TÓM TẮT LÝ THUYẾT

NUMPY - PANDAS

Lê Văn Hạnh

2025

MỤC LỤC

1. NUMPY 1
1.1. NumPy array 1
1.1.1. Giới thiệu 1
1.1.2. Các toán tử trên mảng NumPy 1
1.1.3. Tạo Numpy array 4
1.1.4. Truy cập các phần tử của Numpy array 13
1.1.5. Một số thuộc tính của NumPy array 14
1.1.6. Một số phương thức của NumPy array 16
1.1.7. Một số hàm của NumPy sử dụng trên array 17
1.2. numpy.random 22
1.2.1. random() hoặc rand() 22
1.2.2. randint() 22
1.2.3. choice() 22
1.2.4. normal() 22
1.2.5. uniform() 23
1.2.6. shuffle() 23
1.2.7. seed() 23
1.2.8. binomial() 24
1.2.9. permutation() 24

2. PANDAS 25
2.1. GIỚI THIỆU 25
2.2. MỘT SỐ THUỘC TÍNH THƯỜNG DÙNG TRONG PANDAS 25
2.2.1. iloc &loc 25
2.3. MỘT SỐ HÀM THƯỜNG DÙNG TRONG PANDAS 27
2.3.1. pandas.concat() 27
2.3.2. pandas.date_range() 28
2.3.3. pandas.to_datetime() 29

3. PANDAS DATA SERIES 31
3.1. Giới thiệu 31
3.2. Tạo series 32
3.2.1. Từ list 32
3.2.2. Từ NumPy array 32
3.2.3. Từ Dictionary 32
3.2.4. Từ một giá trị vô hướng (Scalar value) 32
3.3. Truy xuất phần tử dựa trên index 33
3.4. Chuyển đổi từ series sang đối tượng khác 33
3.5. Các phép toán dựa trên vector (Vectorized) 34
3.6. Thuộc tính thường dùng 35
3.7. Phương thức thường dùng 35
3.7.1. apply() 35
3.7.2. argmax() / argmin() hoặc idmax() / idmin() 36
3.7.3. bfill() hoặc ffill() 38
3.7.4. describe() 38
3.7.5. dropna() 39

3.7.6. dt() 41
3.7.7. explode() 44
3.7.8. fillna() 45
3.7.9. head(n) 48
3.7.10. isnull() hoặc isna 48
3.7.11. items() 49
3.7.12. max() 50
3.7.13. map() 50
3.7.14. mean() 50
3.7.15. min() 50
3.7.16. notnull() hay notna() 51
3.7.17. reset_index() 52
3.7.18. sort_index() 53
3.7.19. std() 56
3.7.20. str() 56
3.7.21. sum() 58
3.7.22. tail(n) 58
3.7.23. to_numpy() 58
3.7.24. tolist() 59
3.7.25. unique() 61
3.7.26. value_counts() 61

4. PANDAS DATAFRAME 64
4.1. Giới thiệu 64
4.2. Các đặc điểm chính của một DataFrame: 64
4.3. Cấu trúc bên trong (một cách trừu tượng): 65
4.4. Cách tạo một DataFrame 65
4.4.1. Từ một Dictionary của các list hoặc NumPy array 65
4.4.2. Từ một list của các Dictionary 65
4.4.3. Từ một NumPy array hai chiều 66
4.4.4. Từ một đối tượng pandas.Series 66
4.4.5. Từ việc đọc file 66
4.4.6. Một số thao tác cơ bản với DataFrame 67
4.5. Công dụng phổ biến của DataFrame 68
4.6. Một số thuộc tính thường dùng 68
4.6.1. Values 68
4.6.2. shape 70
4.7. Một số phương thức thường dùng 70
4.7.1. at() 70
4.7.2. head() 72
4.7.3. describe() 72
4.7.4. drop() 72
4.7.5. dropna() 75
4.7.6. fillna() 76
4.7.7. groupby() 77
4.7.8. iat() 80
4.7.9. iterrows() 82
4.7.10. rank() 85
4.7.11. rename() 86

4.7.12. sample() 89
4.8. Duyệt DataFrame theo hàng 91
4.8.1. Lưu ý về hiệu suất và mục đích của việc duyệt DataFrame theo hàng 91
4.8.2. Các cách duyệt DataFrame theo hàng 91
4.8.3. Cách 2: Sử dụng DataFrame.itertuples() 92
4.9. Duyệt DataFrame theo cột 94
4.9.1. Cách 1: Duyệt trực tiếp qua df.columns 94
4.9.2. Cách 2: Sử dụng vòng lặp for với range và chỉ số 95
4.9.3. Cách 3: Chuyển tên cột thành danh sách rồi duyệt 95
4.9.4. Cách 4: Sử dụng df.items() (Duyệt qua (tên cột, Series dữ liệu cột)) 95
5. MỘT SỐ THƯ VIỆN KHÁC TRONG PYTHON 97
5.1. dateutil 97
5.1.1. Hàm dateutil.parser.parse 97
LINK THAM KHẢO 98

1. NUMPY

1.1. NumPy array

1.1.1. Giới thiệu

1.1.1.1. Mảng NumPy là gì?

- Một mảng `NumPy` (*numpy array*) là một lưới các giá trị, thường là các số, tất cả đều có cùng một kiểu dữ liệu (ví dụ: số nguyên, số thực, số phức).
- Mảng `Numpy` khác với `list` của Python ở chỗ các mảng `NumPy` có kích thước cố định và các phần tử có cùng kiểu dữ liệu, giúp cho các phép toán số trên chúng hiệu quả hơn nhiều.

1.1.1.2. Tại sao lại sử dụng mảng NumPy?

- *Hiệu quả bộ nhớ*: Mảng `NumPy` được lưu trữ trong bộ nhớ liền kề, giúp truy cập và thao tác dữ liệu nhanh hơn nhiều so với list của Python.
- *Tốc độ tính toán*: `NumPy` được tối ưu hóa cho các phép toán số học và toán học trên các mảng, nhanh hơn nhiều so với việc thực hiện các phép toán tương tự bằng Python thuần túy.
- *Các hàm và công cụ mạnh mẽ*: `NumPy` cung cấp một lượng lớn các hàm để thao tác, biến đổi và tính toán trên các mảng, bao gồm các hàm đại số tuyến tính, thống kê và biến đổi Fourier.
- *Tiện lợi*: Mảng `NumPy` cung cấp một cách nhất quán và dễ sử dụng để làm việc với dữ liệu số đa chiều.

1.1.1.3. Các trường hợp sử dụng phổ biến của mảng NumPy:

- *Tính toán số học:* Thực hiện các phép toán trên mảng một cách hiệu quả (cộng, trừ, nhân, chia, v.v.).
- *Đại số tuyến tính:* Thực hiện các phép toán ma trận, tìm nghịch đảo, trị riêng, v.v.
- *Thống kê:* Tính trung bình, độ lệch chuẩn, phương sai, v.v.
- *Xử lý ảnh và âm thanh:* Biểu diễn và thao tác dữ liệu hình ảnh và âm thanh dưới dạng mảng.
- *Khoa học dữ liệu và học máy:* Lưu trữ và xử lý dữ liệu số cho các mô hình học máy.

1.1.2. Các toán tử trên mảng NumPy

Lưu ý: Các mảng phải có cùng hình dạng hoặc có thể được mở rộng để có cùng hình dạng (*broadcasting*).

1.1.2.1. Toán tử số học

Các toán tử số học được sử dụng để thực hiện các phép toán số học trên các phần tử của mảng.

Toán tử	*Mô tả*	*Ví dụ*	*Hàm Numpy tương đương*
`+`	Cộng	`a + b`	`np.add(a, b)`
`-`	Trừ	`a - b`	`np.subtract(a, b)`
`* hoặc @`	Nhân	`a * b`	`np.multiply(a, b)`
`/`	Chia	`a / b`	`np.divide(a, b)`
`//`	Chia lấy phần nguyên	`a // b`	`np.floor_divide(a, b)`
`%`	Chia lấy dư	`a % b`	`np.mod(a, b)`
`**`	Lũy thừa	`a ** b`	`np.power(a, b)`
`-`	Đổi dấu	`-a`	

1.1.2.2. Toán tử so sánh

Các toán tử so sánh được sử dụng để so sánh các phần tử của mảng.

Toán tử	*Mô tả*	*Ví dụ*
`==`	Bằng	`a == b`
`!=`	Không bằng	`a != b`
`>`	Lớn hơn	`a > b`
`<`	Nhỏ hơn	`a < b`
`>=`	Lớn hơn hoặc bằng	`a >= b`
`<=`	Nhỏ hơn hoặc bằng	`a <= b`

1.1.2.3. Toán tử logic

Các toán tử logic được sử dụng để thực hiện các phép toán logic trên các phần tử của mảng.

Toán tử	*Mô tả*	*Ví dụ*	*Hàm Numpy tương đương*
`&`	Và	`a & b`	`np.logical_and(a, b)`
`|`	Hoặc	`a | b`	`np.logical_or(a, b)`
`~`	Không	`~a`	`np.logical_not(a)`
`^`	Xor	`a ^ b`	`np.logical_xor(a, b)`

1.1.2.4. Toán tử bitwise

Các toán tử bitwise được sử dụng để thực hiện các phép toán bitwise trên các phần tử của mảng.

Toán tử	*Mô tả*	*Ví dụ*	*Hàm Numpy tương đương*
`&`	AND bitwise	`a & b`	`np.bitwise_and(a, b)`
`|`	OR bitwise	`a | b`	`np.bitwise_or(a, b)`
`^`	XOR bitwise	`a ^ b`	`np.bitwise_xor(a, b)`
`~`	NOT bitwise	`~a`	`np.bitwise_not(a)`
`<<`	Dịch trái	`a << b`	`np.left_shift(a, b)`
`>>`	Dịch phải	`a >> b`	`np.right_shift(a, b)`

1.1.2.6. Ví dụ

- **Mảng 1 chiều**

Mã lệnh	Kết quả

`import numpy as np`
`a = np.array([1, 2, 3, 4])`
`b = np.array([5, 6, 7, 8])`
`print("Array a:", a)`
`print("Array b:", b)`
	`Array a: [1 2 3 4]`
	`Array b: [5 6 7 8]`

`# Cộng` `print("a + b =", a + b)`	`a + b = [ 6  8 10 12]`
`print("np.add(a, b) =", np.add(a, b))`	`np.add(a, b) = [ 6  8 10 12]`
`# Nhân` `print("a * b =", a * b)`	`a * b = [ 5 12 21 32]`
`print("np.multiply(a, b) =", np.multiply(a, b))`	`np.multiply(a, b) = [5 12 21 32]`
`# So sánh` `print("a > 2 =", a > 2)`	`a > 2 = [False False  True True]`
`print("a == b", a == b)`	`a == b [False False False False]`
`# Logic` `print("(a > 2) & (b < 8) =", (a > 2) & (b < 8))`	`(a > 2) & (b < 8) = [False False  True False]`
`print("np.logical_and(a > 2, b < 8) =",` `np.logical_and(a > 2, b < 8))`	`np.logical_and(a > 2, b < 8) = [False False  True False]`

- **Mảng 2 chiều**

Mã lệnh	Kết quả

`# Mảng 2 chiều`
`a = np.array([[1, 2], [3, 4]])`
`b = np.array([[10, 30], [50, 70]])`
`print("Array a:", a)`
`print("Array b:", b)`
	`Array a: [[1 2]`
	`[3 4]]`
	`Array b: [[10 30]`
	`[50 70]]`

`# Cộng`
`print("\na + b =\n", a + b)`
	`a + b =`
	`[[11 32]`
	`[53 74]]`

`print("np.add(a, b) =\n", np.add(a, b))`	`np.add(a, b) =`
	`[[11 32]`
	`[53 74]]`

`# Nhân`
`print("\na * b =\n", a * b)`
	`a * b =`
	`[[ 10  60]`
	`[150 280]]`

`print("np.multiply(a, b) =\n",`
`np.multiply(a, b))`
	`np.multiply(a, b) =`
	`[[ 10  60]`
	`[150 280]]`

`# So sánh`
`print("a > 2 =\n", a > 2)`
	`a > 2 =`
	`[[False False]`
	`[ True  True]]`

`print("a == b \n", a == b)`	`a == b`
	`[[False False]`
	`[False False]]`

`# Logic`
`print("(a > 2) & (b < 8) =\n", (a > 2)`
`& (b < 8))`
	`(a > 2) & (b < 8) =`
	`[[False False]`
	`[False False]]`

`print("np.logical_and(a > 2, b < 8) =\n",`
`np.logical_and(a > 2, b < 8))`
	`np.logical_and(a > 2, b < 8) =`
	`[[False False]`
	`[False False]]`

1.1.3. Tạo Numpy array

1.1.3.1. array()

1.1.3.1.1. Công dụng

`numpy.array()` được sử dụng để tạo ra các mảng (`arrays`) trong `NumPy`, là cấu trúc dữ liệu cơ bản cho hầu hết mọi loại tính toán số trong Python.

1.1.3.1.2. Cú pháp của numpy.array()

```
numpy.array(object, dtype=None, copy=True, order='K',
                    subok=False, ndmin=0)
```

trong đó:

- ***object***: (tham số bắt buộc) là đối tượng đầu vào để tạo mảng. Nó có thể là bất kỳ đối tượng nào kiểu mảng (*array-like*), chẳng hạn như:
  - Một `list` hoặc `tuple` của Python
  - Một mảng lồng nhau (*nested list/tuple*)
  - Một đối tượng có `interface` mảng (*array interface*)
  - Một đối tượng khác của `NumPy array`
- ***dtype*** (tùy chọn):
  - Kiểu dữ liệu mong muốn của các phần tử trong mảng.
  - Có thể chỉ định các kiểu dữ liệu như `numpy.int32`, `numpy.float64`, `numpy.complex128`, `numpy.str_`, v.v.
  - Nếu không được cung cấp, `NumPy` sẽ cố gắng suy ra kiểu dữ liệu từ đối tượng đầu vào.
- ***copy*** (tùy chọn):
  - Nếu là `True` (mặc định), một bản sao mới của đối tượng được tạo ra.
  - Nếu là `False`, `NumPy` sẽ cố gắng sử dụng bộ nhớ của đối tượng ban đầu, nếu có thể, để tránh sao chép tốn kém. Tuy nhiên, nếu không thể (ví dụ: nếu đối tượng đầu vào không liền kề trong bộ nhớ), một bản sao vẫn được tạo.
- ***order*** (tùy chọn):
  - Chỉ định thứ tự lưu trữ dữ liệu trong bộ nhớ. Các giá trị có thể là:
    - `'C'` (*C-style*): Các hàng được lưu trữ liên tiếp trong bộ nhớ.
    - `'F'` (*Fortran-style*): Các cột được lưu trữ liên tiếp trong bộ nhớ.
    - `'A'` (*Any*): Chọn thứ tự lưu trữ tối ưu dựa trên đối tượng đầu vào.
    - `'K'` (*Keep*): Giữ nguyên thứ tự của đối tượng đầu vào nếu có thể.
  - Hầu hết người dùng không cần quan tâm về tham số này.
- ***subok*** (tùy chọn):
  - Nếu là `True`, các mảng con (*subclass*) sẽ được truyền qua.
  - Nếu là `False` (mặc định), mảng trả về sẽ là một mảng cơ sở (*base array*).
- ***ndmin*** (tùy chọn):
  - Chỉ định số chiều tối thiểu của mảng kết quả.
  - Ví dụ: nếu ndmin=2, mảng kết quả sẽ luôn có ít nhất 2 chiều, ngay cả khi đối tượng đầu vào chỉ có 1 chiều.

1.1.3.1.3. Một số ví dụ

Mã lệnh	Kết quả
import numpy as np **#Ví dụ 1: Tạo một mảng 1 chiều từ một list** L = [1, 2, 3, 4, 5] A = np.array(L) print('Array được tạo từ list:',A) print('\nKiểu dữ liệu của array A là:',type(my_array))	Array được tạo từ list: [1 2 3 4 5] Kiểu dữ liệu của array A là: <class 'numpy.ndarray'>
# Ví dụ 2: Tạo một mảng 2 chiều từ một list các list: nestedList = [[1, 2, 3], [4, 5, 6]] A2d = np.array(nestedList) print('Array được tạo từ list các list:') print(A2d)	Array được tạo từ list các list: [[1 2 3] [4 5 6]]
# Ví dụ 3: Tạo một mảng với kiểu dữ liệu cụ thể: L = [1, 2, 3, 4, 5] A = np.array(L, dtype=np.float64) print('Array được tạo từ list:',A) print('\nKiểu dữ liệu có trong array là:',A.dtype)	Array được tạo từ list: [1. 2. 3. 4. 5.] Kiểu dữ liệu có trong array là: float64
# Ví dụ 4: Tạo một mảng với số chiều tối thiểu: L = [1, 2, 3, 4, 5] A3d = np.array(my_list, ndmin=3) print('Array 3 chiều được tạo từ list:',A3d) print('Số chiều của array A3d:', A3d.ndim)	Array 3 chiều được tạo từ list: [[[1 2 3 4 5]]] Số chiều của array A3d: 3

1.1.3.2. *arange()*

1.1.3.2.1. Công dụng

Hàm numpy.arange() được sử dụng để tạo một mảng 1 chiều chứa các giá trị cách đều nhau cùng 1 giá trị.

1.1.3.2.2. Cú pháp

numpy.arange([start,] stop[, step,], dtype=None, *, like=None)

trong đó:

- start (tùy chọn): Giá trị bắt đầu của khoảng. Giá trị mặc định là 0.
- stop: Giá trị cuối của khoảng. Lưu ý rằng giá trị này không được bao gồm trong mảng kết quả (tức là khoảng nửa mở [start, stop)).
- step (tùy chọn): Khoảng cách giữa các giá trị liên tiếp trong mảng. Giá trị mặc định là 1.
- dtype (tùy chọn): Kiểu dữ liệu của các phần tử trong mảng kết quả. Nếu không được cung cấp, kiểu dữ liệu sẽ được suy ra từ các đối số đầu vào (start, stop, và step).
- like (tùy chọn): Đối tượng cho phép ghi đè các thuộc tính của mảng kết quả.

1.1.3.2.3. Ví dụ

Mã lệnh	Kết quả

import numpy as np
# Tạo một mảng từ 0 đến 4 	
a = np.arange(5)
print(a)
	[0 1 2 3 4]

# Tạo một mảng từ 1 đến 9 với bước nhảy 2 	
b = np.arange(1, 10, 2)
print(b)
	[1 3 5 7 9]

#Tạo một mảng các số thực từ 0 đến 1 với bước nhảy 0.1
c = np.arange(0, 1.1, 0.1)
	[0. 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.]

`print(c)`	

`# Tạo một mảng các số nguyên từ 10 đến 1`
`d = np.arange(10, 0, -1)`
`print(d)`
	`[10 9 8 7 6 5 4 3 2 1]`

1.1.3.3. diag()

1.1.3.3.1. Giới thiệu

- Hàm numpy.diag() được sử dụng để trích xuất hoặc tạo một đường chéo từ một mảng.
  - **Trích xuất đường chéo:** Nếu đầu vào là một mảng 2 chiều, numpy.diag() trả về một mảng 1 chiều chứa các phần tử trên đường chéo chính (hoặc đường chéo được chỉ định).

Mã lệnh	*Kết quả*

`import numpy as np`
`a = np.array([[90, 80, 70], [60, 50, 40], [30, 20, 10]])`
`print(a)`
	`[[90 80 70]`
	` [60 50 40]`
	` [30 20 10]]`

`diagonal = np.diag(a)` `print(diagonal)`	`[90 50 10]`

  - **Tạo mảng 2 chiều từ đường chéo:** Nếu đầu vào là một mảng 1 chiều, numpy.diag() trả về một mảng 2 chiều vuông với mảng 1 chiều đầu vào nằm trên đường chéo chính và các phần tử khác là 0.

Mã lệnh	*Kết quả*

`import numpy as np`
`a = np.array([1, 2, 3])`
`print(a)`
	`[1 2 3]`

`diagonal_matrix = np.diag(a)`
`print(diagonal_matrix)`
	`[[1 0 0]`
	` [0 2 0]`
	` [0 0 3]]`

- Hàm numpy.diag() trả về:
  - Một mảng 1 chiều chứa các phần tử trên đường chéo nếu đối số là một mảng 2 chiều.
  - Một mảng 2 chiều vuông với các phần tử trên đường chéo có giá trị được lấy từ đối số v (nếu v là một mảng 1 chiều) và các phần tử khác có giá trị là 0.

1.1.3.3.2. Cú pháp

`numpy.diag(v, k=0).`

Trong đó:

- v:
  - Mảng đầu vào.
  - Nếu là một mảng 2 chiều, các phần tử trên đường chéo thứ k được trả về.
  - Nếu là một mảng 1 chiều, một mảng 2 chiều vuông được trả về với các giá trị của mảng 1 chiều trên đường chéo thứ k.
- k:
  - Chỉ số của đường chéo.
  - Số nguyên, tùy chọn. Giá trị mặc định là 0.

- Sử dụng k để chọn đường chéo phụ. Với k > 0, chọn đường chéo trên đường chéo chính. Với k < 0, chọn đường chéo dưới đường chéo chính.

