

Applied Numerical Methods for Civil Engineering

CGN 3405 - 0002

Week 4: Introduction to Python Programming: Part II

Xinyu Chen

Assistant Professor

University of Central Florida

Quizzes Now!

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

"Class Participation Quiz 8"

Time slot: **2:30PM – 3:00PM**

on Canvas.

Python Functions

Why use functions?

- **Reusability:** Write once, use many times
- **Modularity:** Break code into manageable blocks
- **Abstraction:** Hide complexity behind simple interfaces
- **Testing & Debugging:** Isolate and test individual components

Basic Function Syntax

```
1 def function_name(parameters):  
2     Optional docstring  
3     # Function body  
4     return value # Optional
```

Basic Function Syntax

Engineering example.

- Definition of normal stress:

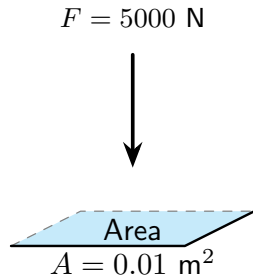
$$\sigma = \frac{F}{A}$$

```
1 def normal_stress(F, A):  
2     return 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 = normal_stress(force, area)  
4 print('stress = {}'.format(stress))
```



Lambda Functions

Quick, one-line functions:

- Example: Quadratic function

$$y = x^2$$

```
1 # Syntax: lambda arguments: expression
2 square = lambda x: x**2
3 print(square(5))      # 25
4
5 # Equivalent def function:
6 def square_func(x):
7     return x**2
8 print(square_func(5)) # 25
```

Lambda Functions

Engineering example.

- Definition of normal stress:

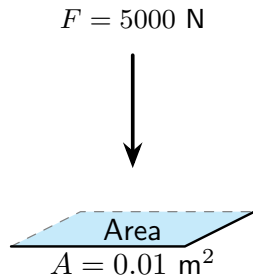
$$\sigma = \frac{F}{A}$$

```
1 stress_lam = lambda F, A: 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 = stress_lam(force, area)
4 print('stress = {}'.format(stress))
```



Lambda Functions

- Example:

$$g(r) = \frac{\pi r^2}{4}$$

```
1 import numpy as np
2
3 g = lambda r: np.pi * r**2 / 4
```

- Evaluate it for $r = 1.5$ and $r = 2.78$

```
1 print(g(1.5))
2 print(g(2.78))
```


Multiple Returns

- 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     term = np.sqrt(b**2 - 4*a*c)
5     x1 = (-b + term) / (2*a)
6     x2 = (-b - term) / (2*a)
7     return x1, x2
```

- Case study: Solve $9x^2 + 3x - 2 = (3x - 1)(3x + 2) = 0$.

```
1 a, b, c = 9, 3, -2
2 x1, x2 = quad_formula(a, b, c)
3 print(x1)
4 print(x2)
```

Recursive Functions

Functions that call themselves

- **Factorial of a non-negative integer** n is the product of all positive integers less than or equal to n :

$$n! = n \times (n - 1) \times (n - 2) \times \cdots \times 3 \times 2 \times 1$$
$$= \begin{cases} 1 & \text{if } n = 1 \\ n \times \underbrace{(n - 1)!}_{\text{factorial}} & \text{if } n > 1 \end{cases}$$

```
1 def factorial(n):  
2     f = 1  
3     for i in range(1, n + 1):  
4         f = f * i  
5     return f
```

- Toy example: $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

```
1 print(factorial(5))
```

Recursive Functions

Functions that call themselves

- **Factorial of a non-negative integer** n is the product of all positive integers less than or equal to n :

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1$$
$$= \begin{cases} 1 & \text{if } n = 1 \\ n \times \underbrace{(n-1)!}_{\text{factorial}} & \text{if } n > 1 \end{cases}$$

```
1 def factorial_r(n):  
2     if n == 0:  
3         return 1  
4     else:  
5         return n * factorial_r(n-1)
```

- Toy example: $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

```
1 print(factorial_r(5))
```

Factorial with NumPy

- Factorial of a non-negative integer n is the product of all positive integers less than or equal to n :

$$n! = n \times (n - 1) \times (n - 2) \times \cdots \times 3 \times 2 \times 1$$

