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

```python
def function_name(parameters):
    """Optional docstring"""
    # Function body
    return value  # Optional
```

Basic Function Syntax

Engineering example.

- Definition of normal stress:

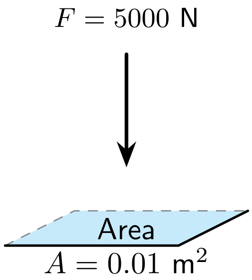
$$\sigma = \frac{F}{A}$$

```python
def normal_stress(F, A):
    return F / A
```

where

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

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



Lambda Functions

Quick, one-line functions:

- Example: Quadratic function

$$y = x^2$$

```python
# Syntax: lambda arguments: expression
square = lambda x: x**2
print(square(5))     # 25

# Equivalent def function:
def square_func(x):
    return x**2
print(square_func(5)) # 25
```

Lambda Functions

Engineering example.

- Definition of normal stress:

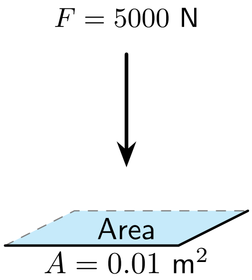
$$\sigma = \frac{F}{A}$$

```python
stress_lam = lambda F, A: F / A
```

where

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

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



Lambda Functions

- Example:

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

```python
import numpy as np

g = lambda r: np.pi * x**2 / 4
```

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

```python
print(g(1.5))
print(g(2.78))
```

Multiple Returns

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

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

```
import numpy as np

def quad_formula(a, b, c):
    term = np.sqrt(b**2 - 4*a*c)
    x1 = (-b + term) / (2*a)
    x2 = (-b - term) / (2*a)
    return x1, x2
```

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

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

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

```python
def factorial(n):
    f = 1
    for i in range(1, n + 1):
        f = f * i
    return f
```

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

```python
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:

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

```python
def factorial_r(n):
    if n == 0:
        return 1
    else:
        return n * factorial_r(n-1)
```

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

```python
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:

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

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

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

```
print(factorial_numpy(5))
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:

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

```python
def factorial_numpy(n):
    if n == 0:
        return 1
    else:
        return np.prod(np.arange(1, n+1))
```

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

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

- Any other built-in function?

```python
import math

print(math.factorial(5))
```

Approximation for Sine Function

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

- Formula

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

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

Approximation for Sine Function

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

- Formula

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

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

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

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

Approximation for Sine Function

- Python programming:

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

```python
import numpy as np

def sin_taylor(x, num_term):
    result = 0
    for n in range(num_term):
        # Term index: 0, 1, 2, ... corresponds to x^1, x^3, x^5, ...
        exp = 2*n + 1
        factorial = np.prod(np.arange(1, exp + 1))
        result += ((-1) ** n) * (x ** exp) / factorial
    return result
```

Approximation for Sine Function

Test case: $\sin(0.9)$

- Ground-truth value:

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

- 1 term:

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

Approximation for Sine Function

Test case: $\sin(0.9)$

- Ground-truth value:

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

- 1 term:

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

- 2 terms:

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

- 3 terms:

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

- 4 terms:

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

- 5 terms:

```python
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 $\sin(x)$

Quizzes Now!

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

  "`Class Participation Quiz 9`"

  Time slot: **2:30PM – 3:00PM**

  on Canvas.

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**, and **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:

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

  for any vector

$$\boldsymbol{x} = (x_1, x_2, \cdots, x_n)^\top$$

ℓ_1-Norm

The ℓ_1-norm measures the total absolute value.

- Mathematical expression:

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

  for any vector

$$\boldsymbol{x} = (x_1, x_2, \cdots, x_n)^\top$$

- Example:

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

```python
import numpy as np

ell_1 = lambda x: np.sum(np.abs(x))
a = np.arange(1, 5)
print(a)
print(ell_1(a))
```

- How to use NumPy?

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

ℓ_1-Norm

The ℓ_1-norm is also called Manhattan norm.