1.1.3.3.3. Ví dụ

Mã lệnh	*Kết quả*

`import numpy as np`
`# Trích xuất đường chéo chính`
`a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])`
`print(a)`
	`[[1 2 3]`
	` [4 5 6]`
	` [7 8 9]]`

`main_diagonal = np.diag(a)` `print("Đường chéo chính:", main_diagonal)`	`Đường chéo chính:[1 5 9]`

`# Tạo ma trận từ một mảng 1 chiều`
`b = np.array([10, 20, 30])`
`diagonal_matrix = np.diag(b)`
`print("\nMa trận đường chéo:\n", diagonal_matrix)`
	`Ma trận đường chéo:`
	` [[10  0  0]`
	` [ 0 20  0]`
	` [ 0  0 30]]`

`# Trích xuất đường chéo trên`
`upper_diagonal = np.diag(a, k=1)`
`print("\nĐường chéo trên (k=1):", upper_diagonal)`
	`Đường chéo trên (k=1):`
	`[2 6]`

`# Trích xuất đường chéo dưới`
`lower_diagonal = np.diag(a, k=-1)`
`print("\nĐường chéo dưới (k-1):", lower_diagonal)`
	`Đường chéo dưới (k-1):`
	`[4 8]`

1.1.3.4. eye()

- Hàm `numpy.eye` tạo ra một mảng 2 chiều với các số `1` trên một đường chéo và các số `0` ở những nơi khác.
- So sánh `identity` và `eye`:
  - `numpy.identity(n)` tạo ra một ma trận vuông `n x n`.
  - `numpy.eye (N, M, k)` linh hoạt hơn `numpy.identity()` vì cho phép chỉ định số hàng, số cột và vị trí của đường chéo.
- Cú pháp:

`numpy.eye(N, M=None, k=0)`

trong đó:

- `N`: Số hàng trong ma trận.
- `M`: Số cột trong ma trận. Nếu None, mặc định là bằng với N.
- `k`: Chỉ số của đường chéo.
  - `k = 0`: (mặc định) Đường chéo chính (bắt đầu từ vị trí hàng 0, cột 0).
  - `k > 0`: Đường chéo trên đường chéo chính.
  - `k < 0`: Đường chéo dưới đường chéo chính.
- Giá trị trả về: Một mảng `NumPy` 2 chiều (ma trận).

- Ví dụ

Mã lệnh	*Kết quả*

`import numpy as np`
`# Tạo ma trận 4x5 với đường chéo chính mặc định (k=0)`
`A = np.eye(4, 5)`
`print('Hàm eye với k=0:\n',A)`
	`Hàm eye với k=0:`
	` [[1. 0. 0. 0. 0.]`
	` [0. 1. 0. 0. 0.]`
	` [0. 0. 1. 0. 0.]`
	` [0. 0. 0. 1. 0.]]`

`import numpy as np`
`# Tạo ma trận 4x5 với đường chéo trên là 1 (k = 1)`
`B = np.eye(4, 5, k=1)`
`print('Hàm eye với k=1:\n',B)`
	`Hàm eye với k=1:`
	` [[0. 1. 0. 0. 0.]`
	` [0. 0. 1. 0. 0.]`
	` [0. 0. 0. 1. 0.]`
	` [0. 0. 0. 0. 1.]]`

`# Tạo ma trận 4x5 với đường chéo dưới là -2 (k = -2)`
`C = np.eye(4, 5, k=-2)`
`print('Hàm eye với k=-2:\n',C)`
	`Hàm eye với k=-2:`
	` [[0. 0. 0. 0. 0.]`
	` [0. 0. 0. 0. 0.]`
	` [1. 0. 0. 0. 0.]`

```
[0. 1. 0. 0. 0.]]
```

1.1.3.5. identity()

- Hàm `numpy.identity()` tạo ra một ma trận đơn vị có kích thước `n x n` (ma trận vuông với giá trị các phần tử trên đường chéo chính đều là `1` và các phần tử còn lại là `0`).
- Cả 2 hàm `numpy`.identity() và numpy.eye() đều có cùng công dụng.
- Cú pháp

`numpy.identity(n)`

trong đó `n`: Kích thước của ma trận vuông (số hàng và số cột)

Giá trị trả về: một mảng NumPy 2 chiều (ma trận vuông) với kích thước `n x n`.

- Ví dụ:

Mã lệnh	*Kết quả*

`import numpy as np`
`# Tạo ma trận đơn vị 4x4`
`I = np.identity(4)`
`print(I)`
	`[[1. 0. 0. 0.]`
	`[0. 1. 0. 0.]`
	`[0. 0. 1. 0.]`
	`[0. 0. 0. 1.]]`

1.1.3.6. linspace()

- Tạo một mảng với các giá trị nằm cách đều nhau trong một khoảng.
- Ví dụ

Mã lệnh	*Kết quả*
`A = np.linspace(0, 1, 5)` `print(A)`	`[0. 0.25 0.5 0.75 1. ]`
`A = np.linspace(0, 10, 7)` `print(A)`	`[0. 1.66666667 3.33333333 5. 6.66666667` `8.33333333 10.]`

1.1.3.7. ones()

1.1.3.7.1. Giới thiệu

- Hàm numpy.ones()được sử dụng để tạo một mảng mới với các phần tử được đặt thành 1.
- Hàm numpy.ones() thường được sử dụng để:
  - Khởi tạo các mảng với các giá trị ban đầu đã biết.
  - Tạo các mặt nạ (masks) trong đó các phần tử có giá trị 1 được chọn cho một số phép toán.
  - Sử dụng trong các phép toán đại số tuyến tính.

1.1.3.7.2. Cú pháp

`numpy.ones(shape, dtype=None, order='C')`

trong đó

- shape :
  - int hoặc tuple của ints
  - Hình dạng của mảng mới. Ví dụ: (2, 3) cho 2 hàng, 3 cột.
- dtype : (tùy chọn) kiểu dữ liệu mong muốn của mảng. Nếu không được cung cấp, mặc định là float64.

- order : (tùy chọn)
  - Có sắp xếp các phần tử ở bộ nhớ theo kiểu C (ưu tiên hàng) hay kiểu Fortran (ưu tiên cột).
  - Là 1 trong 2 giá trị {'C', 'F'}, mặc định là 'C'
- Giá trị trả về của hàm: hàm trả về một mảng với giá trị của tất cả các phần tử đều là 1 và có hình dạng và kiểu dữ liệu đã cho.

1.1.3.7.3. Ví dụ

Mã lệnh	*Kết quả*

`import numpy as np`
`''' Tạo mảng 1 chiều với 5 phần tử 1. Vì dtype không được chỉ định, nên mặc định là float64 '''`
`a = np.ones(5)`
`print(a)`
	`[1. 1. 1. 1. 1.]`

`''' Tạo mảng 2 chiều 3x3 với các phần tử 1. Vì dtype không được chỉ định, nên mặc định là float64 '''`
`b = np.ones((3, 3))`
`print(b)`
	`[[1. 1. 1.]`
	` [1. 1. 1.]`
	` [1. 1. 1.]]`

`'''Tạo mảng 2 chiều 2x4 với kiểu dữ liệu là số nguyên'''`
`c = np.ones((2, 4), dtype=int)`
`print(c)`
	`[[1 1 1 1]`
	` [1 1 1 1]]`

1.1.3.8. random.normal

1.1.3.8.1. Chức năng

- Hàm `numpy.random.normal()` được sử dụng để tạo một mảng các số được lấy mẫu từ phân phối chuẩn (Gaussian).
- Phân phối chuẩn (Gaussian): Phân phối chuẩn là một phân phối xác suất đối xứng, có hình dạng giống như một đường cong hình chuông. Nó được đặc trưng bởi hai tham số:
  - Giá trị trung bình (*mean*): Tâm của phân phối, thường được ký hiệu là μ.
  - Độ lệch chuẩn (*standard deviation*): Độ rộng hoặc độ phân tán của phân phối, thường được ký hiệu là σ.

1.1.3.8.2. Cú pháp

`numpy.random.normal(loc=0.0, scale=1.0, size=None)`

trong đó:

- ***`loc`*** (`float` hoặc `array_like` của `floats`): Giá trị trung bình ("*mean*") của phân phối. Giá trị mặc định là `0`.
- ***`scale`*** (`float` hoặc `array_like` của `floats`): Độ lệch chuẩn ("*standard deviation*") của phân phối. Giá trị mặc định là `1`.
- ***`size`*** (int hoặc tuple các số nguyên, tùy chọn): Hình dạng của mảng đầu ra. Nếu là `None`, sẽ trả về một giá trị đơn lẻ.

1.1.3.8.3. Ví dụ

Mã lệnh	*Kết quả*

`import numpy as np`
`# Tạo một số ngẫu nhiên từ phân phối chuẩn với mean=0 và độ lệch chuẩn=1`
`x = np.random.normal(loc=0, scale=1)`
`print("Số ngẫu nhiên duy nhất:", x)`
	`Số ngẫu nhiên duy nhất: -1.0832378690914826`

`# Tạo một mảng 1D gồm 5 số ngẫu nhiên từ phân phối chuẩn`
`y = np.random.normal(loc=0, scale=1, size=5)`
`print("Mảng 1D:", y)`
	`Mảng 1D: [-0.71053307 -0.6518925 -0.71573965 0.12111454 0.5707885 ]`

`# Tạo một mảng 2D 3x3 từ phân phối chuẩn`
`z = np.random.normal(loc=0, scale=1, size=(3, 3))`
`print("Mảng 2D:\n", z)`
	`Mảng 2D:`
	`[[-0.27404851 1.02447281 -0.26750919]`
	`[-1.10494845 0.38233332 -0.56510558]`
	`[-0.24626115 -1.41522886 0.45403671]]`

`# Tạo một mảng 1D với các giá trị mean khác nhau`
`means = np.array([1, 5, 10])`
`w = np.random.normal(loc=means, scale=2, size=3)`
`print("Mảng 1D với các giá trị mean khác nhau:", w)`
	`Mảng 1D với các giá trị mean khác nhau:`
	`[3.32153671 5.54946357 9.10334896]`

1.1.3.9. random.randint()

1.1.3.9.1. Công dụng

Hàm numpy.random.randint() được sử dụng để tạo một mảng các số nguyên ngẫu nhiên từ một phạm vi cho trước.

1.1.3.9.2. Cú pháp

`numpy.random.randint(low, high=None, size=None, dtype=int)`

trong đó:

- low: Số nguyên thấp nhất có thể được trả về (bao gồm).
- high (tùy chọn): Số nguyên lớn nhất có thể được trả về (không bao gồm). Nếu không được cung cấp, các giá trị sẽ nằm trong khoảng [0, low).
- size (tùy chọn): Hình dạng của mảng đầu ra. Nếu là None, một số nguyên duy nhất được trả về.
- dtype (tùy chọn): Kiểu dữ liệu của mảng kết quả. Giá trị mặc định là int.

1.1.3.9.3. Ví dụ

Mã lệnh	*Kết quả*

`import numpy as np`
`''' Tạo một mảng 1 chiều gồm 5 số nguyên ngẫu nhiên giữa 0 và 9 '''`
`a = np.random.randint(10, size=5)`
`print(a)`
	`[6 0 4 8 5]`

`''' Tạo một mảng 2 chiều 2x3 gồm các số nguyên ngẫu nhiên giữa -10 và 10 '''`
`b = np.random.randint(-10, 10, size=(2, 3))`
`print(b)`
	`[[-7 -7 4]`
	`[-2 -2 -1]]`

`''' Tạo một mảng 2 chiều 2x3x4 gồm các số nguyên ngẫu nhiên chỉ chứa các số 0 hoặc 1'''`
`c = np.random.randint(0, 1, size=(2, 3,4))`
`print(c)`
	`[[[0 0 0 0]`
	`[0 0 0 0]`
	`[0 0 0 0]]`
	
	`[[0 0 0 0]`
	`[0 0 0 0]`
	`[0 0 0 0]]]`

1.1.3.10. tile()

1.1.3.10.1. Công dụng

Hàm numpy.tile() được sử dụng để tạo một mảng mới lớn hơn bằng cách lặp lại một mảng đã cho một số lần nhất định.

1.1.3.10.2. Cú pháp

```
numpy.tile(A, reps)
```

trong đó:

- A: Mảng đầu vào cần lặp lại. Nó có thể là bất kỳ đối tượng nào có thể được chuyển đổi thành một mảng NumPy.
- reps: Số lần lặp lại mảng A. Nó có thể là một số nguyên hoặc một tuple/list các số nguyên:
  - Nếu reps là một số nguyên, A được lặp lại theo số đó lần dọc theo tất cả các chiều của nó.
  - Nếu reps là một tuple hoặc list, nó chỉ định số lần lặp lại cho từng chiều của A. Ví dụ: nếu reps là (2, 3), A được lặp lại 2 lần theo chiều thứ nhất và 3 lần theo chiều thứ hai.

1.1.3.10.3. Ví dụ

Mã lệnh	*Kết quả*

`import numpy as np`
`# Lặp lại một mảng 1 chiều`
`a = np.array([1, 2, 3])`
`b = np.tile(a, 2)`
`print(b)`
	`[1 2 3 1 2 3]`

`# Lặp lại một mảng 1 chiều theo các chiều khác nhau`
`c = np.tile(a, (2, 1))`
`print(c)`
	`[[1 2 3]`
	` [1 2 3]]`

`# Lặp lại một mảng 2 chiều`
`d = np.array([[1, 2], [3, 4]])`
`e = np.tile(d, (2, 2))`
`print(e)`
	`[[1 2 1 2]`
	` [3 4 3 4]`
	` [1 2 1 2]`
	` [3 4 3 4]]`

1.1.3.11.zeros()

1.1.3.11.1. Giới thiệu

Được sử dụng để tạo ra một mảng (`array`) mới với hình dạng (`shape`) và kiểu dữ liệu (`data type`) được chỉ định, và tất cả các phần tử trong mảng này đều được khởi tạo với giá trị là 0.

1.1.3.11.2. Đặc điểm chính của numpy.zeros():

- *Tạo mảng chứa số 0*: Tất cả các phần tử của mảng được tạo ra sẽ có giá trị là 0.
- *Chỉ định hình dạng (Shape)*: phải cung cấp hình dạng (kích thước các chiều) của mảng muốn tạo.
- *Chỉ định kiểu dữ liệu (Data Type - dtype):* có thể tùy chọn chỉ định kiểu dữ liệu cho các phần tử trong mảng (ví dụ: `int`, `float`). Mặc định là `float64`.
- *Thứ tự lưu trữ (Order)*: có thể chỉ định thứ tự lưu trữ trong bộ nhớ là `'C'` (theo hàng, `C-style`) hoặc `'F'` (theo cột, `Fortran-style`). Mặc định là `'C'`.

1.1.3.11.3. Cú pháp

```
numpy.zeros(shape, dtype=float, order='C')
```

trong đó:

- *shape*: là tham số bắt buộc.
  - Có thể là một số nguyên (`integer`) để tạo mảng một chiều. Ví dụ: 5 sẽ tạo mảng [0., 0., 0., 0., 0.].

- Có thể là một tuple các số nguyên để tạo mảng đa chiều. Ví dụ: (2, 3) sẽ tạo mảng 2 chiều với 2 hàng và 3 cột.

- `dtype` (tùy chọn): Kiểu dữ liệu mong muốn cho các phần tử trong mảng (như `numpy.int32`, `numpy.int64`, `numpy.float32`, `numpy.bool`, …). Mặc định là `numpy.float64`.
- `order` (tùy chọn): `{'C', 'F'}`
  - `'C'` (mặc định): Lưu trữ mảng theo thứ tự hàng (*row-major*). Khi duyệt qua bộ nhớ, các phần tử của cùng một hàng sẽ nằm gần nhau.
  - `'F'`: Lưu trữ mảng theo thứ tự cột (*column-major*). Khi duyệt qua bộ nhớ, các phần tử của cùng một cột sẽ nằm gần nhau. Thông thường, tham số này ít khi cần thay đổi trừ khi có lý do cụ thể liên quan đến tối ưu hóa hiệu suất hoặc tương tác với mã Fortran.

1.1.3.11.4. Ví dụ

Mã lệnh	*Kết quả*

`import numpy as np`
`''' 1. Tạo mảng một chiều với 5 phần tử (kiểu float mặc định) '''`
`arr1 = np.zeros(5)`
`print("Mảng 1 chiều (5 phần tử, float):")`
`print(arr1)`
`print("Kiểu dữ liệu của arr1:", arr1.dtype)`
	`Mảng 1 chiều (5 phần tử, float):`
	`[0. 0. 0. 0. 0.]`
	`Kiểu dữ liệu của arr1: float64`

`''' 2. Tạo mảng hai chiều (2 hàng, 3 cột) với kiểu số nguyên int '''`
`arr2 = np.zeros((2, 3), dtype=np.int32)`
`print("Mảng 2 chiều (2x3, int32):")`
`print(arr2)`
`print("Kiểu dữ liệu của arr2:", arr2.dtype)`
	`Mảng 2 chiều (2x3, int32):`
	`[[0 0 0]`
	` [0 0 0]]`
	`Kiểu dữ liệu của arr2: int32`

`''' 3. Tạo mảng ba chiều (2 khối, 3 hàng, 4 cột) '''`
`arr3 = np.zeros((2, 3, 4))`
`print("Mảng 3 chiều (2x3x4, float):")`
`print(arr3)`
	`Mảng 3 chiều (2x3x4, float):`
	`[[[0. 0. 0. 0.]`
	`  [0. 0. 0. 0.]`
	`  [0. 0. 0. 0.]]`
	` [[0. 0. 0. 0.]`
	`  [0. 0. 0. 0.]`
	`  [0. 0. 0. 0.]]]`

`''' 4. Tạo mảng với kiểu boolean'''`
`arr4 = np.zeros((3, 2), dtype=bool) # Hoặc dtype=np.bool_`
`print("Mảng 2 chiều (3x2, boolean):")`
`print(arr4)`
`print("Kiểu dữ liệu của arr4:", arr4.dtype)`
	`Mảng 2 chiều (3x2, boolean):`
	`[[False False]`
	` [False False]`
	` [False False]]`
	`Kiểu dữ liệu của arr4: bool`

`''' 5. Sử dụng tuple để định nghĩa shape trực tiếp '''`
`shape_tuple = (4, 2)`
`arr5 = np.zeros(shape_tuple)`
`print("Mảng 2 chiều (4x2) định nghĩa shape bằng tuple:")`
`print(arr5)`
	`Mảng 2 chiều (4x2) định nghĩa shape bằng tuple:`
	`[[0. 0.]`
	` [0. 0.]`
	` [0. 0.]`
	` [0. 0.]]`

1.1.3.11.5. numpy.zeros() thường được dùng trong các trường hợp:

- Khi cần một mảng "khởi tạo" với các giá trị 0 và biết trước kích thước của nó.
- Để tạo mặt nạ (*mask*) trong xử lý ảnh hoặc dữ liệu.
- Làm bộ đệm (*buffer*) hoặc mảng tích lũy mà các giá trị ban đầu cần phải là 0.
- Khi cần khởi tạo trọng số trong một số mô hình học máy với giá trị 0 (mặc dù các phương pháp khởi tạo khác thường được ưu tiên hơn).