```
1 def factorial_numpy(n):  
2     if n == 0:  
3         return 1  
4     else:  
5         return np.prod(np.arange(1, n+1))
```

- Toy example: $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

```
1 print(factorial_numpy(5))  
2 print(np.prod(np.arange(1, 6)))
```

Factorial with NumPy

- **Factorial of a non-negative integer** n is the product of all positive integers less than or equal to n :

$$n! = n \times (n - 1) \times (n - 2) \times \cdots \times 3 \times 2 \times 1$$

```
1 def factorial_numpy(n):  
2     if n == 0:  
3         return 1  
4     else:  
5         return np.prod(np.arange(1, n+1))
```

- Toy example: $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

```
1 print(factorial_numpy(5))  
2 print(np.prod(np.arange(1, 6)))
```

- Any other built-in function?

```
1 import math  
2  
3 print(math.factorial(5))
```

Approximation for Sine Function

Taylor series expansion for $\sin(x)$:

- Formula

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \frac{x^{13}}{13!} - \frac{x^{15}}{15!} + \dots$$

- Denominator is factorial of odd numbers
- More terms = better approximation

Approximation for Sine Function

Taylor series expansion for $\sin(x)$:

- Formula

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \frac{x^{13}}{13!} - \frac{x^{15}}{15!} + \dots$$

- Denominator is factorial of odd numbers
- More terms = better approximation
- Python programming:

$$\begin{aligned}\sin(x) &= \underbrace{\sum_{n=1}^{+\infty} (-1)^{n-1} \frac{x^{2n-1}}{(2n-1)!}}_{n \text{ starts from } 1} \\ &= \underbrace{\sum_{n=0}^{+\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}}_{n \text{ starts from } 0 \text{ (Python!)}\end{aligned}$$

Approximation for Sine Function

- Python programming:

$$\sin(x) = \underbrace{\sum_{n=0}^{+\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}}_{n \text{ starts from } 0 \text{ (Python!)}$$

```
1 import numpy as np
2
3 def sin_taylor(x, num_term):
4     result = 0
5     for n in range(num_term):
6         # Term index: 0, 1, 2, ... corresponds to x^1,
7           x^3, x^5, ...
8         exp = 2*n + 1
9         factorial = np.prod(np.arange(1, exp + 1))
10        result += ((-1) ** n) * (x ** exp) / factorial
11    return result
```


Approximation for Sine Function

Test case: $\sin(0.9)$

- Ground-truth value:

```
1 print(np.sin(0.9))           # 0.7833269096274834
```

- 1 term:

```
1 print(sin_taylor(0.9, 1))    # 0.9
```

Approximation for Sine Function

Test case: `sin(0.9)`

- Ground-truth value:

```
1 print(np.sin(0.9))           # 0.7833269096274834
```

- 1 term:

```
1 print(sin_taylor(0.9, 1)) # 0.9
```

- 2 terms:

```
1 print(sin_taylor(0.9, 2)) # 0.7785
```

- 3 terms:

```
1 print(sin_taylor(0.9, 3)) # 0.78342075
```

- 4 terms:

```
1 print(sin_taylor(0.9, 4)) # 0.7833258498214286
```

- 5 terms:

```
1 print(sin_taylor(0.9, 5)) # 0.7833269174484375
```

Quick Summary

Monday's Class:

- Basic function syntax
- Lambda function
- Multiple returns
- Recursive functions
- Two examples: Factorial and Taylor series expansion for $\sin(x)$

Norms

What are “**norms**” in mathematics?

- Mathematical rulers for measuring vector and matrix properties
- Distance measures in multi-dimensional space
- Essential tools for **error analysis, optimization, stability**

Why civil engineers needs “**norms**”?

- Error quantification in numerical solutions
- Convergence checking in iterative methods
- Optimization criteria (least squares)
- Stability analysis of structures

Norms

Some important norms:

- ℓ_1 -norm
- ℓ_2 -norm (vector) vs. Frobenius norm (matrix)
- ℓ_∞ -norm

ℓ_1 -Norm

The ℓ_1 -norm measures the total absolute value.

- Mathematical expression:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^n |x_i|$$

for any vector

$$\mathbf{x} = (x_1, x_2, \dots, x_n)^\top$$

- Example:

$$\mathbf{a} = (1, 2, 3, 4)^\top \Rightarrow \|\mathbf{a}\|_1 = 10$$

```
1 import numpy as np
2
3 ell_1 = lambda x: np.sum(np.abs(x))
4 a = np.arange(1, 5)
5 print(a)
6 print(ell_1(a))
```

- How to use NumPy?

