 if a is True:
3     print(1)
4 else:
5     print(0)
```

Checking Data Types

- Use `type()` function:

```
1 # Check types
2 length = 4
3 print(type(length))           # <class 'int'>
4
5 deflection = 0.025
6 print(type(deflection))       # <class 'float'>
7
8 material = 'Steel'
9 print(type(material))         # <class 'str'>
10
11 safe = True
12 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.

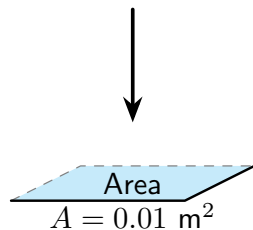
- Definition of normal stress:

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

where

- $F = 5000 \text{ N}$ (force)
- $A = 0.01 \text{ m}^2$ (area)

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



Order of Operations

Python follows PEMDAS:

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

```
1 # Different results!
2 a1 = 10 + 5 * 2      # (5*2 first)
3 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?

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

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

Lists: Storing Multiple Values

- Lists store collections of data

```
1 # List of beam deflections (mm)
2 deflections = [12.3, 15.7, 18.2, 14.9, 16.5]
3 print(deflections) # [12.3, 15.7, 18.2, 14.9, 16.5]
4
5 # List of materials
6 materials = ['Steel', 'Concrete', 'Timber', 'Aluminum']
7
8 # Access elements (0-indexed!)
9 print(deflections[0]) # First: 12.3
10 print(deflections[-1]) # Last: 16.5
```

Lists: Storing Multiple Values

- Lists store collections of data

```
1 # List of beam deflections (mm)
2 deflections = [12.3, 15.7, 18.2, 14.9, 16.5]
3 print(deflections) # [12.3, 15.7, 18.2, 14.9, 16.5]
4
5 # List of materials
6 materials = ['Steel', 'Concrete', 'Timber', 'Aluminum']
7
8 # Access elements (0-indexed!)
9 print(deflections[0]) # First: 12.3
10 print(deflections[-1]) # Last: 16.5
```

- List operations for engineering data

```
1 print(len(deflections))           # Number of deflections
2 print(min(deflections))          # Minimum deflection
3 print(max(deflections))          # Maximum deflection
4 print(sum(deflections))           # Total
5 print(sum(deflections)/len(deflections)) # Average
```

Conditionals (if/elif/else)

- Make decisions in code:

```
1 stress = 235 # MPa
2
3 if stress > 250:
4     print('WARNING: Stress exceeds yield strength!')
5 elif stress > 200:
6     print('Alert: Stress approaching limit')
7 else:
8     print('Stress within safe limits')
```

Conditionals (if/elif/else)

- Make decisions in code:

```
1 stress = 235 # MPa
2
3 if stress > 250:
4     print('WARNING: Stress exceeds yield strength!')
5 elif stress > 200:
6     print('Alert: Stress approaching limit')
7 else:
8     print('Stress within safe limits')
```

- Comparison operators:

- o `>` greater than
- o `<` less than
- o `>=` greater or equal
- o `<=` less or equal
- o `==` equal to
- o `!=` not equal to


```
1 stress = 235
2
3 if stress <= 250 and stress > 200:
4     print('Alert!')
5 else:
6     print('Others')
```

```
1 stress = 235
2
3 if stress > 250 or stress > 200:
4     print('At least alert!')
5 else:
6     print('Safe!')
```

- ```
1 # List of beam deflections
2 deflections = [12.3, 15.7, 18.2, 14.9, 16.5] # mm
3
4 # Check each beam
5 for d in deflections:
6 if d > 15:
7 print('Deflection exceeds limit')
8 else:
9 print('Deflection is OK')
```

## range() Function for Numerical Loops

- Generate sequences of numbers:

```
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)
```

## range() Function for Numerical Loops

- Generate sequences of numbers:

```
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:

```
1 a = [1, 2, 3, 4, 5, 6, 7, 8]
2 i = 0
3 while a[i] < 6:
4 print(a[i])
5 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  $x_1 = \frac{-b+t}{2a}$

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

## 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<sup>4</sup>.
- Compute the **constant factor**:

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

```

1 import numpy as np
2
3 def const(w, E, I):
4 return w / (24 * E * I)
5
6 w = 10 ** 4 # uniform load
7 E = 2 * 10 ** 11 # modulus
8 I = 3.25 * 10 ** (-4) # moment of inertia
9 c = const(w, E, I) # constant factor
10 print(c)

```

### 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).

$$\frac{dy}{dx} = x + y, \quad y(0) = 1$$

The analytical solution is

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

because

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

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



## 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:

```
1 import numpy as np
```

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

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

- Key Advantage: **Vectorization** → **Faster computation**, cleaner code



## Algebraic Data → NumPy Arrays

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

```
1 import numpy as np
2
3 x = np.array(1)
```

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

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

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

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

## Algebraic Data → NumPy Arrays

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

```
1 import numpy as np
2
3 x = np.array(1)
```

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

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

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

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

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

```
1 print (type(X))
```

## Algebraic Data → NumPy Arrays

### A system of linear equations.

- Let's solve:

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

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

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

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

```
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)