1.1.4. Truy cập các phần tử của Numpy array

1.1.4.1. Truy cập một phần tử của mảng NumPy bằng Index

- *Đối với mảng 1 chiều*
  - Có thể truy cập phần tử bằng cách chỉ định chỉ số của nó trong ngoặc vuông, với chỉ số bắt đầu từ 0.
  - Ví dụ:

Mã lệnh	*Kết quả*

`import numpy as np`
`a = np.array([17, 42, -15, 0, 39])`
`print(a)`
	`[ 17  42 -15   0  39]`

`print(a[0])  # In ra phần tử đầu tiên`	`17`
`print(a[3])  # In ra phần tử thứ tư`	`0`

- *Đối với mảng nhiều chiều*
  - Có thể truy cập phần tử bằng cách chỉ định chỉ số của hàng và cột, được phân tách bằng dấu phẩy, trong ngoặc vuông.
  - Ví dụ

Mã lệnh	*Kết quả*

`import numpy as np`
`a = np.array([[90, 80, 70], [60, 50, 40], [30, 20, 10]])`
`print(a)`
	`[[90 80 70]`
	` [60 50 40]`
	` [30 20 10]]`
	`90`
	`40`
	`20`

`print(a[0, 0])  # In ra phần tử ở hàng đầu tiên, cột đầu tiên`	`90`
`print(a[1, 2])  # In ra phần tử ở hàng thứ hai, cột thứ ba`	`40`
`print(a[2, 1])  # In ra phần tử ở hàng thứ ba, cột thứ hai`	`20`

1.1.4.2. Truy cập một phần của mảng bằng slicing

- *Đối với mảng 1 chiều*
  - Để lấy một phần của mảng 1 chiều, sử dụng cú pháp [start:end:step].
  - Ví dụ

Mã lệnh	*Kết quả*

`import numpy as np`
`a = np.array([90, 80, 70, 60, 50, 40, 30, 20, 10])`
`print(a)`
	`[90 80 70 60 50 40 30 20 10]`

`print(a[1:5]) #In ra các phần tử từ chỉ số 1 đến 4`	`[80 70 60 50]`
`print(a[::2]) #In ra các phần tử ở vị trí chẵn`	`[90 70 50 30 10]`
`print(a[:3])  #In ra 3 phần tử đầu tiên`	`[90 80 70]`
`print(a[-3:]) #In ra 3 phần tử cuối cùng`	`[30 20 10]`

- *Đối với mảng nhiều chiều:*
  - Để lấy một phần của mảng nhiều chiều, bạn sử dụng slicing cho từng chiều, được phân tách bằng dấu phẩy.
  - Ví dụ:

Mã lệnh	*Kết quả*

`import numpy as np`
`a=np.array([[90,80,70,60], [50,40,30,20], [10,0,-10,-20]])`
`print(a)`
	`[[ 90 80 70 60]`
	` [ 50 40 30 20]`
	` [ 10 0 -10 -20]]`

`# In ra một mảng 2x2 từ hàng 0 đến 1, cột 1 đến 2` `print(a[0:2, 1:3])`	`[[80 70]` ` [40 30]]`
`''' In ra các giá trị có index là hàng chẵn (0,2) và cột lẻ (1,3) '''`	`[[ 80  60]` ` [  0 -20]]`

`print(a[::2, 1::2] )`	
`# In ra cột thứ 3 của b` `print(a[:, 2])`	`[ 70  30 -10]`
`# In ra hàng thứ 2 của b` `print(a[1,:])`	`[50 40 30 20]`

1.1.5. Một số thuộc tính của NumPy array

1.1.5.1. dtype

Cho biết Kiểu dữ liệu của các phần tử trong mảng. Ví dụ:

```
print(a.dtype)  # Kết quả: dtype('int64')(hoặc int32 tùy hệ thống)
```

Mã lệnh	*Kết quả*
`A = np.array([[1, 2, 3], [4, 5, 6]])` `print(A)`	`[[1 2 3]` ` [4 5 6]]`
`print(f'Kiểu dữ liệu có trong mảng A là {A.dtype}')`	`Kiểu dữ liệu có trong mảng A là int64`

1.1.5.2. itemsize

Cho biết Số byte cho mỗi phần tử. Ví dụ:

```
print(a.itemsize)  # Kết quả: 8 (nếu là int64 và là 4 khi là int32)
```

Mã lệnh	*Kết quả*
`A = np.array([[1, 2, 3], [4, 5, 6]])` `print(f'Mỗi phần tử trong mảng chiếm {A.itemsize} bytes')`	`Mỗi phần tử trong mảng chiếm 8 bytes`

1.1.5.3. ndim

Cho biết số chiều (*dimensions*) của mảng. Ví dụ:

Mã lệnh	*Kết quả*

`import numpy as np`
`A = np.array([[1, 2, 3], [4, 5, 6]])`
`print(A)`
`print(f'A là mảng {A.ndim} chiều')`
	`[[1 2 3]`
	` [4 5 6]]`
	`a là mảng 2 chiều`

1.1.5.4. shape

Cho biết Kích thước của mảng dưới dạng tuple, biểu thị số phần tử trên mỗi chiều. Ví dụ:

Mã lệnh	*Kết quả*

`A = np.array([[1, 2, 3], [4, 5, 6]])`
`print(A)`
`r,c = A.shape`
`print(f'Mảng A gồm {r} hàng và {c} cột')`
	`[[1 2 3]`
	` [4 5 6]]`
	`Mảng A gồm 2 hàng và 3 cột`

1.1.5.5. size

Cho biết tổng số phần tử trong mảng. Ví dụ:

Mã lệnh	*Kết quả*

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

`print(f'Mảng A gồm {r*c} phần tử')`	`Mảng A gồm 6 phần tử`
`print(f'Mảng A gồm {A.size} phần tử')`	`Mảng A gồm 6 phần tử`

1.1.5.6. T

1.1.5.6.1. Chức năng

- Thuộc tính `.T` được sử dụng để lấy ma trận chuyển vị của một mảng NumPy.
- Chuyển vị ma trận là gì? Chuyển vị của một ma trận là một phép toán trong đó các hàng và cột của ma trận được hoán đổi cho nhau. Với một ma trận `A` có kích thước (`m x n`), thì ma trận chuyển vị của nó, ký hiệu là `A.T`, sẽ có kích thước (`n x m`).

1.1.5.6.2. Cách hoạt động của .T

- *Mảng 1 chiều*: Đối với một mảng `1` chiều, chuyển vị không làm thay đổi mảng.
- *Mảng 2 chiều*: Đối với một mảng `2` chiều (ma trận), `.T` hoán đổi các hàng và cột. Phần tử ở vị trí (`i, j`) trong mảng ban đầu sẽ ở vị trí (`j, i`) trong mảng chuyển vị.
- *Mảng nhiều chiều* (> 2 chiều): Đối với một mảng có nhiều hơn 2 chiều, `.T` đảo ngược thứ tự của các chiều. Ví dụ: với một mảng có kích thước (`d1, d2, d3`), thì chuyển vị của nó sẽ có kích thước (`d3, d2, d1`).

1.1.5.6.3. Ví dụ

Mã lệnh	*Kết quả*

`import numpy as np`
 	
`# Mảng 1 chiều`
`a = np.array([1, 2, 3])`
`a_t = a.T`
`print("Mảng 1 chiều:")`
`print(a)`
	`Mảng 1 chiều:`
	`[1 2 3]`

`print("Chuyển vị của mảng 1 chiều:")` `print(a_t)`	`Chuyển vị của mảng 1 chiều:` `[1 2 3]`

`# Mảng 2 chiều`
`b = np.array([[1, 2, 3],`
`[4, 5, 6]])`
`b_t = b.T`
`print("\nMảng 2 chiều:")`
`print(b)`
	`Mảng 2 chiều:`
	`[[1 2 3]`
	` [4 5 6]]`

`print("Chuyển vị của mảng 2 chiều:")`
`print(b_t)`
	`Chuyển vị của mảng 2 chiều:`
	`[[1 4]`
	` [2 5]`
	` [3 6]]`

`# Mảng 3 chiều`
`c = np.array([[[1, 2],`
`[3, 4]],`
`[[5, 6],`
`[7, 8]]])`
`c_t = c.T`
`print("\nMảng 3 chiều:")`
`print(c)`
	`Mảng 3 chiều:`
	`[[[1 2]`
	`[3 4]]`
	
	`[[5 6]`
	`[7 8]]]`

`print("Chuyển vị của mảng 3 chiều:")`
`print(c_t)`
	`Chuyển vị của mảng 3 chiều:`
	`[[[1 5]`
	`[3 7]]`
	
	`[[2 6]`
	`[4 8]]]`

1.1.6. Một số phương thức của NumPy array

1.1.6.1. arange()

- Tạo một mảng từ một dãy số được phát sinh bởi hàm range.
- Ví dụ:

Mã lệnh	Kết quả
`A = np.arange(1,15,3)` `print(A)`	`[ 1  4  7 10 13]`

1.1.6.2. astype()

- Chuyển đổi kiểu dữ liệu của mảng.
- Ví dụ

Mã lệnh	Kết quả
`A = np.array([[1, 2, 3], [4, 5, 6]])` `print('Kiểu dữ liệu của các phần tử trong A là:', A.dtype)`	`Kiểu dữ liệu của các phần tử trong A là: int64`
`B = A.astype(float)` `print('Kiểu dữ liệu của các phần tử trong B là:', B.dtype)`	`Kiểu dữ liệu của các phần tử trong B là: float64`

1.1.6.3. flatten()

- Trả về một mảng 1 chiều từ mảng gốc.
- Ví dụ

Mã lệnh	Kết quả
`A = np.array([[1, 2, 3], [4, 5, 6]])` `print(A)`	`[[1 2 3]` ` [4 5 6]]`
`B = A.flatten()` `print(B)`	`[1 2 3 4 5 6]`

1.1.6.4. reshape()

- Thay đổi hình dạng của mảng mà không làm thay đổi dữ liệu. Khi 1 trong các chiều là -1 (kích thước không xác định), thì kích thước này sẽ do numpy tự tính toán dựa trên tổng số phần tử của mảng và kích thước của các chiều còn lại.
- Ví dụ

Mã lệnh	Kết quả
`A = np.array([[1, 2, 3], [4, 5, 6]])` `print(A)`	`[[1 2 3]` ` [4 5 6]]`

`B = A.`**`reshape`**`(3, 2)`
`print(B)`
	`[[1 2]`
	` [3 4]`
	` [5 6]]`

`C = C.`**`reshape`**`(1, 6)` `print(B)`	`[[1 2 3 4 5 6]]`

`B = A.`**`reshape`**`(`**`-1`**`, 2)`
`print(B)`
	`[[1 2]`
	` [3 4]`
	` [5 6]]`

1.1.6.5. transpose()

- Chuyển vị của mảng (đổi hàng thành cột và ngược lại).
- Ví dụ

Mã lệnh	Kết quả
`A = np.array([[1, 2, 3], [4, 5, 6]])` `print(A)`	`[[1 2 3]` ` [4 5 6]]`

`B = A.tranpose()`
`print(B)`
	`[[1 4]`
	` [2 5]`
	` [3 6]]`

1.1.6.6. sum(), mean(), max(), min()

- Tính tổng, trung bình, giá trị lớn nhất và nhỏ nhất trong mảng.
- Ví dụ

Mã lệnh	Kết quả
`A = np.array([[1, 2, 3], [4, 5, 6]])`	
`print(A.sum())`	21
`print(A.mean())`	3.5
`print(A.max())`	6
`print(A.min())`	1

1.1.7. Một số hàm của NumPy sử dụng trên array

1.1.7.1. argsort

1.1.7.1.1. Chức năng

Hàm `numpy.argsort()` được sử dụng để tìm các chỉ số sắp xếp một mảng. Hàm này trả về 1 mảng các chỉ số mà nếu sử dụng chúng để sắp xếp mảng ban đầu, ta sẽ được một mảng đã sắp xếp.

1.1.7.1.2. Cú pháp

`numpy.argsort(a, axis=-1, kind=None, order=None)`

trong đó:

- `a`: Mảng đầu vào.
- `axis` (tùy chọn): Trục dọc theo đó thực hiện sắp xếp. Giá trị mặc định là `-1`, tức là trục cuối cùng.
- `kind` (tùy chọn): Thuật toán sắp xếp được sử dụng. Các giá trị có thể là `{'quicksort', 'mergesort', 'heapsort', 'stable', 'None'}`. Nếu là `None`, thuật toán mặc định được sử dụng.
- `order` (tùy chọn): Khi `a` là một mảng có các field, tham số này chỉ định field nào cần so sánh trước.

1.1.7.1.3. Ví dụ

Mã lệnh	Kết quả

`import numpy as np`
`# Mảng 1 chiều`
`a = np.array([3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5])`
`print("Mảng ban đầu:", a)`
	`Mảng 1D ban đầu: [3 1 4 1 5 9 2 6 5 3 5]`

`a1D = np.argsort(a)` `print("Chỉ số sắp xếp:", a1D)`	`Chỉ số sắp xếp: [ 1  3  6  0  9  2  4  8 10  7  5]`

`# Mảng 2 chiều`
`b = np.array([[8, 4, 1],`
`[6, 2, 9],`
`[3, 7, 5]])`
`print("\nMảng ban đầu:\n", b)`
	`Mảng 2D ban đầu:`
	`[[8 4 1]`
	`[6 2 9]`
	`[3 7 5]]`

`a2D_axis0 = np.argsort(b, axis=0)`
`print("Chỉ số sắp xếp theo cột:\n", a2D_axis0)`
	`Chỉ số sắp xếp theo cột:`
	`[[2 1 0]`
	`[1 0 2]`
	`[0 2 1]]`

`argS2D_axis1 = np.argsort(b, axis=1)`
`print("Chỉ số sắp xếp theo hàng:\n", a2D_axis1)`
	`Chỉ số sắp xếp theo hàng:`
	`[[2 1 0]`
	`[1 0 2]`
	`[0 2 1]]`

1.1.7.2. dot() hay toán tử @

1.1.7.2.1. Công dụng

Hàm numpy.dot() là một trong những hàm quan trọng nhất trong NumPy, được sử dụng để thực hiện phép nhân ma trận và nhân vector. Nó có thể được sử dụng cho các mảng 1D (vector) và 2D (ma trận):

- *Nhân vector*: Nếu cả a và b đều là các mảng 1D, numpy.dot() thực hiện tích vô hướng (inner product) của hai vector.
- *Nhân ma trận*: Nếu a là một mảng 2D và b là một mảng 2D, numpy.dot() thực hiện phép nhân ma trận thông thường.
- *Nhân ma trận và vector*: Nếu a là một mảng 2D và b là một mảng 1D, numpy.dot() thực hiện phép nhân ma trận và vector.

1.1.7.2.2. Lưu ý

- Hàm numpy.dot() khác với toán tử * trong NumPy. Toán tử * thực hiện phép nhân theo từng phần tử (element-wise multiplication), trong khi numpy.dot() thực hiện phép nhân ma trận.
- Để nhân ma trận, số cột của mảng thứ nhất phải bằng số hàng của mảng thứ hai. Nếu không, bạn sẽ gặp lỗi.
- Trong các phiên bản NumPy gần đây, toán tử @ cũng có thể được sử dụng để thay thế cho numpy.dot() khi nhân ma trận. Ví dụ: a @ b tương đương với numpy.dot(a, b).

1.1.7.2.3. Cú pháp

```
numpy.dot(a, b, out=None)
```

trong đó:

- a: Mảng thứ nhất.
- b: Mảng thứ hai.
- out (tùy chọn): Mảng đầu ra để lưu trữ kết quả. Nếu được cung cấp, nó phải có hình dạng và kiểu dữ liệu phù hợp.

1.1.7.2.4. Giá trị trả về

Tích của hai mảng, có kiểu dữ liệu là kiểu dữ liệu của các phần tử của a và b. Nếu out được cung cấp, nó sẽ trả về một tham chiếu đến out.

1.1.7.2.5. Một số ví dụ

Mã lệnh	*Kết quả*

`import numpy as np`
`''' Nhân hai vector (1D array): Tính tích vô hướng của hai vector. '''`
`a = np.array([1, 2, 3])`
`b = np.array([4, 5, 6])`
`result = np.dot(a, b)`
`print(f'Tích vô hướng của hai vector = {result}')`
	`Tích vô hướng của hai vector = 32`

`''' Nhân ma trận với vector (2D array và 1D array): Nhân một ma trận với một vector cột. Số cột của ma trận phải bằng độ dài của vector.'''` `a = np.array([[1, 2], [3, 4], [5, 6]])`	`Kết quả nhân ma trận với vector:` `[23 53 83]`

```
b = np.array([7, 8])
result = np.dot(a, b)
print(f'Kết quả nhân ma trận với vector:\n{result}')
```

```
''' Nhân hai ma trận (2D array): Khi nhân hai ma trận, số cột
của ma trận thứ nhất phải bằng số hàng của ma trận thứ hai.
'''
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6, 7], [8, 9, 10]])
result = np.dot(a, b)
print(f'Kết quả nhân hai ma trận:\n{result}')
```

```
Kết quả nhân
hai ma trận:
[[21 24 27]
 [47 54 61]]
```

1.1.7.3. hstack()

- Chức năng: Ghép các mảng theo chiều ngang (theo cột).
- Điều kiện: Các mảng đầu vào phải có cùng số hàng.
- Cú pháp: **`numpy.hstack(tup)`**
  Trong đó: tup: Một tuple chứa các mảng cần ghép.
- Ví dụ:

Mã lệnh	*Kết quả*

`import numpy as np`
`a = np.array([1, 2, 3])`
`print(f'Mảng 1 chiều a:{a}')`
	`Mảng 1 chiều a:[1 2 3]`

`b = np.array([4, 5, 6])` `print(f'Mảng 1 chiều b:{b}')`	`Mảng 1 chiều b:[4 5 6]`
`c = np.hstack((a, b))` `print(f'Ghép a và b bằng hstack: {c}')`	`Ghép a và b bằng hstack:` `[1 2 3 4 5 6]`

`d = np.array([[1], [2], [3]])`
`print(f'Mảng 2 chiều d:\n{d}')`
	`Mảng 2 chiều d:`
	`[[1]`
	` [2]`
	` [3]]`

`e = np.array([[4], [5], [6]])`
`print(f'Mảng 2 chiều e: \n {e}')`
	`Mảng 2 chiều e:`
	`[[4]`
	` [5]`
	` [6]]`

`f = np.hstack((d, e))`
`print(f'Ghép d và e bằng hstack: \n{f}')`
	`Ghép d và e bằng hstack:`
	`[[1 4]`
	` [2 5]`
	` [3 6]]`

1.1.7.4. sum()

1.1.7.4.1. Chức năng

Hàm numpy.sum() được sử dụng để tính tổng các phần tử trong một mảng NumPy theo một trục xác định.

1.1.7.4.2. Cú pháp

```
numpy.sum(a, axis=None, dtype=None, out=None, keepdims=False,
                    initial=0, where=True)
```

trong đó:

- a: Mảng cần tính tổng.
- axis (tùy chọn): Trục mà các phần tử được tính tổng. Giá trị mặc định là None, tính tổng của tất cả các phần tử trong mảng.
  - axis = 0: Tính tổng theo các cột.
  - axis = 1: Tính tổng theo các hàng.
- dtype (tùy chọn): Kiểu dữ liệu của tổng trả về.

- out (tùy chọn): Mảng đầu ra để lưu trữ kết quả.
- keepdims (tùy chọn): Nếu là True, các chiều của a sẽ được giữ lại trong kết quả với kích thước 1.
- initial (tùy chọn): Giá trị ban đầu của tổng.
- where (tùy chọn): Các phần tử được đưa vào tính tổng.

1.1.7.4.3. Ví dụ

Mã lệnh	*Kết quả*

`import numpy as np`
`# Mảng 1 chiều`
`a = np.array([11, 12, 13, 14, 15, 16])`
`print("Mảng 1 chiều a:", a)`
	`Mảng 1 chiều a: [11 12 13 14 15 16]`

`tong_a = np.sum(a)` `print("Tổng của a:", tong_a)`	`Tổng của a: 81`
`tong_a_chan = np.sum(c, where=c % 2 == 0)` `print("Tổng các phần tử chẵn của a:", tong_a_chan)`	`Tổng các phần tử chẵn của a: 42`

`# Mảng 2 chiều`
`b = np.array([[1, 2, 3], [4, 5, 6]])`
`print(f"Mảng 2 chiều b:\n{b}")`
	`Mảng 2 chiều b:`
	`[[1 2 3]`
	` [4 5 6]]`

`tong_b_toanbo = np.sum(b)` `print("Tổng của b (tất cả các phần tử):", tong_b_toanbo)`	`Tổng của b (tất cả các phần tử): 21`
`tong_b_theo_cot = np.sum(b, axis=0)` `print("Tổng của b theo từng cột:", tong_b_theo_cot)`	`Tổng của b theo từng cột: [5 7 9]`
`tong_b_theo_hang = np.sum(b, axis=1)` `print("Tổng của b theo từng hàng:", tong_b_theo_hang)`	`Tổng của b theo từng hàng: [ 6 15]`
`tong_b_chan = np.sum(b, where=b % 2 == 0)` `print("Tổng các phần tử chẵn của b:", tong_b_chan)`	`Tổng các phần tử chẵn của b: 12`

1.1.7.5. tolist()

1.1.7.5.1. Chức năng

Hàm `numpy.array.tolist()` được sử dụng để chuyển đổi một mảng NumPy thành một `list` Python. Cụ thể là hàm này trả về một `list` Python có cùng các phần tử và cấu trúc với mảng NumPy ban đầu.

1.1.7.5.2. Cú pháp

`ndarray.tolist()`

1.1.7.5.3. Ví dụ

Mã lệnh	*Kết quả*

`import numpy as np`
`a = np.array([1, 2, 3])`
`print(f'Mảng 1 chiều: {a}')`
	`Mảng 1 chiều: [1 2 3]`

`b = a.tolist()` `print(f'Chuyển mảng 1 chiều thành list: {b}')`	`Chuyển mảng 1 chiều thành list: [1, 2, 3]`

`c = np.array([[1, 2, 3], [4, 5, 6]])`
`print(f'Mảng 2 chiều: \n{c}')`
	`Mảng 2 chiều:`
	`[[1 2 3]`
	` [4 5 6]]`

`d = c.tolist()` `print(f'Chuyển mảng 2 chiều thành list: {d}')`	`Chuyển mảng 2 chiều thành list: [[1,2,3], [4,5,6]]`

`e = np.array([[[1, 2, 3,4], [5, 6,7,8], [9,10,11,12]]])`
`print(f'Mảng 3 chiều: \n{e}')`
	`Mảng 3 chiều:`
	`[[[ 1  2  3  4]`
	`  [ 5  6  7  8]`
	`  [ 9 10 11 12]]]`

`f = e.tolist()` `print(f'Chuyển mảng 3 chiều thành list: {f}')`	`Chuyển mảng 3 chiều thành list: [[[1,2,3,4], [5,6,7,8], [9,10,11,12]]]`

1.1.7.6. vstack()

- Chức năng: Ghép các mảng theo chiều dọc (theo hàng).
- Điều kiện: Các mảng đầu vào phải có cùng số cột.
- Cú pháp: **`numpy.vstack(tup)`**

  Trong đó: tup: Một tuple chứa các mảng cần ghép.
- Ví dụ:

Mã lệnh	*Kết quả*

`import numpy as np`
`a = np.array([1, 2, 3])`
`print(f'Mảng 1 chiều a:{a}')`
	`Mảng 1 chiều a:[1 2 3]`

`b = np.array([4, 5, 6])` `print(f'Mảng 1 chiều b:{b}')`	`Mảng 1 chiều b:[4 5 6]`

`c = np.vstack((a, b))`
`print(f'Ghép a và b bằng vstack: {c}')`
	`Ghép a và b bằng vstack:`
	`[[1 2 3]`
	` [4 5 6]]`

`d = np.array([[1], [2], [3]])`
`print(f'Mảng 2 chiều d:\n{d}')`
	`Mảng 2 chiều d:`
	`[[1]`
	` [2]`
	` [3]]`

`e = np.array([[4], [5], [6]])`
`print(f'Mảng 2 chiều e: \n {e}')`
	`Mảng 2 chiều e:`
	`[[4]`
	` [5]`
	` [6]]`

`f = np.vstack((d, e))`
`print(f'Ghép d và e bằng vstack: \n{f}')`
	`Ghép d và e bằng vstack:`
	`[[1]`
	` [2]`
	` [3]`
	` [4]`
	` [5]`
	` [6]]`

1.1.7.7. where()

1.1.7.7.1. Công dụng

Hàm `numpy.where()` giúp thực hiện các hoạt động có điều kiện trên các mảng. Nó trả về các phần tử được chọn từ các mảng dựa trên một điều kiện.