```
1 print(np.linalg.norm(a, 1))
```

ℓ_1 -Norm

The ℓ_1 -norm is also called Manhattan norm.

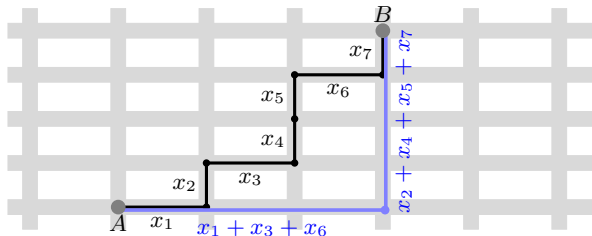
- Mathematical expression:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^n |x_i|$$

for any vector

$$\mathbf{x} = (x_1, x_2, \dots, x_n)^\top$$

- “Walking along city blocks” - only horizontal/vertical moves



ℓ_1 -Norm

- Physical meaning in engineering:
 - Total absolute error across all measurements
 - Resource consumption (total material used)
 - Cost summation across multiple components
- Error analysis: **Mean Absolute Error (MAE)** such that

$$\text{MAE} = \frac{1}{n} \|\epsilon\|_1 = \frac{1}{n} \sum_{i=1}^n |\epsilon_i| = \frac{1}{n} \sum_{i=1}^n |x_i - \hat{x}_i|$$

with the errors:

$$\epsilon_i = \underbrace{x_i}_{\text{true}} - \underbrace{\hat{x}_i}_{\text{approximate}} \quad i = 1, 2, \dots, n$$

- It represents the “average” absolute deviation in the same units as the data

ℓ_1 -Norm

Example: Deflection

- Step-by-step computations:

```
1 import numpy as np
2
3 # True vs measured deflections (mm)
4 true = np.array([12.3, 15.7, 18.2, 14.9, 16.5])
5 measured = np.array([12.5, 15.3, 18.5, 14.7, 16.8])
6
7 # Absolute errors at each point
8 abs_errors = np.abs(measured - true)
9
10 # L1 norm of error = total absolute error
11 total_abs_error = np.sum(abs_errors)
```

- Using NumPy

```
1 np.linalg.norm(measured - true, 1)
```

ℓ_2 -Norm

- Mathematical expression:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2}$$

for any vector

$$\mathbf{x} = (x_1, x_2, \dots, x_n)^\top$$

- Example:

$$\mathbf{a} = (1, 2, 3, 4)^\top \Rightarrow \|\mathbf{a}\|_2 = \sqrt{1^2 + 2^2 + 3^2 + 4^2} = \sqrt{30}$$

```
1 import numpy as np
2
3 ell_2 = lambda x: np.sqrt(np.sum(x ** 2))
4 a = np.arange(1, 5)
5 print(a)
6 print(ell_2(a))
```

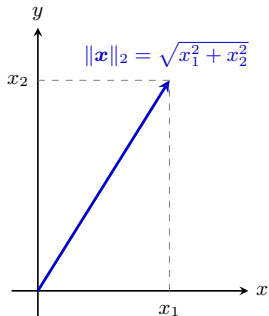
- How to use NumPy?