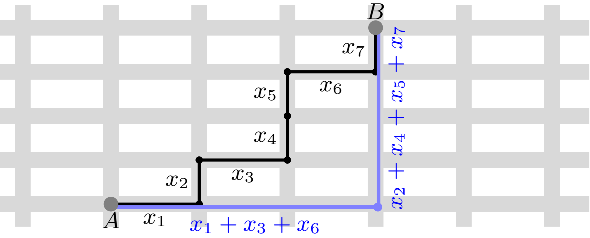
- Mathematical expression:

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

  for any vector

$$\boldsymbol{x} = (x_1, x_2, \cdots, x_n)^\top$$

- "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

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

with the errors:

$$\varepsilon_i = \underbrace{\hat{x}_i}_{\text{approximate}} - \underbrace{x_i}_{\text{true}} \qquad i = 1, 2, \ldots, n$$

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

ℓ_1-Norm

Example: Deflection

- Step-by-step computations:

$$\text{MAE} = \frac{|0.2| + |-0.4| + |0.3| + |-0.2| + |0.3|}{5} \approx 0.28$$

```python
import numpy as np

# True vs measured deflections (mm)
true = np.array([12.3, 15.7, 18.2, 14.9, 16.5])
measured = np.array([12.5, 15.3, 18.5, 14.7, 16.8])

# Absolute errors at each point
abs_errors = np.abs(measured - true)

# L1 norm of error = total absolute error
total_abs_error = np.sum(abs_errors)
```

- Using NumPy

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

ℓ_2-Norm

- Mathematical expression:

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

for any vector

$$\boldsymbol{x} = (x_1, x_2, \cdots, x_n)^\top$$

ℓ_2-Norm

- Mathematical expression:

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

for any vector

$$\boldsymbol{x} = (x_1, x_2, \cdots, x_n)^\top$$

- Example:

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

```
import numpy as np

ell_2 = lambda x: np.sqrt(np.sum(x ** 2))
a = np.arange(1, 5)
print(a)
print(ell_2(a))
```

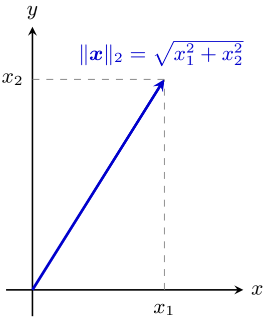
- How to use NumPy?

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

ℓ_2-Norm

Intuitive understanding?

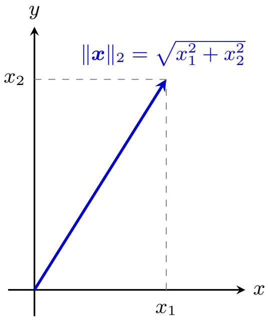
- ℓ_2-norm is the Euclidean distance in space.
- Vectors $\boldsymbol{x} = (x_1, x_2)^\top$ vs. $\boldsymbol{x} = (x_1, x_2, x_3)^\top$

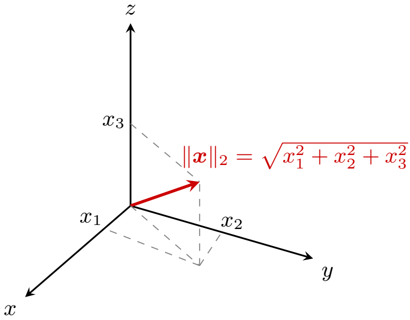


ℓ_2-Norm

Intuitive understanding?

- ℓ_2-norm is the Euclidean distance in space.
- Vectors $\boldsymbol{x} = (x_1, x_2)^\top$ vs. $\boldsymbol{x} = (x_1, x_2, x_3)^\top$