1.1.7.7.2. Cú pháp

`numpy.where(condition, [x, y])`

trong đó:

- `condition`: Một mảng hoặc điều kiện boolean. Các giá trị `True` chỉ định nơi lấy giá trị từ `x`.
- `x`: (Tùy chọn) Mảng các giá trị để chọn khi condition là `True`.
- `y`: (Tùy chọn) Mảng các giá trị để chọn khi condition là `False`.

1.1.7.7.3. Giá trị trả về

- Khi chỉ có `condition`: Một tuple của các mảng chỉ số, một cho mỗi chiều của array, trỏ đến các phần tử True.
- Khi cả `condition`, `x` và `y` đều được cung cấp: Một mảng với các phần tử từ `x` nếu `condition` là `True`, và các phần tử từ `y` nếu `condition` là `False`.

1.1.7.7.4. Ví dụ

Mã lệnh	Kết quả

`''' Tìm các chỉ số của các phần tử True'''`
`arr = np.array([10, 20, 30, 40, 50])`
`index = np.where(arr > 30)`
`print("Các chỉ số của các phần tử có giá trị > 30:", index)`
	`Các chỉ số của các phần tử có giá trị > 30:`
	`(array([3, 4], dtype=int64),)`

`#Chọn các giá trị từ x hoặc y dựa trên điều kiện'''`
`x = np.array(['x1', 'x2', 'x3', 'x4', 'x5'])`
`y = np.array([['y1', 'y2', 'y3', 'y4', 'y5']])`
`result = np.where(arr > 30, x, y)`
`print("Các giá trị được chọn:", result)`
	`Các giá trị được chọn:`
	`[['y1' 'y2' 'y3' 'x4' 'x5']]`

`#Sử dụng với mảng 2 chiều'''`
`matrix = np.array([[1, 9, 3], [7, 5, 6], [4, 8, 2]])`
`result2 = np.where(matrix > 5, matrix, 0)`
`print("Các giá trị được chọn từ ma trận:\n", result2)`
	`Các giá trị được chọn từ ma trận:`
	`[[0 9 0]`
	`[7 0 6]`
	`[0 8 0]]`

1.2. numpy.random

Các phương thức chính

1.2.1. random() hoặc rand()

- Tạo mảng các số ngẫu nhiên trong khoảng [0, 1) với kích thước cụ thể.
- Lưu ý khi dùng `np.random.random`: kích thước mảng được đặt trong 1 tuple.
- Ví dụ: tạo mảng 2x3 các số ngẫu nhiên từ [0, 1)

Mã lệnh	Kết quả

`import numpy as np`
`A = np.random.random((2, 3))`
`print(A)`
	`[[0.10509832 0.92227135 0.87790596]`
	`[0.42463625 0.11442098 0.33705612]]`

`A = np.random.rand(2, 3)` `print(A)`	`[[0.86569341 0.69308854 0.8224038 ]` `[0.29007718 0.01644588 0.83445395]]`

1.2.2. randint()

- Tạo các số nguyên ngẫu nhiên trong một khoảng cho trước.
- Ví dụ: tạo mảng 2x3 các số nguyên từ 1 đến 9

Mã lệnh	Kết quả
`A = np.random.randint(1, 10, size=(2, 3))` `print(A)`	`[[7 4 3]` `[6 2 9]]`

1.2.3. choice()

- Chọn ngẫu nhiên các phần tử từ một mảng hoặc danh sách.
- Ví dụ: tạo mảng A gồm 5 phần tử bằng cách chọn ngẫu nhiên trong các phần tử có trong list (tham số thứ 1)

Mã lệnh	Kết quả
`A = np.random.choice([i for i in range (0,100,10)], size = 5)` `print(A)`	`[10 20 90 90 50]`

1.2.4. normal()

- Tạo các số tuân theo phân phối chuẩn (Gaussian distribution).
- Cú pháp: `result = np.random.normal(loc = a, scale = b, size = c)`
  - `loc`: trung bình.
  - `scale`: độ lệch chuẩn.

- `size`: số lượng phần tử cần tạo.
- `a, b, c`: các hằng số, với `a` và `b` có kiểu float.

- Ví dụ: tạo một mảng 5 số ngẫu nhiên theo phân phối chuẩn

Mã lệnh	*Kết quả*

`A = np.random.normal(loc=0.2, scale=10.5, size=5)`
`print(A)`
	`[ 5.23547213 -1.58412538`
	`7.71336532 -1.0186128`
	`6.20252605]`

1.2.5. uniform()

- Tạo các số ngẫu nhiên theo phân phối đồng đều trong khoảng `[low, high)`.
- Ví dụ: Mảng 2x3 các số ngẫu nhiên trong khoảng `[1.0, 10.0)`

Mã lệnh	*Kết quả*

`A = np.random.uniform(low=1.0,`
`high=10.0, size=(2, 3))`
`print(A)`
	`[[3.54232942 2.59458178 6.66759107]`
	`[5.91254447 2.86972511`
	`9.18519727]]`

1.2.6. shuffle()

- Trộn ngẫu nhiên thứ tự các phần tử trong mảng.
- Ví dụ:

Mã lệnh	*Kết quả*

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

1.2.7. seed()

- Thiết lập hạt giống (seed) cho trình tạo số ngẫu nhiên để tái tạo kết quả giống nhau trong các lần chạy sau đó, giúp đảm bảo tính tái lập (reproducibility) của kết quả.
- *Hạt giống (seed)*: Là một số nguyên được dùng để khởi tạo trình tạo số ngẫu nhiên (random number generator). Khi cùng một hạt giống được thiết lập, trình tạo số ngẫu nhiên sẽ tạo ra một chuỗi giá trị ngẫu nhiên giống nhau.
- *Tính tái lập*: Nếu không thiết lập hạt giống, các giá trị ngẫu nhiên sẽ thay đổi mỗi lần chạy chương trình.
- *Không dùng chung hạt giống trong nhiều phần của chương trình*: Nếu cần tạo ra các giá trị ngẫu nhiên khác biệt trong mỗi phần, có thể thiết lập hạt giống khác nhau cho từng phần.
- *Ứng dụng*: Hữu ích trong học máy, nghiên cứu và kiểm thử khi cần kết quả ngẫu nhiên giống nhau để tái hiện hoặc kiểm tra.
- Ví dụ:

Mã lệnh	*Kết quả*

`#CÓ thiết lập seed => Kết quả KHÔNG thay đổi mỗi lần chạy`
`# Thiết lập hạt giống`
`np.random.seed(42)`
`# Tạo một mảng GỒM 5 số ngẫu nhiên`
`A = np.random.rand(5)`
`print(A)`
	`[0.37454012`
	`0.95071431`
	`0.73199394`
	`0.59865848`
	`0.15601864]`

`''' Nếu chạy lại với cùng hạt giống, giá trị các số không thay đổi '''`
`np.random.seed(42)`
`A2 = np.random.rand(5)`	

`print(A2)`	

`# KHÔNG thiết lập SEED => Kết quả thay đổi mỗi lần chạy:`
`result = np.random.rand(5)`
`print(result)`
	`[0.86310343`
	`0.62329813`
	`0.33089802`
	`...]`

1.2.8. binomial()

- Tạo các số ngẫu nhiên tuân theo phân phối nhị thức.
- Được sử dụng để tạo các số ngẫu nhiên theo phân phối nhị thức (*binomial distribution*). Phân phối nhị thức thường được sử dụng để mô phỏng kết quả của một chuỗi các phép thử độc lập (vd: tung đồng xu).
- Công thức:

  `np.random.binomial(n=a, p=b, size=c)`

  trong đó:
  - `a, b, c` là các giá trị số
  - `n` (int):
    - Tổng số phép thử (số lần thực hiện thí nghiệm).
    - Ví dụ: Khi tung một đồng xu 10 lần, thì `a=10`, tức là `n = 10`.
  - `p` (float):
    - Xác suất thành công của mỗi phép thử (phải nằm trong khoảng `[0, 1]`).
    - Ví dụ: Nếu xác suất tung đồng xu ra mặt ngửa là `0.5`, thì `b=0.5`, tức là `p = 0.5`.
  - `size` (int hoặc tuple):
    - Số lượng giá trị ngẫu nhiên cần tạo.
    - Nếu `size = (2, 3)`, phương thức sẽ trả về một mảng 2x3 chứa các giá trị ngẫu nhiên.
- Ví dụ:

Mã lệnh	*Kết quả*

`# mảng 1 chiều với 5 số từ phân phối nhị thức`
`A = np.random.binomial(n=10, p=0.5, size=5)`
`print(A)`
	`[5 6 2 8 7]`

`# mảng 2 chiều 3x4 với các số từ phân phối nhị thức`
`A2 = np.random.binomial(n=10, p=0.5, size=(3,4))`
`print(A2)`
	`[[4 4 4 4]`
	` [5 5 4 5]`
	` [3 4 4 5]]`

1.2.9. permutation()

- Trả về một hoán vị ngẫu nhiên của mảng đầu vào.
- Ví dụ:

Mã lệnh	*Kết quả*
`A = np.random.permutation(np.array([1, 2, 3, 4]))` `print(A)`	`[4 1 2 3]`

2. PANDAS

2.1. GIỚI THIỆU

`Pandas` là một thư viện mã nguồn mở cung cấp các cấu trúc dữ liệu và công cụ phân tích dữ liệu mạnh mẽ cho ngôn ngữ lập trình Python. `Pandas` cho phép xử lý dữ liệu dễ dàng và hiệu quả hơn, từ việc nhập và xuất dữ liệu, xử lý dữ liệu, đến việc thực hiện các thao tác phân tích và trực quan hóa dữ liệu.

- ***Các ứng dụng chính của* `Pandas`**
  - *Xử lý dữ liệu*: `Pandas` giúp làm sạch, biến đổi và thao tác với dữ liệu một cách dễ dàng và hiệu quả.
  - *Phân tích dữ liệu*: `Pandas` cung cấp các công cụ mạnh mẽ để phân tích dữ liệu, từ thống kê mô tả đến các phân tích phức tạp hơn.
  - *Trực quan hóa dữ liệu*: `Pandas` có thể kết hợp với các thư viện trực quan hóa dữ liệu khác như `Matplotlib` và `Seaborn` để tạo ra các biểu đồ và đồ thị trực quan.
  - *Tích hợp dữ liệu*: Pandas hỗ trợ nhập và xuất dữ liệu từ nhiều định dạng khác nhau như `CSV`, `Excel`, `SQL`, ...
- ***Các chức năng cơ bản của* `Pandas`**

`Pandas` cung cấp nhiều chức năng và phương thức để làm việc với dữ liệu, dưới đây là một số chức năng cơ bản:

- *Nhập và xuất dữ liệu*: `Pandas` hỗ trợ đọc và ghi dữ liệu từ/đến nhiều định dạng khác nhau.
- *Thao tác với dữ liệu*: `Pandas` cho phép thực hiện các thao tác như thêm, xóa, lọc và sắp xếp dữ liệu.
- *Tính toán thống kê*: `Pandas` cung cấp các phương thức để tính toán các số liệu thống kê mô tả như trung bình, độ lệch chuẩn, v.v.
- *Xử lý dữ liệu thiếu*: `Pandas` cung cấp các công cụ để xử lý dữ liệu thiếu, như điền giá trị mặc định hoặc loại bỏ các hàng/cột chứa giá trị thiếu.
- *Tích hợp với các công cụ phân tích khác*: `Pandas` có thể kết hợp với các thư viện phân tích và trực quan hóa dữ liệu khác như `NumPy`, `SciPy`, `Matplotlib` và `Seaborn`.

2.2. MỘT SỐ THUỘC TÍNH THƯỜNG DÙNG TRONG PANDAS

2.2.1. iloc &loc

- **`iloc`**: Là một thuộc tính được sử dụng để truy cập dữ liệu của một `DataFrame` hoặc `Series` theo vị trí số nguyên (index-based) tính từ 0 trở đi. Thuộc tính này rất hữu ích khi muốn lấy dữ liệu bằng vị trí hoặc thực hiện các thao tác dựa trên thứ tự trong `DataFrame` hoặc `Series`.
- **`loc`**: tương tự như `iloc` nhưng truy cập theo nhãn của index (label index).
- Cú pháp:
  - `Series`:

SeriesName.iloc[index]

Hoặc **SeriesName.loc[label_index]**

- DataFrame:

DataframeName.iloc[row_index:column_dindex]

Hoặc **DataframeName.loc[labelrow_index:labelcolumnindex]**

- Ví dụ với Series

Mã lệnh	*Kết quả*

• *Truy cập một phần tử theo vị trí:*
`import pandas as pd`
`# Tạo Series với chỉ số tùy chỉnh`
`s = pd.Series([10, 20, 30, 40], index=['a', 'b', 'c', 'd'])`
`# Lấy giá trị ở vị trí thứ 2 (tương ứng giá trị '30')`
`print(s.iloc[2])`
	`30`

• *Truy cập một dải các phần tử bằng slicing:*
`# Lấy phần tử từ vị trí 1 đến 3 (không bao gồm vị trí 3)`
`print(s.iloc[1:3])`
	`b 20`
	`c 30`
	`dtype: int64`

• *Truy cập nhiều vị trí bằng danh sách:*
`# Lấy phần tử ở vị trí thứ 0 và 2`
`print(s.iloc[[0, 2]])`
	`a 10`
	`c 30`
	`dtype: int64`

• *Cập nhật giá trị tại vị trí cụ thể:*
`# Thay đổi giá trị tại vị trí 1 thành 25`
`s.iloc[1] = 25`
`print(s)`
	`a 10`
	`b 25`
	`c 30`
	`d 40`
	`dtype: int64`

- Ví dụ với DataFrame

Mã lệnh	*Kết quả*

• *Truy cập giá trị đơn:*
`import pandas as pd`
`# Tạo DataFrame mẫu`
`df = pd.DataFrame({ 'A': [1, 2, 3],`
`'B': [4, 5, 6],`
`'C': [7, 8, 9]`
`})`
	`A B C`
	`0 1 4 7`
	`1 2 5 8`
	`2 3 6 9`

`# Lấy giá trị ở hàng thứ 2, cột thứ 1 (0-based index)` `print(df.iloc[1, 0])`	`2`

• *Truy cập một hàng:*
`# Lấy toàn bộ hàng thứ 1`
`print(df.iloc[0])`
	`A 1`
	`B 4`
	`C 7`
	`Name: 0,`
	`dtype: int64`

• *Truy cập một cột:*
`# Lấy toàn bộ cột thứ 2`
`print(df.iloc[:, 1])`
	`0 4`
	`1 5`
	`2 6`
	`Name: B,`
	`dtype: int64`

• *Truy cập nhiều hàng hoặc cột bằng slicing:*
`# Lấy hàng từ 0 đến 2 (không bao gồm hàng thứ 2) và tất cả các cột`
`print(df.iloc[0:2, :])`
	`A B C`
	`0 1 4 7`
	`1 2 5 8`

• *Sử dụng danh sách chỉ số:*
`# Lấy cột thứ 0 và thứ 2 của hàng thứ 1 và thứ 2`
	`A C`
	`1 2 8`
	`2 3 9`

```
print(df.iloc[[1, 2], [0, 2]])
```

	A	B	C
1	2	5	8
2	3	6	9

• *Cập nhật giá trị:*

```
# Thay đổi giá trị tại hàng thứ 0, cột thứ 1
df.iloc[0, 1] = 10
print(df)
```

	A	B	C
0	1	10	7
1	2	5	8
2	3	6	9

- Lưu ý
  - *Chỉ số ngoài phạm vi:* Nếu cố truy cập một chỉ số ngoài phạm vi của Series, sẽ xuất hiện lỗi IndexError.
  - *Dữ liệu nhiều chiều:*
    - Khi sử dụng Series với MultiIndex (Series nhiều cấp), iloc vẫn truy cập theo thứ tự số nguyên chứ không phụ thuộc vào mức độ chỉ số.
    - *DataFrame:*
      - Hoạt động trên DataFrame nhiều chiều: iloc không phụ thuộc vào nhãn, mà chỉ dựa vào vị trí.
      - Tương thích với slicing: có thể sử dụng slicing (:) để lấy nhiều hàng hoặc cột cùng một lúc.

2.3. MỘT SỐ HÀM THƯỜNG DÙNG TRONG PANDAS

2.3.1. pandas.concat()

- Được sử dụng để nối (*concatenate* hay hợp nhất - *merge*) các `DataFrame`, `Series`, hoặc các đối tượng tương tự theo chiều dọc (hàng, `axis=0`) hoặc chiều ngang (cột, `axis=1`). Kết quả trả về là một đối tượng pandas mới (`Series` hoặc `DataFrame`).
- Cú pháp:

```
pandas.concat(objs, axis=0, join='outer', ignore_index=False,
                                    keys=None, levels=None, names=None,
                          verify_integrity=False, sort=False, copy=True)
```

trong đó:

- ***objs***: `list` (hoặc `tuple`) các đối tượng pandas (`DataFrame` hoặc `Series`) cần ghép.
- ***axis***: Xác định trục ghép:
  - `axis=0` (mặc định): Nối theo chiều dọc (tức là ghép hàng).
  - `axis=1`: Nối theo chiều ngang (tức là ghép cột).
- ***join***:
  - `'outer'` (mặc định): Hợp nhất toàn bộ các chỉ mục.
  - `'inner'`: Chỉ giữ các chỉ mục chung.
- ***ignore_index***: Nếu `True`, bỏ qua chỉ số ban đầu và tạo chỉ số mới cho kết quả.
- ***keys***: Sử dụng một chuỗi (hoặc danh sách) làm nhãn để phân biệt các phần trong kết quả ghép.
- ***sort***: Nếu `True`, sắp xếp chỉ mục theo thứ tự tăng dần.

- Ví dụ

Mã lệnh	*Kết quả*

• *Nối các DataFrame theo hàng (mặc định axis=0):*
`import pandas as pd`
`# Tạo các DataFrame`
`df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})`
`df2 = pd.DataFrame({'A': [5, 6], 'B': [7, 8]})`
`print('df1:\n',df1)`
`print('df2:\n',df2)`
	`df1:`
	`  A  B`
	`0  1  3`
	`1  2  4`
	`df2:`
	`  A  B`
	`0  5  7`
	`1  6  8`

`# Nối dọc (ghép hàng)`
`result = pd.concat([df1, df2])`
`print(result)`
	`  A  B`
	`0  1  3`
	`1  2  4`
	`0  5  7`
	`1  6  8`

• *Nối các DataFrame theo cột (axis=1):*
`result = pd.concat([df1, df2], axis=1)`
`print(result)`
	`  A  B  A  B`
	`0  1  3  5  7`
	`1  2  4  6  8`

• *Sử dụng ignore_index:*
`result = pd.concat([df1, df2], ignore_index=True)`
`print(result)`
	`  A  B`
	`0  1  3`
	`1  2  4`
	`2  5  7`
	`3  6  8`

• *Nối với keys để phân biệt các phần trong DataFrame:*
`result = pd.concat([df1, df2], keys=['df1', 'df2'])`
`print(result)`
	`# Kết quả:`
	`#  A  B`
	`# df1 0  1  3`
	`#  1  2  4`
	`# df2 0  5  7`
	`#  1  6  8`

• *Sử dụng join='inner' (chỉ ghép cột chung):*
`df3 = pd.DataFrame({'A': [9, 10], 'C': [11, 12]})`
`result = pd.concat([df1, df3], join='inner',`
`ignore_index=True)`
`print(result)`
	`# Kết quả:`
	`#  A`
	`# 0  1`
	`# 1  2`
	`# 2  9`
	`# 3 10`

2.3.2. pandas.date_range()

- Được sử dụng để tạo ra một dãy các mốc thời gian (*dates*) hoặc khoảng thời gian (*timestamps*) theo quy tắc do người dùng xác định. Hàm trả về một đối tượng DatetimeIndex, chứa các mốc thời gian được tạo
- Đây là công cụ hữu ích trong việc phân tích dữ liệu theo thời gian hoặc xây dựng các bộ dữ liệu có tính chất thời gian.
- Cú pháp:

```
pandas.date_range(start=None, end=None, periods=None,
                  freq='D', tz=None, closed=None)
```

trong đó:

- ***start***: Ngày bắt đầu (chuỗi, `datetime` hoặc `Timestamp`).
- ***end***: Ngày kết thúc (chuỗi, `datetime` hoặc `Timestamp`).
- ***periods***: Số lượng mốc thời gian cần tạo ra (không dùng `periods` cùng với `end` hoặc `start`).
- ***freq***: Tần suất giữa các mốc thời gian

▫ `'D'`: Ngày	▫ `'H'`: Giờ	▫ `'T'` hoặc `'min'`: Phút
▫ `'S'`: Giây	▫ `'W'`: Tuần	▫ `'M'`: Tháng

- ***tz***: Múi giờ (tùy chọn).

- ***closed***: Xác định xem có bao gồm các mốc đầu và cuối không (`'left'`, `'right'`, hoặc mặc định là cả hai).