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

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

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

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

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

```
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

```
1 import numpy as np
2
3 # Bungee jumping velocity
4 delta_t = 0.1
5 t_start = 0
6 t_end = 20
7 time_step = np.arange(t_start, t_end, delta_t)
8 print(time_step)
```

will not count `t_end = 20`.

- Toy examples:

```
1 import numpy as np
2
3 a = np.arange(1, 10, 2) # step size: 2
4 b = np.arange(1, 10, 2.5) # step size: 2.5
```

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



## Basic Operations: Element-Wise Product

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

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

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

## Basic Operations: Element-Wise Product

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

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

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

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

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

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



## Matrix-Vector Multiplication

### A system of linear equations.

- Let's solve:

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

```
1 import numpy as np
2
3 A = np.array([[3, 2], [1, -1]])
4 xy = np.array([1, 1])
5 b = A @ xy # multiplication with the symbol @
6 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**:

```
1 import numpy as np
2 np.random.seed(0)
3
4 a = np.random.rand(4)
```

$$\mathbf{a} = (0.5488135, 0.71518937, 0.60276338, 0.54488318)^T$$

- Generate a **matrix**:

```
1 import numpy as np
2 np.random.seed(0)
3
4 A = np.random.rand(2, 3)
```

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

## `np.reshape()`: Reshaping Arrays

- **Converting matrix into vector**

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

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

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

## `np.reshape()`: Reshaping Arrays

- **Converting vector into matrix**

How about this?

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

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

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

## Indexing

- Given a vector

```
1 import numpy as np
2 np.random.seed(0)
3
4 a = np.random.rand(10)
5 print(a)
```

Result:

```
1 [0.5488135 0.71518937 0.60276338 0.54488318 0.4236548
 0.64589411 0.43758721 0.891773 0.96366276
 0.38344152]
```

- Indexing

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

## Indexing

- Given a matrix

```
1 import numpy as np
2 np.random.seed(0)
3
4 A = np.random.rand(7, 5)
5 print(A)
6 print(A[2 : 4, 3 : 5])
```

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

## 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

## Indexing

- Given a matrix

```
1 import numpy as np
2 np.random.seed(0)
3
4 A = np.random.rand(7, 5)
5 print(A)
6 print(A[2, 4]) # 3rd row, 5th column
```

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



## Indexing

- Given a matrix

```
1 import numpy as np
2 np.random.seed(0)
3
4 A = np.random.rand(7, 5)
5 print(A)
6 print(A[:, 4]) # 5th column
```

|       |            |            |            |            |            |
|-------|------------|------------|------------|------------|------------|
| $A =$ | 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  |

## Indexing

- Given a matrix

```
1 import numpy as np
2 np.random.seed(0)
3
4 A = np.random.rand(7, 5)
5 print(A)
6 print(A[:, : 4]) # 1st to 4th columns
```

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

## 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()`:

```
1 import numpy as np
2 np.random.seed(0)
3
4 a = np.random.rand(5)
5 print(a)
6 index = np.where(a > 0.5)
7 print(index)
```

- Condition: `a > 0.5`

`a = (0.5488135, 0.71518937, 0.60276338, 0.54488318, 0.4236548)T`

- Output: `print(index)`

```
1 (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()`:

```
1 import numpy as np
2 np.random.seed(0)
3
4 a = np.random.rand(5)
5 print(a)
6 index = np.where((a > 0.5) & (a < 0.7))
7 print(index)
```

- Condition: `a > 0.5` and `a < 0.7`

`a = (0.5488135, 0.71518937, 0.60276338, 0.54488318, 0.4236548)ᵀ`

- Output: `print(index)`

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

## `np.where()`: Performing Conditional Operations

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

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

- Condition:  $A > 1$  and  $A < 5$

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

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

## `np.where()`: Performing Conditional Operations

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

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

- Condition:  $A > 1$  and  $A < 5$

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

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

```
print(index)
```

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

row index + column index!!!

## Transposing Arrays

- Transpose of matrix:

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

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

- An alternative:

```
1 print(A.transpose())
```

## np.flip(): Flipping Arrays

- Flip or reverse a vector

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

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



## np.flip(): Flipping Arrays

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

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

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

## np.flip(): Flipping Arrays

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

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

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

## np.flip(): Flipping Arrays

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

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad \Rightarrow \quad A_{\text{flip}} = \begin{bmatrix} 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

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

## Stacking Together Different Arrays

### `np.vstack()`

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

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

- Stacking  $A$  and  $B$  vertically

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

with 5 rows and 3 columns.

## Stacking Together Different Arrays

`np.hstack()`

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

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

- Stacking  $A$  and  $B$  horizontally

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

with 2 rows and 5 columns.

## Basic Statistics

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

- Sum

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

```
1 import numpy as np
2
3 x = np.arange(10)
4 print(np.sum(x))
```

- Mean

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

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

- Minimum and maximum values

```
1 print(np.min(x))
2 print(np.max(x))
```

## Basic Statistics

cov, std, var

TBD



## Saving and Loading Arrays

How to save and load NumPy objects?

The deflection of cantilever beam

.npy, .csv, .txt