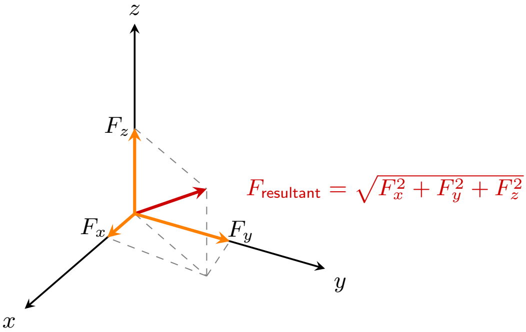
ℓ_2-Norm

- If forces F_x, F_y, F_z act on a joint, resultant force magnitude:

$$F_{\text{resultant}} = \sqrt{F_x^2 + F_y^2 + F_z^2}$$

- Example: $F_x = 3, F_y = 4, F_z = 12$ kN, then

$$F_{\text{resultant}} = \sqrt{3^2 + 4^2 + 12^2} = 13\,\text{kN}$$



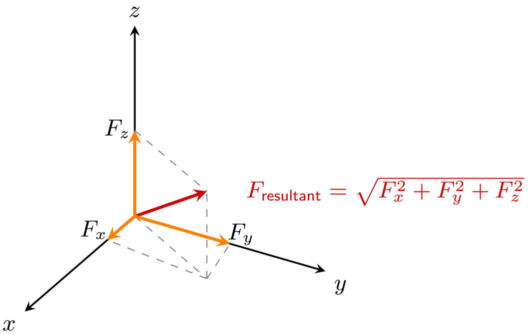
ℓ_2-Norm

- Example: $F_x = 3.5, F_y = 2.1, F_z = 4.8$ kN, then

$$F_{\text{resultant}} = \sqrt{3.5^2 + 2.1^2 + 4.8^2} \approx 6.30\,\text{kN}$$

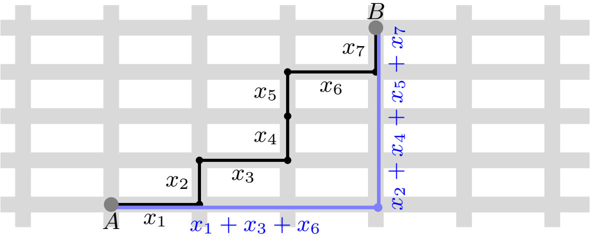
```python
import numpy as np

F = np.array([3.5, 2.1, 4.8])
print(np.linalg.norm(F, 2))
```



ℓ_1-Norm vs. ℓ_2-Norm

In a city grid, walking from $(0, 0)$ to $(3, 4)$:



- ℓ_1 distance $= |3| + |4| = 7$ blocks
- ℓ_2 distance $= \sqrt{3^2 + 4^2} = 5$ blocks (not walkable!)

Frobenius Norm

- **ℓ_2-norm**:

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

for any **vector**

$$\boldsymbol{x} = (x_1, x_2, \cdots, x_n)^\top$$

- **Frobenius norm**:

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

for any **matrix**

$$\boldsymbol{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix}$$

m rows & n columns

Frobenius Norm

- Example:

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

```python
import numpy as np

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

- How to use NumPy?

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

ℓ_∞-Norm

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

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

- Example:

```
import numpy as np

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

- Write a function?

```
ell_inf = lambda x: np.max(np.abs(x))
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

ℓ_∞-Norm

Example: Worst-case prediction error

- Focuses only on the worst-case element - conservative design
- Python codes

```
# Errors in temperature predictions at different
    locations
errors = np.array([-1.2, 0.8, -2.1, 1.5, -0.3, 1.9])

# L_infinity norm = maximum absolute error
max_abs_error = np.max(np.abs(errors))
worst_location = np.argmax(np.abs(errors))
```

- Maximum absolute error: -2.1
- Location: the 3rd value

ℓ_∞-Norm

Example: Safety factor based on extreme values

- In design codes, the **maximum stress** must not exceed allowable stress:

$$\sigma_{\max} = \max\{|\sigma_1|, |\sigma_2|, \ldots\} = \|\boldsymbol{\sigma}\|_\infty$$

- If measured stresses $= [120, -150, 130]$ MPa,

$$\|\boldsymbol{\sigma}\|_\infty = 150\,\text{MPa}$$

  Compare to allowable stress (e.g., 200 MPa) for safety.

Quick Summary

Wednesday's Class:

- ℓ_1-norm: Sum of absolute values $\rightarrow$ total deviation
- ℓ_2-norm: magnitude in space
- ℓ_∞-norm: Maximum absolute value $\rightarrow$ worst-case measure
- Frobenius Norm: For matrices, like ℓ_2-norm for vectors