- Ví dụ

Mã lệnh	*Kết quả*

• *Tạo dãy ngày đơn giản:*
`import pandas as pd`
`# Tạo dãy ngày từ 1/1/2023 đến 10/1/2023`
`dates = pd.date_range(start='2023-01-01', end='2023-01-10')`
`print(dates)`
	`DatetimeIndex(['2023-01-01', '2023-01-02', ..., '2023-01-10'], dtype='datetime64[ns]', freq='D')`

• *Tạo dãy mốc thời gian với số lượng chỉ định:*
`# Tạo dãy 5 mốc thời gian từ ngày 1/1/2023`
`dates = pd.date_range(start='2023-01-01', periods=5, freq='D')`
`print(dates)`
	`DatetimeIndex(['2023-01-01', '2023-01-02', ..., '2023-01-05'], dtype='datetime64[ns]', freq='D')`

• *Tạo dãy mốc thời gian với khoảng cách là giờ:*
`# Tạo mốc thời gian cách nhau 2 giờ`
`dates = pd.date_range(start='2023-01-01', periods=6, freq='2H')`
`print(dates)`
	`DatetimeIndex(['2023-01-01 00:00:00', '2023-01-01 02:00:00', ..., '2023-01-01 10:00:00'], dtype='datetime64[ns]', freq='2H')`

• *Dãy thời gian với múi giờ:*
`# Tạo dãy mốc thời gian với múi giờ 'Asia/Tokyo'`
`dates = pd.date_range(start='2023-01-01', periods=3, freq='D', tz='Asia/Tokyo')`
`print(dates)`
	`DatetimeIndex(['2023-01-01 00:00:00+09:00', '2023-01-02 00:00:00+09:00', '2023-01-03 00:00:00+09:00'], dtype='datetime64[ns, Asia/Tokyo]', freq='D')`

• *Dãy thời gian theo tháng:*
`''' Tạo mốc thời gian là ngày cuối tháng trong năm 2023'''`
`dates = pd.date_range(start='2023-01-01', end='2023-12-31', freq='M')`
`print(dates)`
	`DatetimeIndex(['2023-01-31', '2023-02-28', ..., '2023-12-31'], dtype='datetime64[ns]', freq='M')`

2.3.3. pandas.to_datetime()

- Được sử dụng để chuyển đổi các giá trị như chuỗi (`string`), số, hoặc đối tượng `datetime` không đồng nhất thành đối tượng `Datetime` chuẩn của pandas.
- Kết quả đầu ra:
  - Trả về một đối tượng `DatetimeIndex` nếu xử lý `Series` hoặc `DataFrame`.
  - Trả về kiểu dữ liệu `datetime64` nếu xử lý một giá trị đơn lẻ.
- Cú pháp

```
pandas.to_datetime(arg, format=None, errors='raise', utc=False,
                   exact=True, dayfirst=False, yearfirst=False)
```

trong đó:

- ***arg***: Dữ liệu đầu vào (chuỗi, số, danh sách, hoặc `Series`).
- ***format***: Định dạng cụ thể của thời gian (vd: `%Y-%m-%d`), giúp tối ưu hóa chuyển đổi.
- ***errors***:
  - ***'raise'***: Báo lỗi nếu có giá trị không hợp lệ (mặc định).
  - ***'coerce'***: Chuyển giá trị không hợp lệ thành `NaT` (`Not a Time`).
  - ***'ignore'***: Bỏ qua giá trị không hợp lệ.
- ***utc***: Nếu **`True`**, chuyển đổi sang thời gian `UTC`.

- ***dayfirst***: Nếu **True**, ưu tiên ngày trước tháng (vd: `dd-mm-yyyy`).
- ***yearfirst***: Nếu **True**, ưu tiên năm trước (vd: `yyyy-mm-dd`).

- Ví dụ

Mã lệnh	*Kết quả*

• *Chuyển đổi chuỗi thời gian đơn giản:*
`import pandas as pd`
`date = pd.to_datetime("2023-01-01")`
`print(date)`
	`2023-01-01 00:00:00`

• *Chuyển đổi danh sách các chuỗi thời gian:*
`dates = pd.to_datetime(["2023-01-01", "2023-02-15", "2023-03-20"])`
`print(dates)`
	`DatetimeIndex(['2023-01-01', '2023-02-15', '2023-03-20'], dtype='datetime64[ns]', freq=None)`

• *Sử dụng định dạng thời gian cụ thể:*
`dates= pd.to_datetime(["01-01-2023","15-02-2023"],format="%d-%m-%Y")`
`print(dates)`
	`DatetimeIndex(['2023-01-01', '2023-02-15'], dtype='datetime64[ns]', freq=None)`

• *Xử lý lỗi với errors='coerce':*
`dates = pd.to_datetime(["2023-01-01", "invalid_date"], errors='coerce')`
`print(dates)`
	`DatetimeIndex(['2023-01-01', 'NaT'], dtype='datetime64[ns]', freq=None)`

• *Chuyển đổi với múi giờ UTC:*
`date = pd.to_datetime("2023-01-01", utc=True)`
`print(date)`
	`2023-01-01 00:00:00+00:00`

- Ứng dụng thực tế của `pandas.to_datetime()`
  - *Chuẩn hóa dữ liệu thời gian*: Chuyển đổi dữ liệu không đồng nhất thành dạng thời gian chuẩn để phân tích.
  - *Xử lý giá trị lỗi*: Dùng `errors='coerce'` để làm sạch dữ liệu thời gian không hợp lệ.
  - *Tạo tập dữ liệu thời gian:* Kết hợp với `date_range()` để xây dựng chuỗi dữ liệu theo thời gian.

3. PANDAS DATA SERIES[1]

3.1. Giới thiệu

Có thể hình dung `Series` như một mảng một chiều có nhãn (*labeled one-dimensional array*), có khả năng chứa bất kỳ kiểu dữ liệu nào (số nguyên, chuỗi, số thực, đối tượng Python, v.v.).

Series có thể chứa các kiểu dữ liệu khác nhau và có nhãn (index) cho mỗi phần tử, giúp truy xuất dữ liệu dễ dàng.

Cấu trúc của 1 Series gồm 2 thành phần chính:

(i).- ***Dữ liệu (Data):*** Một mảng các giá trị (thường là `NumPy ndarray`).

(ii).- ***Chỉ mục (Index):*** Một mảng các nhãn liên kết với từng phần tử dữ liệu. Chỉ mục này không nhất thiết phải là số nguyên; nó có thể là chuỗi, ngày tháng, hoặc bất kỳ kiểu dữ liệu băm (`hashable`) nào. Nếu chỉ mục không được cung cấp một cách rõ ràng, Pandas sẽ tự động tạo một chỉ mục số nguyên mặc định bắt đầu từ 0 (0, 1, 2, ...).

Đặc điểm chính của `pandas.Series`:

(i).- ***Một chiều (1D):*** Chỉ có một trục dữ liệu.

(ii).- ***Có nhãn (Labeled):*** Mỗi phần tử dữ liệu được liên kết với một nhãn trong chỉ mục.

(iii).- ***Đồng nhất kiểu dữ liệu*** (*Homogeneous Data*): Mặc dù một `Series` có thể chứa các kiểu dữ liệu hỗn hợp (ví dụ: `dtype: object`), nhưng thông thường và hiệu quả nhất là khi tất cả các phần tử trong `Series` có cùng một kiểu dữ liệu (ví dụ: tất cả đều là số nguyên, hoặc tất cả đều là chuỗi).

(iv).- ***Kích thước không thay đổi*** (*Size-immutable*): Kích thước của một `Series` thường không thay đổi sau khi tạo. Để thêm hoặc bớt phần tử, người ta thường tạo một `Series` mới.

(v).- ***Giá trị có thể thay đổi*** (*Value-mutable*): có thể thay đổi giá trị của các phần tử trong `Series`.

(vi).- ***Tương tự NumPy ndarray:*** `Series` hoạt động rất giống với mảng `NumPy`, và nhiều hàm của `NumPy` có thể được áp dụng trực tiếp lên `Series`. Thực tế, dữ liệu bên trong `Series` thường được lưu trữ dưới dạng một `NumPy array`.

(vii).- ***Tích hợp nhiều phương thức:*** Cung cấp nhiều phương thức tiện lợi để thực hiện các thao tác trên dữ liệu, như tính toán thống kê, xử lý chuỗi, xử lý ngày tháng, xử lý dữ liệu thiếu, v.v.

[1] Pandas Data Series: Exercises, Practice, Solution

3.2. Tạo series

Sử dụng hàm pandas.Series

3.2.1. Từ list

Nếu dữ liệu là 1 `list` thì độ dài của `list` và độ dài của mảng `index` phải bằng nhau.

Mã lệnh	*Kết quả*

`import pandas as pd`
`data_list = [10, 20, 30, 40, 50]`
`s_from_list = pd.Series(data_list)`
`print(s_from_list)`
	`# chỉ mục mặc định được tạo`
	`0  10`
	`1  20`
	`2  30`
	`3  40`
	`4  50`
	`dtype: int64`

`index_labels = ['a', 'b', 'c', 'd', 'e']`
`s_with_custom_index = pd.Series(data_list,`
`index=index_labels)`
`print(s_with_custom_index)`
	`#Chỉ định chỉ mục`
	`a  10`
	`b  20`
	`c  30`
	`d  40`
	`e  50`
	`dtype: int64`

3.2.2. Từ NumPy array

Mã lệnh	*Kết quả*

`import numpy as np`
`data_numpy = np.array([1.1, 2.2, 3.3, 4.4])`
`s_from_numpy=pd.Series(data_numpy, index=['w','x','y','z'])`
`print(s_from_numpy)`
	`w  1.1`
	`x  2.2`
	`y  3.3`
	`z  4.4`
	`dtype: float64`

3.2.3. Từ Dictionary

- Các khóa (`keys`) của dictionary sẽ trở thành chỉ mục (`index`).
- Các giá trị (`values`) của `dictionary` sẽ trở thành dữ liệu của `Series`.
- Thứ tự các phần tử trong `Series` sẽ theo thứ tự chèn vào `dictionary` hoặc có thể chỉ định một `index` riêng để sắp xếp lại hoặc chọn lọc.

Mã lệnh	*Kết quả*

`data_dict={'name':'Alice', 'age':30, 'city':'New York'}`
`s_from_dict = pd.Series(data_dict)`
`print(s_from_dict)`
	`name  Alice`
	`age  30`
	`city  New York`
	`dtype: object`

Khi tạo `Series` từ dictionary và được cung cấp một `index` nó sẽ chọn các giá trị tương ứng với các khóa trong index. Nếu một khóa trong `index` không có trong `dictionary`, giá trị sẽ là `NaN` (*Not a Number*).

Mã lệnh	*Kết quả*

`s_from_dict_with_index = pd.Series(data_dict,`
`index=['age', 'city', 'country'])`
`print(s_from_dict_with_index)`
	`age  30`
	`city  New York`
	`country  NaN`
	`dtype: object`

3.2.4. Từ một giá trị vô hướng (Scalar value)

Nếu dữ liệu là một giá trị vô hướng, cần phải cung cấp một chỉ mục. Giá trị này sẽ được lặp lại để khớp với độ dài của chỉ mục.

Mã lệnh	*Kết quả*

`s_from_scalar = pd.Series(5, index=['a', 'b', 'c'])`
`print(s_from_scalar)`
	`a  5`
	`b  5`
	`c  5`

	`dtype: int64`

3.3. Truy xuất phần tử dựa trên index

- Theo nhãn chỉ mục: `my_series['label']`
- Theo vị trí số nguyên: `my_series[0]` (nếu chỉ mục là mặc định hoặc không trùng với nhãn số) hoặc sử dụng `my_series.iloc[0]`.
- Truy cập nhiều phần tử: `my_series[['label1', 'label2']]`
  hoặc `my_series.iloc[[0, 2]]`.
- Slicing (cắt lát):
  `my_series['start_label':'end_label']` (bao gồm cả end_label)
  hoặc `my_series[start_pos:end_pos]` (không bao gồm end_pos).

Một số ví dụ

Mã lệnh	*Kết quả*

`import pandas as pd`
`import numpy as np`
`D = {'a': 90, 'b':np.NaN, 'd':-80, 'f':70, 'z':60}`
`S = pd.Series(D, index=['a','c','b','d','e','f'])`
`print(S)`
	`a  90.0`
	`c  NaN`
	`b  NaN`
	`d  -80.0`
	`e  NaN`
	`f  70.0`
	`dtype: float64`

`print("S[\'a\']=",S['a'])`	`S['a']= 90.0`
`print("S[\'b\']=",S['b'])`	`S['b']= nan`

`print("S[:\'d\']=",S[:'d'])`	`S[:'d']=`
	`a  90.0`
	`c  NaN`
	`b  NaN`
	`d  -80.0`
	`dtype: float64`

`print("S[\'d\':]=",S['d':])`	`S['d':]=`
	`d  -80.0`
	`e  NaN`
	`f  70.0`
	`dtype: float64`

`print("Từ index=-3 đến cuối:\n",S[-3:])`	`Từ index=-3 đến cuối:`
	`d  -80.0`
	`e  NaN`
	`f  70.0`
	`dtype: float64`

3.4. Chuyển đổi từ series sang đối tượng khác

(i).- Sử dụng thuộc tính .values hoặc phương thức .to_numpy() hay `asarray` để chuyển Series thành NumPy array.

Mã lệnh	*Kết quả*

`import numpy as np`
`S=pd.Series(list(range(5,10)))`
`print(S)`
	`0  5`
	`1  6`
	`2  7`
	`3  8`
	`4  9`
	`dtype: int64`

`A1 = S.`**`values`** `print(A1)`	`[5 6 7 8 9]`
`A2 = S.`**`to_numpy()`** `print(A2)`	`[5 6 7 8 9]`
`A3 = `**`np.asarray`**`(S)` `print(A3)`	`[5 6 7 8 9]`

(ii).- Chuyển sang DataFrame của Pandas

Mã lệnh	*Kết quả*

`import numpy as np`
`S=pd.Series(list(range(5,10)))`
`print(S)`
	`0  5`
	`1  6`
	`2  7`
	`3  8`
	`4  9`
	`dtype: int64`

`df = S.to_frame(name='values')`
`print(df)`
	`  values`
	`0  5`
	`1  6`
	`2  7`
	`3  8`
	`4  9`

(iii).- Chuyển sang danh sách (list)

Mã lệnh	*Kết quả*

`import numpy as np`
`S=pd.Series(list(range(5,10)))`
`print(S)`
	`0  5`
	`1  6`
	`2  7`
	`3  8`
	`4  9`
	`dtype: int64`

`L = S.tolist()` `print(L)`	`[5, 6, 7, 8, 9]`

(iv).- Chuyển sang dictionary

Mã lệnh	*Kết quả*

`import numpy as np`
`S=pd.Series(list(range(5,10)))`
`print(S)`
	`0  5`
	`1  6`
	`2  7`
	`3  8`
	`4  9`
	`dtype: int64`

`D = S.to_dict()` `print(D)`	`{0: 5, 1: 6, 2: 7, 3: 8, 4: 9}`

(v).- Chuyển sang string

Mã lệnh	*Kết quả*

`import numpy as np`
`S=pd.Series(list(range(5,10)))`
`print(S)`
	`0  5`
	`1  6`
	`2  7`
	`3  8`
	`4  9`
	`dtype: int64`

`Str = S.to_string()`
`print(Str)`
	`0  5`
	`1  6`
	`2  7`
	`3  8`
	`4  9`

3.5. Các phép toán dựa trên vector (Vectorized)

Có thể thực hiện các phép toán số học, logic trực tiếp trên Series, và chúng sẽ được áp dụng cho từng phần tử (giống NumPy).

Ví dụ:

Mã lệnh	*Kết quả*

`s1 = pd.Series([1, 2, 3])`
`s2 = pd.Series([10, 20, 30])`
`print(f's1 + s2 =\n{s1 + s2}')`
	`s1 + s2 =`
	`0  11`
	`1  22`
	`2  33`
	`dtype: int64`

`print(f's1 * 2 =\n{s1 * 2}')`	`s1 * 2 =`
	`0  2`
	`1  4`
	`2  6`

	dtype: int64
print(f's1 > s2 =\n{s1 > s2}')	s1 > s2 = 0 False 1 False 2 False dtype: bool

3.6. Thuộc tính thường dùng

- ***dtype*** : Trả về kiểu dữ liệu của các phần tử trong `Series`.
- ***hasnans*** : Kiểm tra xem `Series` có chứa giá trị `NaN` nào không.
- ***index*** : Trả về hoặc thiết lập chỉ số của `Series`. Khi trả về, kết quả là đối tượng `index`
- ***name*** : Trả về hoặc thiết lập tên của `Series`. Nếu `Series` được lấy từ `DataFrame` thì tên này chính là tên cột của `DataFrame` đã cung cấp dữ liệu cho `Series`.
- ***shape*** : Trả về một tuple chứa kích thước (số phần tử).
- ***size*** : Trả về số lượng phần tử.
- ***values*** : Trả về mảng các giá trị của `Series` dưới dạng `NumPy array`

3.7. Phương thức thường dùng

3.7.1. apply()

- Được sử dụng để áp dụng một hàm (function) của người dùng lên từng phần tử của một `Series`.
- Khác biệt với map(): Trong khi `map()` chỉ hoạt động trên từng giá trị, `apply()` có thể xử lý cả hàm yêu cầu logic phức tạp và thường áp dụng cho các hàm yêu cầu nhiều đối số hơn.
- Cú pháp

```
Series.apply(func, convert_dtype=True, args=())
```

  - ***func***: Hàm được áp dụng cho từng giá trị.
  - ***convert_dtype***: (Tùy chọn) Nếu True, tự động chuyển đổi kiểu dữ liệu của kết quả.
  - ***args***: Các tham số bổ sung có thể truyền vào hàm.
- Ví dụ

Mã lệnh	*Kết quả*

• *Sử dụng hàm đơn giản:*
`import pandas as pd`
`# Tạo Series`
`s = pd.Series([1, 2, 3, 4, 5])`
`# Định nghĩa hàm để tính bình phương`
`def square(x):`
`  return x ** 2`
`# Áp dụng hàm square`
`result = s.apply(square)`
`print(result)`
	0 1
	1 4
	2 9
	3 16
	4 25
	dtype: int64

• *Sử dụng hàm lambda:*
`# Áp dụng hàm lambda để nhân đôi giá trị`
`result = s.apply(lambda x: x * 2)`
`print(result)`
	0 2
	1 4
	2 6
	3 8
	4 10
	dtype: int64

• *Sử dụng hàm với tham số bổ sung:*	0 11 1 12

```
# Định nghĩa hàm có tham số bổ sung
def add_value(x, value):
    return x + value
# Áp dụng hàm với tham số bổ sung
result = s.apply(add_value, args=(10,))
print(result)
```

```
2    13
3    14
4    15
dtype: int64
```

- *Xử lý dữ liệu dạng chuỗi:*

```
# Tạo Series chứa chuỗi
s_string = pd.Series(['python', 'pandas', 'data'])
# Áp dụng hàm lambda để viết hoa
result = s_string.apply(lambda x: x.upper())
print(result)
```

```
0    PYTHON
1    PANDAS
2      DATA
dtype: object
```

- Ứng dụng thực tế
  - *Xử lý và làm sạch dữ liệu:* Áp dụng các hàm của người dùng để làm sạch hoặc chuẩn hóa dữ liệu trong Series.
  - *Chuyển đổi dữ liệu:* Tính toán hoặc thay đổi dữ liệu để phù hợp với phân tích tiếp theo.
  - *Phân tích dữ liệu:* Áp dụng các hàm phức tạp như phân tích logic hoặc tính toán của người dùng.

3.7.2. argmax() / argmin() hoặc idmax() / idmin()

- ***Giới thiệu***
  - `Series.argmax` duyệt qua tất cả các giá trị trong `Series` đó và trả về nhãn chỉ mục (`index label`) của phần tử có giá trị lớn nhất (đối với `argmax`) và của phần tử có giá trị nhỏ nhất (đối với `argmin`).
  - Nếu có nhiều phần tử cùng có giá trị lớn nhất, `argmax()` sẽ trả về nhãn chỉ mục của lần xuất hiện đầu tiên. Tương tự cho `argmin`
  - Theo mặc định, `argmax()` và `argmin` bỏ qua các giá trị `NaN` (Not a Number) khi tìm giá trị lớn nhất (hành vi này được kiểm soát bởi tham số `skipna=True`). Nếu tất cả các giá trị là `NaN`, nó sẽ gây ra lỗi `ValueError`.
  - `Argmax()` và `argmin()` trả trả về tên nhãn của chỉ mục, còn `idmax()` và `idmin()` trả về vị trí số nguyên (bắt đầu từ `0`) bất kể nhãn `index` có kiểu dữ liệu là gì.
  - Lưu ý `pandas.Series` không có 2 phương thức `idmax()` và `idmin()`.
- ***Cú pháp***

```
Series.argmax(axis=None, skipna=True, *args, **kwargs)
Series.argmin(axis=None, skipna=True, *args, **kwargs)
```

Trong đó:

- ***axis***: Tham số này thường không được sử dụng cho `Series.argmax()` vì `Series` chỉ có một chiều. Nó được giữ lại để tương thích.
- ***skipna*** (mặc định là ***True***):
  - *True*: Bỏ qua các giá trị `NaN` khi tìm kiếm.
  - *False*: Nếu có `NaN`, kết quả có thể không như mong đợi hoặc gây lỗi tùy thuộc vào phiên bản Pandas và cách `NumPy` xử lý `argmax` với `NaN`. Thông thường, nên để `skipna=True`.