```
1 print(np.linalg.norm(a, 2))
```

ℓ_2 -Norm

Intuitive understanding?

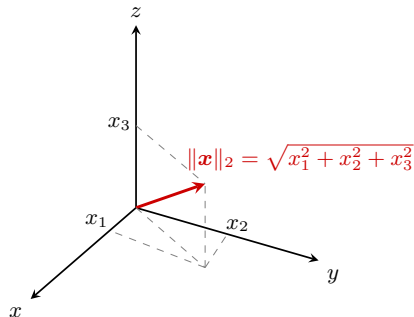
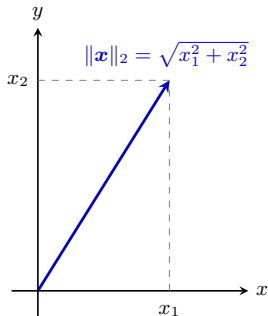
- Vectors $\mathbf{x} = (x_1, x_2)^\top$ vs. $\mathbf{x} = (x_1, x_2, x_3)^\top$



ℓ_2 -Norm

Intuitive understanding?

- Vectors $\mathbf{x} = (x_1, x_2)^\top$ vs. $\mathbf{x} = (x_1, x_2, x_3)^\top$



Frobenius Norm

- ℓ_2 -norm:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2}$$

for any **vector**

$$\mathbf{x} = (x_1, x_2, \dots, x_n)^\top$$

- **Frobenius norm:**

$$\|\mathbf{X}\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n x_{ij}^2}$$

for any **matrix**

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix} \quad m \text{ rows \& } n \text{ columns}$$

Frobenius Norm

- Example:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \Rightarrow \|\mathbf{A}\|_F = \sqrt{1^2 + 2^2 + 3^2 + 4^2} = \sqrt{30}$$

```
1 import numpy as np
2
3 frob = lambda X: np.sqrt(np.sum(X ** 2))
4 A = np.array([[1, 2], [3, 4]])
5 print(frob(A))
```

- How to use NumPy?

```
1 print(np.linalg.norm(A, 'f'))
```

ℓ_∞ -Norm

- Mathematical expression of ℓ_∞ -norm (“Worst-case” or “maximum” distance):

$$\|x\|_\infty = \max\{|x_1|, |x_2|, \dots, |x_n|\}$$

- Example:

```
1 import numpy as np
2
3 a = np.array([-1, 2, -3, 4])
4 print(np.linalg.norm(a, np.inf))
```

- Write a function?

```
1 ell_inf = lambda x: np.max(np.abs(x))
2 print(ell_inf(a))
```

- Physical meaning in engineering:
 - Maximum stress in a structure
 - Peak deflection in a beam
 - Worst-case error in measurements
 - Safety factor based on extreme values

Singular Value Decomposition

Graphical demonstration <https://xinychen.github.io/slides/opp.pdf>

Singular Value Decomposition

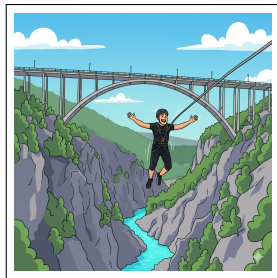
Norms

Norms: Engineering Intuition

How do we quantify "How far are we from the truth?" in Engineering?

The Bungee Analogy:

- ℓ_2 -norm (Distance / Energy)
 - $\|\mathbf{x}\|_2 = \sqrt{\sum x_i^2}$. Is the cord long enough? Measures the "Total Energy" of the error.
- ℓ_∞ -norm (Max Tolerance)
 - $\|\mathbf{x}\|_\infty = \max |x_i|$. Will the jumper hit the canyon wall in *any* single direction?



Choosing the right norm is safety-critical.

Absolute Error Definition

$$E_{abs} = \|\mathbf{x}_{true} - \mathbf{x}_{calc}\|$$

Norms: Error Analysis & Python

- **Why Relative Error matters?** An absolute error of 0.1m is fine for a 1km bridge, but fatal for a 1cm micro-chip.

$$e_{rel} = \frac{\|\mathbf{x}_{true} - \mathbf{x}_{calc}\|}{\|\mathbf{x}_{true}\|}$$

- **Numerical Computation:**

```
1 import numpy as np
2 x_true = np.array([1.0, 2.0, 3.0])
3 x_calc = np.array([1.05, 1.95, 3.10])
4 err_vec = x_true - x_calc
5 # 1. Absolute Error (L2 and Inf)
6 abs_l2 = np.linalg.norm(err_vec, ord=2)
7 abs_inf = np.linalg.norm(err_vec, ord=np.inf)
8 # 2. Relative Error (Standard for Stability)
9 rel_err = abs_l2 / np.linalg.norm(x_true, ord=2)
10 print(f L2 Error: {abs_l2:.4f}, Worst-case: {abs_inf:.4f} )
11 print(f Relative Error: {rel_err:.2%} )
```

- **Takeaway:** In SVD and PCA, the ℓ_2 -norm (or Frobenius for matrices) is the primary tool for measuring information loss.

Matrix Norms: Frobenius Norm

- **Mathematical Definition:** The square root of the sum of absolute squares:

$$\|A\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2} = \sqrt{\text{trace}(A^H A)}$$