- ***Ví dụ***

  - *Với* `Series`

Mã lệnh	*Kết quả*

`import pandas as pd`
`data1=[5,19,2,6,-8]`
`S = pd.Series(data1)`
`print(S.argmax())`
`print(S.argmin())`
	
	1
	4

`data2=['F','K','Z','B','L']`
`S = pd.Series(data2)`
`print(S.argmax())`
`print(S.argmin())`
	
	2
	3

  - *Với* `DataFrame`

Cho `dataframe` sau:

```
import pandas as pd
import numpy as np
# Tạo DataFrame mẫu
data = {'Sản phẩm': ['A', 'B', 'C',
                     'D', 'E'],
        'Doanh số': [150, 200, 180,
                     200, np.nan],
        'Lợi nhuận': [20, 35, 25,
                      35, 10]}
df = pd.DataFrame(data, index=['Q1',
            'Q2', 'Q3', 'Q4', 'Q5'])
print("DataFrame gốc:")
print(df)
```

DataFrame gốc:

	Sản phẩm	Doanh số	Lợi nhuận
Q1	A	150.0	20
Q2	B	200.0	35
Q3	C	180.0	25
Q4	D	200.0	35
Q5	E	NaN	10

```
    # 1. Tìm nhãn chỉ mục của doanh số cao nhất
    idx_max_doanhso = df['Doanh số'].argmax()
    # Output: 'Q2' (vì 200 ở Q2 xuất hiện trước 200 ở Q4)
    print(f"Nhãn chỉ mục của doanh số cao nhất: '{idx_max_doanhso}'")
    print(f"Giá trị doanh số tại chỉ mục '{idx_max_doanhso}':
                                        {df.iloc[idx_max_doanhso, 1]}")
    '''Kết quả:
    Nhãn chỉ mục của doanh số cao nhất: '1'
    Giá trị doanh số tại chỉ mục '1': 200.0    '''
    # 2. Tìm nhãn chỉ mục của lợi nhuận cao nhất
    idx_max_loinhuan = df['Lợi nhuận'].argmax()
    print(f"Nhãn chỉ mục của lợi nhuận cao nhất: '{idx_max_loinhuan}'")
    print(f"Giá trị lợi nhuận tại chỉ mục '{idx_max_loinhuan}':
    {df.iloc[idx_max_loinhuan, 2]}")
    '''Kết quả:
    Nhãn chỉ mục của lợi nhuận cao nhất: '1'
    Giá trị lợi nhuận tại chỉ mục '1': 35    '''
```

  - *Với DataFrame có index là số nguyên mặc định*

```
    data2 = {'Điểm': [8, 9, 7, 9, 6]}
    df2 = pd.DataFrame(data2)
    print("DataFrame 2:")
    print(df2)
    idx_max_diem = df2['Điểm'].argmax()
    print(f"Nhãn chỉ mục (là số) của điểm cao nhất: {idx_max_diem}")
    # Output: 1 (vì 9 ở vị trí index 1 xuất hiện trước)
    print(f"Giá trị điểm tại chỉ mục {idx_diem_cao_nhat}:
                                        {df2.loc[idx_max_diem, 'Điểm']}")
```

```
'''Kết quả:
Nhãn chỉ mục (là số) của điểm cao nhất: 1
Giá trị điểm tại chỉ mục 1: 9    '''
```

3.7.3. bfill() hoặc ffill()

- Được sử dụng để điền các giá trị thiếu (`NaN`, `None`).
- Hai phương thức này rất hữu ích khi muốn điền giá trị thiếu dựa trên các giá trị lân cận: như trong dữ liệu chuỗi thời gian, nơi giá trị hiện tại có thể được suy đoán từ giá trị trước đó hoặc sau đó.
- **`ffill`** (*forward-fill*): Điền giá trị `NaN` bằng giá trị hợp lệ đứng ngay phía trước nó.
- **`bfill`** (*backward-fill*): Điền giá trị `NaN` bằng giá trị hợp lệ đứng ngay phía sau nó.
- Ví dụ

Mã lệnh	*Kết quả*

`import pandas as pd`
`import numpy as np`
`s = pd.Series([1, np.nan, np.nan, 4, 5, np.nan])`
`print("s gốc=\n",s)`
	`s gốc=`
	`0  1.0`
	`1  NaN`
	`2  NaN`
	`3  4.0`
	`4  5.0`
	`5  NaN`
	`dtype: float64`

`print("s.ffill() với imit=1:")`
`s_ffilled = s.ffill(limit=1)`
`print(s_ffilled)`
	`s.ffill() với imit=1`
	`0  1.0`
	`1  **1.0**`
	`2  **NaN**`
	`3  4.0`
	`4  5.0`
	`5  5.0`
	`dtype: float64`

`print("Fillna trên s với phương thức s.ffill():")`
`s_ffilled = s.ffill()`
`print(s_ffilled)`
	`Fillna trên s với phương thức s.ffill():`
	`0  1.0`
	`1  **1.0**`
	`2  **1.0**`
	`3  4.0`
	`4  5.0`
	`5  5.0`
	`dtype: float64`

`print("Fillna trên s1 với phương thức s.bfill():")`
`s_bfilled = s.bfill()`
`print(s_bfilled)`
	`Fillna trên s với phương thức s.bfill():`
	`0  1.0`
	`1  **4.0**`
	`2  **4.0**`
	`3  4.0`
	`4  5.0`
	`5  **NaN**`
	`dtype: float64`

3.7.4. describe()

- Được sử dụng để tạo ra các thống kê mô tả cho một đối tượng `Series`.
- Kết quả trả về phụ thuộc vào kiểu dữ liệu của `Series` đó.
  - *Đối với dữ liệu kiểu số* (*Numeric Data:* `integers`, `floats`): sẽ trả về một `Series` mới chứa các giá trị sau:
    - ***`count`*** : Tổng số lượng các giá trị không rỗng (non-null).
    - ***`mean`*** : Giá trị trung bình của các giá trị.

- ***std*** : Độ lệch chuẩn (standard deviation), cho biết mức độ phân tán của các giá trị.
- ***min*** : Giá trị nhỏ nhất trong `Series`.
- ***25%*** : Giá trị phân vị thứ 25 (hay còn gọi là tứ phân vị thứ nhất - Q1).
- ***50%*** : Giá trị phân vị thứ 50 (hay còn gọi là trung vị - median).
- ***75%*** : Giá trị phân vị thứ 75 (hay còn gọi là tứ phân vị thứ ba - Q3).
- ***max*** : Giá trị lớn nhất trong `Series`.

- **Đối với dữ liệu kiểu đối tượng** (Object Data như: strings, hoặc timestamps): sẽ trả về một `Series` mới chứa các giá trị sau:
  - ***count*** : Tổng số lượng các giá trị không rỗng.
  - ***unique*** : Số lượng các giá trị duy nhất (không trùng lặp).
  - ***top*** : Giá trị xuất hiện nhiều nhất (phổ biến nhất).
  - ***freq*** : Tần suất xuất hiện của giá trị `top`.

Nếu `Series` chứa dữ liệu kiểu `timestamps`, ngoài các giá trị trên, `describe()` cũng có thể bao gồm:

- ***first*** : Giá trị `timestamp` sớm nhất.
- ***last*** : Giá trị `timestamp` muộn nhất.

- **Tùy chỉnh Percentiles:**

Có thể tùy chỉnh các giá trị phân vị được trả về bằng cách sử dụng tham số `percentiles`. Tham số này nhận vào một danh sách các số từ 0 đến 1.

Ví dụ, `seriesName.describe(percentiles=[.1, .9])` sẽ trả về phân vị thứ 10 và 90 cùng với các thống kê mặc định khác (thay thế cho các phân vị `25%, 50%, 75%` mặc định).

3.7.5. dropna()

- **Giới thiệu:**

- Phương thức series.dropna() được sử dụng để loại bỏ các giá trị bị thiếu (missing values) khỏi một đối tượng Series. Giá trị bị thiếu thường được biểu thị bằng NaN (Not a Number) hoặc None.
- Phương thức này giúp làm sạch dữ liệu bằng cách loại bỏ các quan sát (phần tử) không hoàn chỉnh trong Series.

- **Giá trị trả về**

- Mặc định, series.dropna() sẽ trả về một **Series mới** không chứa các giá trị NaN. Series gốc sẽ không bị thay đổi.
- Nếu đặt tham số inplace=True, phương thức sẽ sửa đổi trực tiếp Series gốc và trả về None.

- **Cú pháp**

```
series.dropna( axis=0, inplace=False, how=None, **kwargs)
```

trong đó:

- **axis**:

- **Kiểu dữ liệu**: {0 hoặc 'index'}.
- **Mô tả**: Chỉ định trục mà từ đó các giá trị NaN sẽ bị loại bỏ.
- **Lưu ý**: Đối với một đối tượng Series, chỉ có một trục (trục 0, tương ứng với các chỉ mục - index). Do đó, tham số này thường được để ở giá trị mặc định là 0 và ít khi cần thay đổi khi làm việc với Series.

- **inplace**:
  - **Kiểu dữ liệu**: bool.
  - **Mô tả**:
    - Nếu False (mặc định): Phương thức sẽ trả về một bản sao của Series đã được loại bỏ các giá trị NaN. Series gốc không thay đổi.
    - Nếu True: Phương thức sẽ thực hiện việc loại bỏ trực tiếp trên Series gốc và trả về None.
- **how**:
  - **Kiểu dữ liệu**: {'any', 'all'}.
  - **Mô tả**: Tham số này quan trọng hơn đối với DataFrame (khi có nhiều cột và muốn quyết định loại bỏ một hàng/cột nếu tất cả các giá trị hoặc bất kỳ giá trị nào là NaN).
  - **Đối với Series**: Vì Series chỉ là một chiều, việc một phần tử là NaN đồng nghĩa với việc "tất cả" thông tin của phần tử đó là NaN (vì chỉ có một giá trị). Do đó, how='any' (mặc định nếu không được chỉ định) và how='all' thường cho kết quả giống nhau khi áp dụng cho một Series đơn lẻ. Về cơ bản, nếu một phần tử là NaN, nó sẽ bị loại bỏ.

- **Ví dụ**

Mã lệnh	*Kết quả*

`import pandas as pd`
`import numpy as np`
`# Tạo một Series với các giá trị NaN`
`data = pd.Series([10, 20, np.nan, 40, None, 50, np.nan])`
`print("Series ban đầu:")`
`print(data)`
	`Series ban đầu:`
	`0 10.0`
	`1 20.0`
	`2 NaN`
	`3 40.0`
	`4 NaN`
	`5 50.0`
	`6 NaN`
	`dtype: float64`

`# Loại bỏ các giá trị NaN (mặc định inplace=False)`
`data_cleaned = data.dropna()`
`print("\nSeries sau khi loại bỏ NaN:")`
`print(data_cleaned)`
	`Series sau khi loại bỏ NaN:`
	`0 10.0`
	`1 20.0`
	`3 40.0`
	`5 50.0`
	`dtype: float64`

`# Series gốc không thay đổi`
`print("\nSeries gốc sau khi gọi dropna() (inplace=False):")`
`print(data)`
	`Series gốc sau khi gọi dropna() (inplace=False):`
	`0 10.0`
	`1 20.0`
	`2 NaN`
	`3 40.0`
	`4 NaN`
	`5 50.0`
	`6 NaN`
	`dtype: float64`

`# Loại bỏ các giá trị NaN và thay đổi trực tiếp Series gốc`	`Series mới cho ví dụ inplace=True:`

`data_inplace = pd.Series([10, 20, np.nan, 40, None, 50, np.nan])`
`print("\nSeries mới cho ví dụ inplace=True:")`
`print(data_inplace)`
	`0 10.0`
	`1 20.0`
	`2 NaN`
	`3 40.0`
	`4 NaN`
	`5 50.0`
	`6 NaN`
	`dtype: float64`

`data_inplace.dropna(inplace=True)`
`print("\nSeries sau khi loại bỏ NaN (inplace=True):")`
`print(data_inplace)`
	`Series sau khi loại bỏ NaN (inplace=True):`
	`0 10.0`
	`1 20.0`
	`3 40.0`
	`5 50.0`
	`dtype: float64`

3.7.6. dt()

3.7.6.1. Giới thiệu

- `series.dt` là một thuộc tính truy cập (*accessor*) đặc biệt, tương tự như `series.str`. Nó được sử dụng để truy cập các thuộc tính và phương thức liên quan đến ngày giờ (*datetime-like properties*) của các phần tử trong một đối tượng `Series`.
- `series.dt` cung cấp một cách vector hóa để trích xuất thông tin và thực hiện các thao tác liên quan đến ngày giờ trên từng phần tử của `Series`. Giống như `series.str`, điều này giúp mã nguồn ngắn gọn, dễ đọc và hiệu quả hơn so với việc lặp thủ công.
- Công dụng của `series.dt`:
  - *Xử lý* `NaT` *(Not a Time) tự động*: Tương tự như `NaN` trong dữ liệu số hoặc chuỗi, `NaT` là giá trị bị thiếu cho dữ liệu ngày giờ. Các thuộc tính và phương thức dt thường xử lý `NaT` một cách hợp lý (ví dụ: trả về `NaN` hoặc `NaT`).
  - *Hiệu suất*: Các thao tác được vector hóa, nhanh hơn so với việc lặp thủ công.
  - *Dễ đọc và tiện lợi*: Cung cấp API rõ ràng để làm việc với các thành phần của ngày giờ.

3.7.6.2. Điều kiện tiên quyết

Để sử dụng `series.dt`, `Series` đó phải có kiểu dữ liệu là `datetime` (ví dụ: `datetime64[ns]`) hoặc `timedelta`. Nếu `Series` chứa các chuỗi biểu diễn ngày giờ, cần chuyển đổi chúng sang kiểu `datetime` trước bằng cách sử dụng `pd.to_datetime()`.

3.7.6.3. Các thuộc tính (properties) thường dùng:

- **`series.dt.date`**: Trả về một `Series` chứa đối tượng `date` (chỉ ngày, không có thông tin thời gian).
- **`series.dt.day`**: Trả về một Series chứa ngày trong tháng (`1-31`, kiểu `integer`).
- **`series.dt.dayofweek`**: Tương tự như `weekday`.
- **`series.dt.dayofyear`**: Trả về một Series chứa ngày thứ mấy trong năm (`1-365` hoặc `1-366`, kiểu `integer`).
- **`series.dt.hour`**: Trả về một `Series` chứa giờ (`0-23`, kiểu `integer`).

- **`series.dt.is_month_end`**: Trả về `Series boolean` cho biết ngày đó có phải là ngày cuối tháng không.
- **`series.dt.is_month_start`**: Trả về `Series boolean` cho biết ngày đó có phải là ngày đầu tháng không.
- **`series.dt.is_quarter_end`**: Trả về `Series boolean` cho biết ngày đó có phải là ngày cuối quý không.
- **`series.dt.is_quarter_start`**: Trả về `Series boolean` cho biết ngày đó có phải là ngày đầu quý không.
- **`series.dt.is_year_end`**: Trả về `Series boolean` cho biết ngày đó có phải là ngày cuối năm không.
- **`series.dt.is_year_start`**: Trả về `Series boolean` cho biết ngày đó có phải là ngày đầu năm không.
- **`series.dt.microsecond`**: Trả về một `Series` chứa microgiây.
- **`series.dt.minute`**: Trả về một `Series` chứa phút (`0-59`, kiểu `integer`).
- **`series.dt.month`**: Trả về một `Series` chứa tháng (1-12, kiểu `integer`).
- **`series.dt.nanosecond`**: Trả về một `Series` chứa nanogiây.
- **`series.dt.quarter`**: Trả về một Series chứa quý trong năm (`1-4`, kiểu `integer`).
- **`series.dt.second`**: Trả về một `Series` chứa giây (`0-59`, kiểu `integer`).
- **`series.dt.time`**: Trả về một `Series` chứa đối tượng `time` (chỉ thời gian, không có thông tin ngày).
- **`series.dt.tz`**: Trả về thông tin múi giờ (`timezone`) nếu có.
- **`series.dt.weekday`**: Trả về một `Series` chứa ngày trong tuần (`Thứ Hai=0, Chủ Nhật=6`, kiểu `integer`).
- **`series.dt.weekofyear`** (hoặc **`series.dt.isocalendar().week`**): Trả về một `Series` chứa tuần thứ mấy trong năm.
- **`series.dt.year`**: Trả về một `Series` chứa năm (kiểu `integer`).

3.7.6.4. Các phương thức (Methods) thường dùng:

- **`series.dt.ceil('D')`:** Làm tròn lên ngày giờ đến tần suất gần nhất.
- **`series.dt.day_name()`**: Trả về một `Series` chứa tên của ngày trong tuần (ví dụ: `'Monday', 'Tuesday'`).
- **`series.dt.floor('D')`:** Làm tròn xuống ngày giờ đến tần suất gần nhất.
- **`series.dt.month_name()`**: Trả về một Series chứa tên của tháng (ví dụ: `'January', 'February'`).
- **`series.dt.normalize()`:** Chuẩn hóa ngày giờ về nửa đêm (`00:00:00`) của ngày đó, giữ nguyên thông tin ngày.
- **`series.dt.round('D')`:** Làm tròn ngày giờ đến tần suất gần nhất (ví dụ: `'D'` cho ngày, `'H'` cho giờ).
- **`series.dt.strftime('%Y-%m-%d')`**: Định dạng ngày giờ thành chuỗi theo một định dạng cụ thể (ví dụ: `'YYYY-MM-DD'`).

- **`series.dt.to_period('M')`:** Chuyển đổi Series thành kiểu `PeriodIndex` hoặc `Series` của các đối tượng `Period` với tần suất nhất định (ví dụ: `'M'` cho tháng, `'Q'` cho quý).
- **`series.dt.tz_convert('US/Eastern')`:** Chuyển đổi `Series` ngày giờ sang một múi giờ khác.
- **`series.dt.tz_localize('UTC')`:** Gán múi giờ cho `Series` ngày giờ (nếu chưa có).

3.7.6.5. Ví dụ

Mã lệnh	*Kết quả*

`import pandas as pd`
`# Tạo một Series chứa dữ liệu ngày giờ`
`dates_str = ['2023-01-15 10:30:00',`
`'2024-05-20 14:45:30', '2025-12-01 08:00:15',`
`None]`
`# Chuyển đổi sang kiểu datetime64[ns]`
`s_dates = pd.Series(pd.to_datetime(dates_str))`
`print("Series ngày giờ ban đầu:")`
`print(s_dates)`
	`Series ngày giờ ban đầu:`
	`0 2023-01-15 10:30:00`
	`1 2024-05-20 14:45:30`
	`2 2025-12-01 08:00:15`
	`3 NaT`
	`dtype: datetime64[ns]`

`# Trích xuất năm`
`print("\nNăm:")`
`print(s_dates.dt.year)`
	`Năm:`
	`0 2023.0`
	`1 2024.0`
	`2 2025.0`
	`3 NaN`
	`dtype: float64`

`# Trích xuất tháng`
`print("\nTháng:")`
`print(s_dates.dt.month)`
	`Tháng:`
	`0 1.0`
	`1 5.0`
	`2 12.0`
	`3 NaN`
	`dtype: float64`

`# Trích xuất tên ngày trong tuần`
`print("\nTên ngày trong tuần:")`
`print(s_dates.dt.day_name())`
	`Tên ngày trong tuần:`
	`0 Sunday`
	`1 Monday`
	`2 Monday`
	`3 NaN`
	`dtype: object`

`# Trích xuất giờ`
`print("\nGiờ:")`
`print(s_dates.dt.hour)`
	`Giờ:`
	`0 10.0`
	`1 14.0`
	`2 8.0`
	`3 NaN`
	`dtype: float64`

`# Định dạng ngày giờ thành chuỗi`
`print("\nĐịnh dạng YYYY/MM/DD:")`
`print(s_dates.dt.strftime('%Y/%m/%d'))`
	`Định dạng YYYY/MM/DD:`
	`0 2023/01/15`
	`1 2024/05/20`
	`2 2025/12/01`
	`3 NaN`
	`dtype: object`

`# Kiểm tra xem có phải là ngày cuối tháng không`
`print("\nCó phải ngày cuối tháng không?")`
`print(s_dates.dt.is_month_end)`
	`Có phải ngày cuối tháng không?`
	`0 False`
	`1 False`
	`2 False`
	`3 False`
	`dtype: bool`

`# Cộng thêm 1 ngày (sử dụng pd.Timedelta)`
`print("\nCộng thêm 1 ngày:")`
`print(s_dates + pd.Timedelta(days=1))`
	`Cộng thêm 1 ngày:`
	`0 2023-01-16 10:30:00`
	`1 2024-05-21 14:45:30`
	`2 2025-12-02 08:00:15`
	`3 NaT`
	`dtype: datetime64[ns]`

`# Đối với dữ liệu timedelta` `time_deltas = pd.Series(pd.to_timedelta(`	`Series timedelta:`

```
    ['1 days 06:05:01.00003', '1 days 00:00:00',
                              '2 days 01:00:00']))
print("\nSeries timedelta:")
print(time_deltas)
```

```
0   1 days
06:05:01.000030
1          1 days
00:00:00
2          2 days
01:00:00
dtype: timedelta64[ns]
```

```
# Trích xuất số ngày từ timedelta
print("\nSố ngày từ timedelta:")
print(time_deltas.dt.days)
```

```
Số ngày từ timedelta:
0    1
1    1
2    2
dtype: int64
```

```
# Trích xuất tổng số giây từ timedelta
print("\nTổng số giây từ timedelta:")
print(time_deltas.dt.total_seconds())
```

```
Tổng số giây từ
timedelta:
0    108301.00003
1     86400.00000
2    176400.00000
dtype: float64
```

3.7.7. explode()

- Được sử dụng để "tách" các giá trị trong một `Series` chứa các `list`, `tuple`, hoặc các giá trị `iterable` khác, sao cho mỗi phần tử trong danh sách trở thành một hàng riêng biệt trong `Series`.
- Hỗ trợ các giá trị không phải `iterable`:
  - Các giá trị không phải là `list/tuple` (như số, chuỗi) sẽ được giữ nguyên.
  - Giá trị `NaN` không bị ảnh hưởng.
- Cú pháp:

`SeriesName.explode()`

- Ví dụ

Mã lệnh	*Kết quả*

• *Tách một Series chứa list:*
`import pandas as pd`
`# Tạo Series`
`s = pd.Series([[1, 2, 3], 'python', [4, 5], None])`
`# Sử dụng explode`
`result = s.explode()`
`print(result)`
	`0       1`
	`0       2`
	`0       3`
	`1  python`
	`2       4`
	`2       5`
	`3    None`
	`dtype: object`

• *Ứng dụng với giá trị không phải list:*
`s = pd.Series([[1, 2], 'pandas', 10, None])`
`result = s.explode()`
`print(result)`
	`0       1`
	`0       2`
	`1   pandas`
	`2      10`
	`3    None`
	`dtype: object`

• *Kết hợp explode với DataFrame:*
`# Tạo DataFrame`
`df = pd.DataFrame({`
`'A': [1, 2, 3],`
`'B':[['apple','banana'],['cherry'], None] })`
`print(df)`
	`   A                  B`
	`0  1  [apple, banana]`
	`1  2          [cherry]`
	`2  3              None`

`# Sử dụng explode trên cột 'B'`
`result = df.explode('B')`
`print(result)`
	`   A       B`
	`0  1   apple`
	`0  1  banana`
	`1  2  cherry`
	`2  3    None`

- Lưu ý
  - *Giữ lại chỉ số gốc:* Chỉ số gốc của `Series` sẽ được giữ nguyên sau khi "tách". Có thể sử dụng `.reset_index()` nếu cần đánh lại số thứ tự.

- *Giá trị không phải danh sách:* Nếu `Series` có giá trị không phải là list hoặc `iterable`, giá trị đó sẽ được giữ nguyên.
- *Giá trị* `None` *hoặc* `NaN`*:* Các giá trị `None` hoặc `NaN` không bị ảnh hưởng bởi `explode()` và được giữ nguyên.

- Ứng dụng thực tế của `Series.explode()`
  - *Xử lý dữ liệu phức tạp*: Tách các cột chứa danh sách (`list`) hoặc mảng để dễ dàng xử lý từng phần tử.
  - *Chuẩn hóa dữ liệu*: Làm "phẳng" dữ liệu để phù hợp với các phép toán hoặc phân tích.
  - *Tối ưu hóa dữ liệu đầu vào cho mô hình*: Đưa dữ liệu phức tạp về dạng đơn giản hơn.

3.7.8. fillna()

- **Giới thiệu**: `pandas.Series.fillna()` dùng để điền (thay thế) các giá trị bị thiếu (thường là `NaN` - *Not a Number*, hoặc `None`) trong một đối tượng `Series` bằng các giá trị được chỉ định hoặc bằng các phương pháp điền khác nhau.
- **Chức năng chính:** Phương thức này cho phép xử lý dữ liệu bị thiếu một cách linh hoạt, giúp làm sạch và chuẩn bị dữ liệu cho các bước phân tích tiếp theo.
- **Giá trị trả về:**
  - Mặc định, `fillna()` sẽ trả về một đối tượng `Series` mới đã được điền các giá trị thiếu. `Series` gốc sẽ không bị thay đổi.
  - Nếu tham số `inplace=True`, phương thức sẽ sửa đổi trực tiếp `Series` gốc và trả về `None`.

- **Cú pháp**:

```
Series.fillna( value=None, method=None, axis=None, inplace=False,
               limit=None, downcast=None, **kwargs )
```

Trong đó:

- ***value***:
  - *Kiểu dữ liệu*: không thể dùng list cho đối số này
    - `scalar` :giá trị đơn lẻ như số, chuỗi.
    - `Dictionary`: Điền các giá trị thiếu dựa trên chỉ mục (`index`) của `Series`. Key của `dictionary` là chỉ mục, và `value` là giá trị muốn điền. Cách này ít phổ biến hơn cho `Series` so với `DataFrame`.
    - `Series`: Điền các giá trị thiếu trong `Series` hiện tại bằng các giá trị tương ứng từ một `Series` khác (dựa trên sự khớp chỉ mục).
  - *Mô tả*: Giá trị được sử dụng để điền vào các ô bị thiếu (`NaN`).
    - Nếu là `scalar`: Tất cả các giá trị `NaN` sẽ được thay thế bằng giá trị này.
    - Nếu là `dict`: Dùng để chỉ định giá trị thay thế cho từng chỉ mục (index) cụ thể. Các chỉ mục không có trong dict sẽ không được điền. (Cách này ít phổ biến hơn cho `Series` so với `DataFrame`).

  - Nếu là `Series`: Các giá trị `NaN` trong `Series` hiện tại sẽ được điền bằng các giá trị tương ứng từ `Series` này, dựa trên sự khớp chỉ mục.
- *Giá trị mặc định*: `None`.

- ***method***:
  - *Kiểu dữ liệu*: {`'backfill'`, `'bfill'`, `'pad'`, `'ffill'`, `None`}.
  - *Mô tả*: Phương pháp sẽ được sử dụng để điền các giá trị thiếu.
    - `'pad'` / `'ffill'` `(forward-fill)`: Điền giá trị `NaN` bằng giá trị hợp lệ đứng ngay phía trước nó.
    - `'backfill'` / `'bfill'` `(backward-fill)`: Điền giá trị `NaN` bằng giá trị hợp lệ đứng ngay phía sau nó.
  - *Mặc định*: `None`.
  - *Lưu ý* (từ Pandas 2.1.0): Nên sử dụng 1 trong 2 phương thức `ffill` hoặc `bfill` trực tiếp thay vì thông qua tham số method này, mặc dù method vẫn được hỗ trợ. Do đó trong các ví dụ sau sẽ không dùng đối số method mà sử dụng trực tiếp các phương thức `ffill()` hoặc `bfill()`
- ***axis***:
  - *Kiểu dữ liệu*: {`0` hoặc `'index'`}.
  - *Mô tả*: Trục mà theo đó các giá trị thiếu sẽ được điền.
  - *Lưu ý*: Đối với `Series`, tham số này thường không được sử dụng và mặc định là `0` (hoặc `'index'`) vì `Series` chỉ có một trục. Nó có ý nghĩa hơn khi làm việc với `DataFrame`.
- ***inplace***:
  - *Kiểu dữ liệu*: `bool`.
  - *Mô tả*:
    - Nếu `True`: Thực hiện việc điền giá trị trực tiếp trên `Series` gốc (và phương thức sẽ trả về `None`).
    - Nếu `False`: (mặc định) Trả về một bản sao của `Series` đã được điền giá trị (`Series` gốc không thay đổi).
- ***limit***:
  - *Kiểu dữ liệu:* `int`.
  - *Mô tả*: Số lượng tối đa các giá trị `NaN` được điền.
    - Nếu method được chỉ định: Đây là số lượng tối đa các giá trị `NaN` liên tiếp được điền theo chiều `forward` hoặc `backward`. Nếu có một khoảng trống với nhiều hơn số `NaN` này, nó sẽ chỉ được điền một phần.
    - Nếu method không được chỉ định (tức là điền bằng `value`): Đây là số lượng tối đa các ô `NaN` sẽ được điền trên toàn bộ `Series`.
  - Phải lớn hơn `0` nếu không phải là `None`.
  - *Mặc định*: `None` (không có giới hạn).
- ***downcast***:
  - *Kiểu dữ liệu*: dict hoặc chuỗi `'infer'`.
  - *Mô tả*: Dùng để ép kiểu dữ liệu (downcast) sau khi điền giá trị, nếu có thể.

- Ví dụ: `{'item': 'dtype'}` để chỉ định kiểu dữ liệu cụ thể, hoặc `'infer'` để Pandas tự động cố gắng ép kiểu xuống kiểu nhỏ hơn phù hợp (ví dụ: từ `float64` xuống `int64` nếu tất cả giá trị đều là số nguyên sau khi điền).

- *Mặc định*: `None`.

- *****kwargs***:
  - Các đối số từ khóa (*keyword arguments*) khác không được dùng.

- **Một số ví dụ:**

(i).- ***value***:

- **Giá trị vô hướng** (*scalar value*): Điền tất cả các giá trị thiếu bằng một giá trị cụ thể.

Mã lệnh	*Kết quả*

`import pandas as pd`
`import numpy as np`
`S = pd.Series([1, 2, np.nan, 4, np.nan, 6])`
`print("Series gốc:\n",S)`
	`Fillna with zero:`
	`0 1.0`
	`1 2.0`
	`2 0.0`
	`3 4.0`
	`4 0.0`
	`5 6.0`
	`dtype: float64`

`S_filled_scalar = S.fillna(0)`
`print("Fillna with zero:\n",S_filled_scalar)`
	`Fillna with zero:`
	`0 1.0`
	`1 2.0`
	`2 0.0`
	`3 4.0`
	`4 0.0`
	`5 6.0`
	`dtype: float64`

- ***Series***: Điền giá trị trên `s1` dựa trên việc so khớp `index` của `s1` và `s2`. Nếu index khớp ở cả 2 bên và giá trị bên `s1` là `NaN`, thì giá trị này sẽ được thay thế bởi giá trị vừa so khớp (trên chỉ mục) của s2.

Mã lệnh	*Kết quả*

`import pandas as pd`
`import numpy as np`
`s1 = pd.Series([np.nan, 2, np.nan, 4, np.nan],`
`index=['a', 'b', 'c', 'd','e'])`
`print("s1 gốc=\n",s1)`
	`s1 gốc=`
	`a NaN`
	`b 2.0`
	`c NaN`
	`d 4.0`
	`e NaN`
	`dtype: float64`

`s2 = pd.Series([100, 300], index=['a', 'c'])`
`print("s2 gốc=\n",s2)`
	`s2 gốc=`
	`a 100`
	`c 300`
	`dtype: int64`

`s_filled_series = s1.fillna(s2)`
`print("Fillna trên s1 với giá trị lấy từ s2:\n")`
`print(s_filled_series)`
	`Fillna trên s1 với giá trị lấy từ s2:`
	`a 100.0`
	`b 2.0`
	`c 300.0`
	`d 4.0`
	`e NaN`
	`dtype: float64`

(ii).- ***limit***: Một số nguyên, chỉ định số lượng tối đa các giá trị `NaN` liên tiếp (hoặc tổng số, tùy thuộc vào việc có sử dụng `method` hay không) được điền.

Mã lệnh	*Kết quả*

`import pandas as pd`
`import numpy as np`
`s = pd.Series([np.nan, np.nan, np.nan, 4, 5])`
`print("s gốc=\n",s1)`
	`s gốc=`
	`0 NaN`
	`1 NaN`
	`2 NaN`
	`3 4.0`
	`4 5.0`
	`dtype: float64`

`import pandas as pd`
`import numpy as np`
`# Điền tối đa 2 giá trị NaN bằng 0`
`s_limited_value = s.fillna(0, limit=2)`
`print('Chỉ điền zero cho 2 giá trị NaN đầu tiên\n', s_limited_value)`
	`Chỉ điền zero cho 2 giá trị NaN đầu tiên`
	
	`0 0.0`
	`1 0.0`
	`2 NaN`
	`3 4.0`
	`4 5.0`
	`dtype: float64`

- **Các trường hợp sử dụng phổ biến sử dụng *`fillna`*:**
  - **Thay thế giá trị thiếu bằng một hằng số**: Ví dụ, thay thế tất cả các giá trị thiếu bằng `0`, `"Không có thông tin"`, hoặc giá trị `mean/median` của cột.
  - **Điền giá trị thiếu một cách có điều kiện**: Kết hợp `fillna()` với các phương thức khác của Pandas để điền giá trị dựa trên các điều kiện phức tạp hơn.

3.7.9. head(n)

- Trả về n phần tử đầu tiên của `Series`.
- `seriesName.head(5)`

3.7.10. isnull() hoặc isna

- pandas.Series.isnull()được sử dụng để phát hiện các giá trị bị thiếu (*missing values*) trong một đối tượng Series.
- Giá trị trả về: seriesName.isnull() trả về một đối tượng Series mới có cùng kích thước (số lượng phần tử) với Series ban đầu. Tuy nhiên, thay vì chứa các giá trị gốc, Series mới này sẽ chứa các giá trị boolean (True hoặc False):
  - **True**: Nếu phần tử tương ứng trong Series gốc là một giá trị bị thiếu (ví dụ: None hoặc numpy.NaN).
  - **False**: Nếu phần tử tương ứng trong Series gốc không phải là giá trị bị thiếu.
- **Lưu ý:**
  - pandas.Series.isnull() là một tên gọi khác (alias) của pandas.Series.isna(). Cả hai đều thực hiện cùng một chức năng.
  - Theo mặc định, các chuỗi rỗng ('') hoặc giá trị vô cực (numpy.inf) không được coi là giá trị bị thiếu, trừ khi thay đổi cài đặt mặc định của Pandas (`pandas.options.mode.use_inf_as_na = True`).
- **Cách sử dụng phổ biến:** seriesName.isnull() rất hữu ích trong các tác vụ làm sạch dữ liệu (data cleaning), xử lý dữ liệu (data wrangling) và phân tích dữ liệu.
- Ví dụ:

Mã lệnh	*Kết quả*

• *Kiểm tra xem có giá trị nào bị thiếu không:*
`import pandas as pd`
	`0 False`
	`1 False`
	`2 True`

```
import numpy as np
data = pd.Series([1, 2, np.nan, 4, None, 6])
print(data.isnull())
```

```
3    False
4     True
5    False
dtype: bool
```

• *Đếm số lượng giá trị bị thiếu: có thể kết hợp isnull() với phương thức .sum() để đếm tổng số giá trị True (tức là tổng số giá trị bị thiếu).*

```
import pandas as pd
import numpy as np
data = pd.Series([1, 2, np.nan, 4, None, 6])
so_luong_thieu = data.isnull().sum()
print(f"Số lượng giá trị bị thiếu: {so_luong_thieu}")
```

```
Số lượng giá trị bị thiếu: 2
```

• *Lọc các hàng có giá trị bị thiếu hoặc không bị thiếu: có thể sử dụng Series boolean trả về từ isnull() để lọc dữ liệu.*

```
import pandas as pd
import numpy as np
data = pd.Series([1, 2, np.nan, 4, None, 6],
                 index=['a', 'b', 'c', 'd', 'e', 'f'])
```

```
a    1.0
b    2.0
c    NaN
d    4.0
e    NaN
f    6.0
dtype: float64
```

```
# Lấy các giá trị không bị thiếu
gia_tri_khong_thieu = data[~data.isnull()]
# Hoặc data[data.notnull()]
print("Các giá trị không bị thiếu:\n",
gia_tri_khong_thieu)
```

```
Các giá trị không bị thiếu:
a    1.0
b    2.0
d    4.0
f    6.0
dtype: float64
```

```
# Lấy các giá trị bị thiếu
gia_tri_bi_thieu = data[data.isnull()]
print("\nCác giá trị bị thiếu:\n", gia_tri_bi_thieu)
```

```
Các giá trị bị thiếu:
c   NaN
e   NaN
dtype: float64
```

3.7.11. items()

- Được sử dụng để trả về một `iterator` chứa các cặp giá trị `index` và `value` từ `Series` (tương tự như `key` và `values` của `dictionary`). Với `items()`, có thể duyệt qua từng phần tử của `Series` một cách hiệu quả.
- Ví dụ

Mã lệnh	*Kết quả*

• *Cách sử dụng cơ bản của Series.items():*
`import pandas as pd`
`# Tạo Series`
`s = pd.Series([10, 20, 30], index=['a', 'b', 'c'])`
`# Duyệt qua các phần tử bằng items()`
`for index, value in s.items():`
`  print(f"Index: {index}, Value: {value}")`
	`# Index: a, Value: 10`
	`# Index: b, Value: 20`
	`# Index: c, Value: 30`

• *Sử dụng items() để tạo danh sách các cặp giá trị:*
`pairs = list(s.items())`
`print(pairs)`
	`[('a', 10), ('b', 20), ('c', 30)]`

• *Ứng dụng thực tế: Tạo dictionary từ Series bằng items():*
`dictionary = {index: value for index, value in s.items()}`
`print(dictionary)`
	`{'a': 10, 'b': 20, 'c': 30}`

3.7.12. max()

- Trả về giá trị lớn nhất trong `Series`.
- `seriesName.max()`

3.7.13. map()

- Dùng để ánh xạ (`mapping`) từng phần tử trong `Series` (tương tự như vòng lặp (`loop`) nhưng hiệu quả hơn) thông qua một hàm (`function`), `dictionary`, hoặc một chuỗi ánh xạ (`Series mapping`). Phương thức này thường được sử dụng để áp dụng các phép biến đổi hoặc thay đổi dữ liệu của `Series`.
- Cú pháp

`Series.map(arg, na_action=None)`

  - ***`arg`***: Hàm, dictionary, hoặc `Series` được sử dụng để ánh xạ giá trị.
  - ***`na_action`*** (Tùy chọn): Quyết định cách xử lý giá trị `NaN`: nếu `na_action='ignore'`, bỏ qua giá trị `NaN`.

- Ví dụ

Mã lệnh	*Kết quả*

• *Sử dụng với hàm (function):*
`import pandas as pd`
`# Tạo Series`
`s = pd.Series([1, 2, 3, 4, 5])`
`# Áp dụng hàm lambda để nhân đôi mỗi giá trị`
`result = s.map(lambda x: x * 2)`
`print(result)`
	`0  2`
	`1  4`
	`2  6`
	`3  8`
	`4  10`
	`dtype: int64`

• *Sử dụng với dictionary:*
`# Tạo dictionary ánh xạ`
`mappingD = {1: 'one', 2: 'two', 3: 'three'}`
`# Áp dụng ánh xạ qua dictionary`
`result = s.map(mappingD)`
`print(result)`
	`0  one`
	`1  two`
	`2  three`
	`3  NaN`
	`4  NaN`
	`dtype: object`

• *Sử dụng với Series khác:*
`# Tạo Series khác để ánh xạ`
`mappingS = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])`
`# Ánh xạ qua Series`
`result = s.map(mappingS)`
`print(result)`
	`0  A`
	`1  B`
	`2  C`
	`3  NaN`
	`4  NaN`
	`dtype: object`

• *Xử lý giá trị NaN:*
`s_with_nan = pd.Series([1, 2, None, 4])`
`# Áp dụng map và bỏ qua giá trị NaN`
`result = s_with_nan.map(lambda x: x * 2, na_action='ignore')`
`print(result)`
	`0  2.0`
	`1  4.0`
	`2  NaN`
	`3  8.0`
	`dtype: float64`

3.7.14. mean()

- Tính giá trị trung bình của `Series`.
- **`seriesName.mean()`**

3.7.15. min()

- Trả về giá trị nhỏ nhất trong `Series`.
- **`seriesName.min()`**

3.7.16. notnull() hay notna()

- `pandas.Series.notnull()` được sử dụng để phát hiện các giá trị không bị thiếu (*non-missing/existing values*) trong một đối tượng `Series`.
- Chức năng chính: Phương thức này sẽ duyệt qua từng phần tử trong `Series` và kiểm tra xem phần tử đó có phải là giá trị hợp lệ (tức là không bị thiếu) hay không.
- Giá trị trả về: `seriesName.notnull()` trả về một đối tượng `Series` mới có cùng kích thước (số lượng phần tử) với `Series` ban đầu. `Series` mới này sẽ chứa các giá trị `boolean` (`True` hoặc `False`):
  - `True`: Nếu phần tử tương ứng trong Series gốc là một giá trị hợp lệ (không phải None hoặc `numpy.NaN`).
  - `False`: Nếu phần tử tương ứng trong Series gốc là một giá trị bị thiếu (ví dụ: `None` hoặc `numpy.NaN`).
- Mối quan hệ với `isnull()`: `pandas.Series.notnull()` thực chất là phương thức đối nghịch với `pandas.Series.isnull()`. Nếu `isnull()` trả về `True` cho một vị trí, `notnull()` sẽ trả về `False` cho vị trí đó và ngược lại.
- Lưu ý:
  - `pandas.Series.notnull()` là một tên gọi khác (`alias`) của `pandas.Series.notna()`. Cả hai đều thực hiện cùng một chức năng.
  - Tương tự như `isnull()`, theo mặc định, các chuỗi rỗng (`''`) hoặc giá trị vô cực (`numpy.inf`) không được coi là giá trị bị thiếu (và do đó `notnull()` sẽ trả về `True` cho chúng), trừ khi trước đó có thực hiện thay đổi cài đặt mặc định của `Pandas` (`pandas.options.mode.use_inf_as_na = True`).
- Cách sử dụng phổ biến: `seriesName.notnull()` rất hữu ích trong các tác vụ làm sạch dữ liệu, xử lý dữ liệu và phân tích dữ liệu, đặc biệt khi muốn tập trung vào các giá trị hợp lệ.
- Một số ví dụ

Mã lệnh	*Kết quả*

• *Kiểm tra xem giá trị nào không bị thiếu:*
`import pandas as pd`
`import numpy as np`
`data = pd.Series([1, 2, np.nan, 4, None, 6])`
	`Series gốc:`
	`0 1.0`
	`1 2.0`
	`2 NaN`
	`3 4.0`
	`4 NaN`
	`5 6.0`

`print(data.notnull())`	`0 True`
	`1 True`
	`2 False`
	`3 True`
	`4 False`
	`5 True`
	`dtype: bool`

• *Đếm số lượng giá trị không bị thiếu: có thể kết hợp* `notnull()` *với phương thức* `.sum()` *để đếm tổng số giá trị* `True` *(tức là tổng số giá trị không bị thiếu).*
`import pandas as pd`
`import numpy as np`
`S = pd.Series([1, 2, np.nan, 4, None, 6])`
`so_luong_khong_thieu = S.notnull().sum()`
	`Số lượng giá trị không bị thiếu: 4`

```
print(f"Số lượng giá trị không bị thiếu:
                                    {so_luong_khong_thieu}")
```

- *Lọc các hàng có giá trị không bị thiếu: có thể sử dụng* `Series boolean` *trả về từ* `notnull()` *để chỉ giữ lại các phần tử hợp lệ trong* `Series`.

```
import pandas as pd
import numpy as np
S = pd.Series([1, 2, np.nan, 4, None, 6], index=['a', 'b',
                                                 'c', 'd', 'e', 'f'])
# Lấy các giá trị không bị thiếu
gia_tri_khong_thieu = S[data.notnull()]
print("Các giá trị không bị thiếu:\n", gia_tri_khong_thieu)
```

```
Các giá trị không bị thiếu:
a    1.0
b    2.0
d    4.0
f    6.0
dtype: float64
```

3.7.17. reset_index()

- Được sử dụng để đặt lại chỉ số (`index`) của một `Series` về giá trị mặc định (số nguyên liên tiếp, bắt đầu từ `0`). Chỉ số ban đầu sẽ được chuyển thành một cột trong `DataFrame` kết quả. Hàm này rất hữu ích khi cần làm việc với dữ liệu mà chỉ số không còn cần thiết hoặc khi cần chuyển `Series` thành `DataFrame`.
- Cú pháp

```
Series.reset_index(level=None, drop=False, name=None, inplace=False)
```

Trong đó

- ***level***: (Tùy chọn) Xác định mức chỉ số (cho `Series` nhiều cấp - `MultiIndex`) cần đặt lại. Mặc định là `None`, nghĩa là đặt lại tất cả các mức.
- ***drop***: Nếu `True`, bỏ qua chỉ số ban đầu thay vì chuyển nó thành một cột.
- ***name***: Tên của cột giá trị `Series` trong `DataFrame` mới. Mặc định là không tên.
- ***inplace***: Nếu `True`, áp dụng thay đổi trực tiếp trên `Series` ban đầu (không trả về `DataFrame` mới). Mặc định là `False`.