```
1 def frobenius_norm(A):  
2     return np.sqrt(np.sum(np.abs(A)**2))
```

- **Example:** $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, $\|A\|_F = \sqrt{1^2 + 2^2 + 3^2 + 4^2} = \sqrt{30} \approx 5.477$

```
1 A = np.array([[1, 2], [3, 4]])  
2 # Using Built-in NumPy Function  
3 f_norm = np.linalg.norm(A, ord='fro')  
4 print(f Frobenius Norm: {f_norm:.4f} )
```

Orthogonal Procrustes Problem (OPP)

- **Mathematical Definition:** Find an orthogonal matrix F that best aligns F to Q :

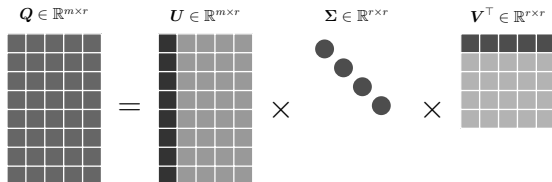
$$\min_F \|F - Q\|_F^2$$

$$\text{s.t. } F^\top F = I_r \quad (\text{Orthogonal Constraint})$$

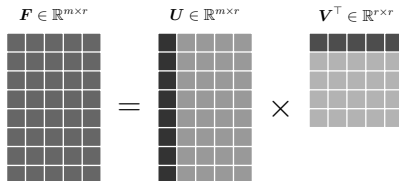
```
1 import numpy as np
2 from numpy.linalg import svd
3
4 def opp(Q):
5     # Step 1: Compute SVD
6     U, Sigma, V_h = svd(Q, full_matrices=False)
7     # Step 2: Return U @ V^T (Note: svd returns V_h as
8     #           V^T)
9     return U @ V_h
```

OPP: Step-by-Step Solution

- **Singular Value Decomposition (SVD):** To find the closest orthogonal matrix, we first decompose the target matrix Q :

$$Q \in \mathbb{R}^{m \times r} = U \in \mathbb{R}^{m \times r} \times \Sigma \in \mathbb{R}^{r \times r} \times V^T \in \mathbb{R}^{r \times r}$$


- **Optimal Reconstruction:** The solution F is formed by the product of U and V^T :

$$F \in \mathbb{R}^{m \times r} = U \in \mathbb{R}^{m \times r} \times V^T \in \mathbb{R}^{r \times r}$$


Application: Point Cloud Registration (1/2)

- **The Task:** Align a "Source" point cloud P to a "Target" point cloud Q by finding the optimal rotation R .
- **Preprocessing (Centering):** Before solving OPP, both point clouds must be centered at the origin to remove translation:

$$P' = P - \bar{p}, \quad Q' = Q - \bar{q}$$

- **Formulating as OPP:** We seek an orthogonal matrix R that minimizes the Mean Squared Error (MSE):

$$\min_R \|RP' - Q'\|_F^2 \quad \text{s.t.} \quad R^\top R = I$$

- **Engineering Intuition:** This is the "Kabsch Algorithm". It ensures that the geometric shape is preserved (no stretching) while maximizing the overlap between two sets of sensor data (e.g., LiDAR scans).

Application: Point Cloud Registration (2/2)

- Python Implementation (The Kabsch Algorithm):

```
1 import numpy as np
2 def register_points(P, Q):
3     # 1. Centering
4     P_centered = P - np.mean(P, axis=0)
5     Q_centered = Q - np.mean(Q, axis=0)
6     # 2. Compute Covariance Matrix H
7     H = P_centered.T @ Q_centered
8     # 3. SVD to find optimal Rotation
9     U, S, Vt = np.linalg.svd(H)
10    R = Vt.T @ U.T
11    # Special case: handle reflection if det(R) < 0
12    if np.linalg.det(R) < 0:
13        Vt[-1, :] *= -1
14        R = Vt.T @ U.T
15    return R
```

Application: Visualization of Alignment

The following diagram illustrates the transformation process using the **Orthogonal Procrustes** solution:



Summary

The **ℓ_2 -norm** minimization in OPP ensures that the average squared distance between all corresponding points is globally minimized, effectively "snapping" the two shapes together.