- Ví dụ

Mã lệnh	*Kết quả*

`import pandas as pd`
`# Tạo một Series với index tùy chỉnh`
`s = pd.Series([10, 20, 30], index=`
`['a', 'b', 'c'])`
`print('Series gốc:')`
`print(s)`
	`Series gốc:`
	`a  10`
	`b  20`
	`c  30`
	`dtype: int64`

`# Đặt lại chỉ số mặc định`
`result = s.reset_index() # drop=False`
`print('Series sau khi đặt lại chỉ số:')`
`print(result)`
	`Series sau khi đặt lại chỉ số:`
	`  index  0`
	`0  a  10`
	`1  b  20`
	`2  c  30`

`print('Bỏ qua (không lưu) chỉ số`
`ban đầu:')`
`result = s.reset_index(drop=True)`
`print(result)`
	`Bỏ qua (không lưu) chỉ số ban đầu:`
	`0  10`
	`1  20`
	`2  30`
	`dtype: int64`

`print('Chỉ định tên cột giá trị:')`
`result = s.reset_index(name='values')`
`print(result)`
	`Chỉ định tên cột giá trị:`
	`  index  values`
	`0  a  10`
	`1  b  20`
	`2  c  30`

`print('Đặt lại một mức chỉ số với`
`level=0:')`
`result = s_multi.reset_index(level=0)`
`print(result)`
	`Sử dụng với Series có MultiIndex:`
	`Đặt lại một mức chỉ số với level=0:`
	`  level_0  0`
	`1  A  100`
	`2  A  200`

```
1       B  300
```

```
print('Đặt lại một mức chỉ số với
                                level=1:')
result = s_multi.reset_index(level=1)
print(result)
```

```
Đặt lại một mức chỉ số với
level=1:
   level_1    0
A        1  100
A        2  200
B        1  300
```

3.7.18. sort_index()

- Được sử dụng để sắp xếp một đối tượng `Series` dựa trên các nhãn của chỉ mục (`index labels`) theo thứ tự (tăng dần hoặc giảm dần). Điều này khác với `Series.sort_values()`, vốn sắp xếp `Series` dựa trên các giá trị dữ liệu.
- Cú pháp cơ bản:

```
pandas.Series.sort_index(axis=0, level=None, ascending=True,
    inplace=False, kind='quicksort', na_position='last',
                    sort_remaining=True)
```

trong đó:

- ***axis*** (mặc định là 0): Đối với `Series`, tham số này không có nhiều ý nghĩa vì `Series` chỉ có một trục (trục 0). Nó được giữ lại để tương thích với API của `DataFrame`.
- ***level***:
  - Được sử dụng khi `Series` có `MultiIndex` (chỉ mục đa cấp).
  - Nếu là một số nguyên hoặc tên của một cấp chỉ mục, nó chỉ định cấp chỉ mục nào sẽ được sử dụng để sắp xếp.
  - Nếu là một danh sách các số nguyên hoặc tên, nó chỉ định nhiều cấp để sắp xếp theo (sắp xếp theo cấp đầu tiên, sau đó theo cấp thứ hai cho các giá trị bằng nhau ở cấp đầu tiên, v.v.).
  - Nếu `=None` (mặc định), `Series` sẽ được sắp xếp theo tất cả các cấp của chỉ mục.
- ***ascending***:
  - Nếu `True` (mặc định): Sắp xếp các nhãn chỉ mục theo thứ tự tăng dần (ví dụ: A-Z, 0-9).
  - Nếu `False`: Sắp xếp các nhãn chỉ mục theo thứ tự giảm dần (ví dụ: Z-A, 9-0).
  - Cũng có thể truyền một danh sách các giá trị boolean nếu đang sắp xếp theo nhiều cấp (`level`) để chỉ định thứ tự tăng/giảm cho từng cấp.
- ***inplace*** (mặc định là False):
  - Nếu `False`: Hàm sẽ trả về một `Series` mới đã được sắp xếp, `Series` gốc không thay đổi.
  - Nếu `True`: Hàm sẽ sắp xếp `Series` gốc trực tiếp (tại chỗ) và trả về `None`. Việc sử dụng `inplace=True` có thể tiết kiệm bộ nhớ cho các đối tượng lớn nhưng cần cẩn thận vì nó làm thay đổi đối tượng gốc.
- ***kind***:
  - Chỉ định thuật toán sắp xếp sẽ được sử dụng. Các lựa chọn bao gồm:
    - `'quicksort'` (mặc định): thường nhanh nhất

  - `'mergesort'` : ổn định (stable), nghĩa là nếu hai phần tử có khóa bằng nhau, thứ tự tương đối của chúng sẽ được giữ nguyên)
  - `'heapsort'`
  - `'stable'`: tương đương với `'mergesort'`
- ***`na_position`*** (mặc định là `'last'`):
  - Chỉ định vị trí của các nhãn chỉ mục `NaN` (nếu có trong chỉ mục) sau khi sắp xếp:
    - `'last'`: Đặt các giá trị `NaN` ở cuối.
    - `'first'`: Đặt các giá trị `NaN` ở đầu.
  - Tham số này không áp dụng cho việc sắp xếp `MultiIndex` theo nhiều cấp.
- ***`sort_remaining`*** (mặc định là `True`):
  - Chỉ có tác dụng khi sắp xếp theo một hoặc nhiều cấp cụ thể của `MultiIndex` (sử dụng tham số `level`).
  - Nếu `True`: Sau khi sắp xếp theo các cấp được chỉ định trong `level`, các cấp còn lại (nếu có) cũng sẽ được sắp xếp theo thứ tự tăng dần.
  - Nếu `False`: Các cấp còn lại sẽ không được sắp xếp thêm.
- **Kết quả trả về:**
  - Nếu `inplace=False` (mặc định): Trả về một `Series` mới đã được sắp xếp theo chỉ mục.
  - Nếu `inplace=True`: Trả về `None`, và `Series` gốc sẽ được sửa đổi trực tiếp.

- Công dụng của `sort_index()`:
  - *Tổ chức dữ liệu*: giúp đảm bảo rằng dữ liệu được sắp xếp theo một thứ tự logic dựa trên chỉ mục, giúp việc đọc và phân tích dễ dàng hơn.
  - *Chuẩn bị cho các thao tác khác*: Một số thao tác trong Pandas (ví dụ: `reindex`, một số loại `merge` hoặc `join`, truy cập dựa trên lát cắt (`slicing`) của chỉ mục đã sắp xếp) hoạt động hiệu quả hơn hoặc cho kết quả như mong đợi khi chỉ mục đã được sắp xếp.
  - *Truy cập hiệu quả*: Với chỉ mục đã được sắp xếp, việc tìm kiếm và truy cập dữ liệu dựa trên nhãn chỉ mục có thể nhanh hơn.
  - *Hiển thị và báo cáo*: Khi hiển thị dữ liệu, việc sắp xếp theo chỉ mục thường làm cho bảng dữ liệu dễ theo dõi hơn.

- Ví dụ minh họa:

Mã lệnh	*Kết quả*

`import pandas as pd`
`import numpy as np`
`''' Ví dụ 1: Sắp xếp Series với chỉ mục đơn giản'''`
`s1 = pd.Series([10, 20, 5, 30], index=['d', 'a', 'c', 'b'])`
`print("Series gốc s1:")`
`print(s1)`
	`Series gốc s1:`
	`d  10`
	`a  20`
	`c  5`
	`b  30`
	`dtype: int64`

`print("\nSắp xếp s1 tăng dần theo chỉ mục:")`
`# Mặc định sort_ ascending=True`
`s1_sorted_asc = s1.sort_index()`
`print(s1_sorted_asc)`
	`Sắp xếp s1 tăng dần theo chỉ mục:`
	`a  20`
	`b  30`
	`c  5`
	`d  10`

```
                                                dtype: int64
print("\nSắp xếp s1 giảm dần theo chỉ mục:")
s1_sorted_desc = s1.sort_index(ascending=False)
print(s1_sorted_desc)
```
```
Sắp xếp s1 giảm dần theo chỉ mục:
d    10
c     5
b    30
a    20
dtype: int64
```

```
''' Ví dụ 2: Sắp xếp tại chỗ (inplace=True) '''
s2 = pd.Series({'Z': 1, 'Y': 2, 'X': 3})
print("\nSeries gốc s2:")
print(s2)
```
```
Series gốc s2:
Z    1
Y    2
X    3
dtype: int64
```

```
# Sắp xếp s2 trực tiếp
print("\nSeries s2 sau khi sắp xếp tại chỗ:")
s2.sort_index(inplace=True)
print(s2)
```
```
Series s2 sau khi sắp xếp tại chỗ:
X    3
Y    2
Z    1
dtype: int64
```

```
''' Ví dụ 3: Sắp xếp Series với MultiIndex'''
index_tuples = [('group1', 'a'), ('group2', 'c'),
                ('group1', 'b'), ('group2', 'a')]
multi_idx =
         pd.MultiIndex.from_tuples(index_tuples,
                    names=['level_1', 'level_2'])
s_multi = pd.Series([100, 200, 300, 400],
                                  index=multi_idx)
print("\nSeries với MultiIndex (s_multi):")
print(s_multi)
```
```
Series với MultiIndex (s_multi):
level_1  level_2
group1   a          100
group2   c          200
group1   b          300
group2   a          400
dtype: int64
```

```
# Sắp xếp theo tất cả các cấp (mặc định)
print("\nSắp xếp s_multi theo tất cả các cấp:")
s_multi_sorted_all = s_multi.sort_index()
print(s_multi_sorted_all)
```
```
Sắp xếp s_multi theo tất cả các cấp:
level_1  level_2
group1   a          100
         b          300
group2   a          400
         c          200
dtype: int64
```

```
# Sắp xếp chỉ theo level_2
print("\nSắp xếp s_multi chỉ theo level_2
                (level_1 không đảm bảo thứ tự):")
s_multi_sorted_level2 =
             s_multi.sort_index(level='level_2')
print(s_multi_sorted_level2)
```
```
Sắp xếp s_multi chỉ theo level_2 (level_1 không đảm bảo thứ tự):
level_1  level_2
group1   a          100
group2   a          400
group1   b          300
group2   c          200
dtype: int64
```

```
''' Sắp xếp theo level_1 (giảm dần), sau đó
level_2 (tăng dần) '''
print("\nSắp xếp s_multi: level_1 giảm dần,
                              level_2 tăng dần:")
s_multi_sorted_specific =
            s_multi.sort_index(level=['level_1',
            'level_2'], ascending=[False, True])
print(s_multi_sorted_specific)
```
```
Sắp xếp s_multi:
level_1 giảm dần,
level_2 tăng dần:
level_1  level_2
group2   a          400
         c          200
group1   a          100
         b          300
dtype: int64
```

```
''' Ví dụ 4: Xử lý NaN trong chỉ mục'''
s_with_nan_index = pd.Series([1, 2, 3, 4],
                   index=['x', np.nan, 'y', 'a'])
print("\nSeries với NaN trong chỉ mục:")
print(s_with_nan_index)
```
```
Series với NaN trong chỉ mục:
x      1
NaN    2
y      3
a      4
dtype: int64
```

```
print("\nSắp xếp với NaN ở cuối:")
# Mặc định na_position='last'
s_nan_last = s_with_nan_index.sort_index()
print(s_nan_last)
```
```
Sắp xếp với NaN ở cuối:
a      4
x      1
y      3
NaN    2
dtype: int64
```

```
s_nan_first =
```
```
Sắp xếp với NaN ở đầu:
```

```
s_with_nan_index.sort_index(na_position='first')
print("\nSắp xếp với NaN ở đầu:")
print(s_nan_first)
```

```
NaN    2
a      4
x      1
y      3
dtype: int64
```

3.7.19. std()

- Tính độ lệch chuẩn các giá trị của `Series`.
- `seriesName.std()`

3.7.20. str()

3.7.20.1. Giới thiệu

`series.str` không phải là một hàm (*function*) được gọi trực tiếp như `series.str()`. Thay vào đó, `series.str` là một thuộc tính truy cập (*accessor*) cho phép áp dụng các phương thức xử lý chuỗi (string methods) lên từng phần tử trong một đối tượng `Series`.

Điều kiện tiên quyết là `Series` đó phải chứa các đối tượng kiểu chuỗi (`string`) hoặc các đối tượng có thể được coi là chuỗi (ví dụ: `None` hoặc `NaN` sẽ được xử lý tương ứng bởi các phương thức chuỗi).

- Công dụng của series.str
  - *Xử lý NaN tự động*: Hầu hết các phương thức str sẽ tự động xử lý các giá trị `NaN` (thường trả về `NaN` hoặc `None` tùy theo phương thức) mà không gây lỗi.
  - *Hiệu suất*: Các thao tác được vector hóa thường nhanh hơn so với việc lặp qua từng phần tử.
  - *Dễ đọc*: Cú pháp rõ ràng và dễ hiểu hơn so với việc viết các vòng lặp phức tạp.

3.7.20.2. Chức năng chính:

`series.str` cung cấp một cách vector hóa (*vectorized way*) để thực hiện các thao tác trên chuỗi. Điều này có nghĩa là có thể áp dụng một phương thức chuỗi cho toàn bộ `Series` mà không cần phải viết vòng lặp `for` thủ công, giúp mã nguồn ngắn gọn và hiệu quả hơn.

3.7.20.3. Các phương thức "con"

- `series.str.lower()`: Chuyển tất cả chuỗi trong Series thành chữ thường.
- `series.str.upper()`: Chuyển tất cả chuỗi trong Series thành chữ hoa.
- `series.str.len()`: Trả về độ dài của mỗi chuỗi trong Series.
- `series.str.contains('substring')`: Kiểm tra xem mỗi chuỗi có chứa một chuỗi con (substring) cụ thể hay không, trả về một Series boolean.
- `series.str.startswith('prefix')`: Kiểm tra xem mỗi chuỗi có bắt đầu bằng một tiền tố (`prefix`) cụ thể hay không.
- `series.str.endswith('suffix')`: Kiểm tra xem mỗi chuỗi có kết thúc bằng một hậu tố (`suffix`) cụ thể hay không.
- `series.str.replace('old', 'new')`: Thay thế một chuỗi con cũ bằng một chuỗi con mới trong mỗi chuỗi.

- `series.str.split('delimiter')`: Tách mỗi chuỗi thành một danh sách các chuỗi con dựa trên một ký tự phân cách (delimiter).
- `series.str.strip()`: Loại bỏ khoảng trắng ở đầu và cuối mỗi chuỗi.
- `series.str.get(i)`: Trích xuất phần tử ở vị trí `i` nếu mỗi chuỗi là một danh sách hoặc `tuple` (thường sau khi dùng `split()`).
- `series.str.extract(r'regex_pattern')`: Trích xuất các nhóm khớp từ một biểu thức chính quy (regular expression).

3.7.20.4. Ví dụ

Mã lệnh	*Kết quả*

`import pandas as pd`
`import numpy as np`
`# Tạo một Series chứa dữ liệu chuỗi`
`data = pd.Series(['  Apple Pie  ',`
`'banana bread', 'CHERRY CAKE',`
`np.nan, 'Orange Juice'])`
`print("Series gốc:")`
`print(data)`
	`Series gốc:`
	`0  Apple Pie`
	`1  banana bread`
	`2  CHERRY CAKE`
	`3  NaN`
	`4  Orange Juice`
	`dtype: object`

`# Chuyển thành chữ thường`
`print("\nChuyển tất cả sang chữ thường:")`
`print(data.str.lower())`
	`Chuyển tất cả sang chữ thường:`
	`0  apple pie`
	`1  banana bread`
	`2  cherry cake`
	`3  NaN`
	`4  orange juice`
	`dtype: object`

`# Kiểm tra xem có chứa 'bread' không`
`print("\nPhần tử có chứa từ 'bread':")`
`print(data.str.contains('bread'))`
	`Phần tử có chứa từ 'bread':`
	`0  False`
	`1  True`
	`2  False`
	`3  NaN`
	`4  False`
	`dtype: object`

`# Lấy độ dài của mỗi chuỗi (NaN sẽ là NaN)`
`print("\nĐộ dài mỗi chuỗi:")`
`print(data.str.len())`
	`Độ dài mỗi chuỗi:`
	`0  15.0`
	`1  12.0`
	`2  11.0`
	`3  NaN`
	`4  12.0`
	`dtype: float64`

`# Loại bỏ khoảng trắng thừa`
`print("\nLoại bỏ khoảng trắng đầu/cuối chuỗi:")`
`print(data.str.strip())`
	`Loại bỏ khoảng trắng đầu/cuối chuỗi:`
	`0  Apple Pie`
	`1  banana bread`
	`2  CHERRY CAKE`
	`3  NaN`
	`4  Orange Juice`
	`dtype: object`

`# Tách chuỗi dựa trên khoảng trắng`
`print("\nTách chuỗi:")`
`print(data.str.split(' '))`
	`Tách chuỗi:`
	`0  [, , , Apple, Pie, , , ]`
	`1  [banana, bread]`
	`2  [CHERRY, CAKE]`
	`3  NaN`
	`4  [Orange, Juice]`
	`dtype: object`

`''' Thay thế 'Juice' bằng 'Smoothie'`
`fillna('') được dùng để xử lý giá trị NaN trước khi áp dụng replace,`
`vì .str.replace trên NaN có thể không hoạt động như mong đợi trong mọi trường hợp. '''`
`print("\nThay thế chuỗi 'Juice' bằng 'Smoothie':")`
`print(data.fillna('').str.replace('Juice', 'Smoothie'))`
	`Thay thế chuỗi 'Juice' bằng 'Smoothie':`
	`0  Apple Pie`
	`1  banana bread`
	`2  CHERRY CAKE`
	`3`
	`4  Orange Smoothie`
	`dtype: object`

3.7.21. sum()

- Tính tổng các giá trị của `Series`.
- `seriesName.sum()`

3.7.22. tail(n)

- Trả về n phần tử cuối cùng của `Series`.
- `seriesName.tail(5)`

3.7.23. to_numpy()

- Được sử dụng để chuyển đổi một `pandas Series` thành một mảng `NumPy` (`NumPy array`). Điều này hữu ích khi cần thực hiện các phép toán số học hoặc xử lý dữ liệu bằng các công cụ `NumPy`.
- Lợi ích:
  - Thực hiện các phép toán nhanh chóng và hiệu quả với `NumPy`.
  - Tích hợp tốt hơn với các thư viện khác, chẳng hạn như `scikit-learn` hoặc `TensorFlow`.
- Cú pháp:

```
Series.to_numpy(dtype=None, copy=False)
```

  - ***dtype***: (Tùy chọn) Chỉ định kiểu dữ liệu của mảng kết quả. Nếu không cung cấp, nó sẽ sử dụng kiểu dữ liệu mặc định của `Series`.
  - ***copy***: (Tùy chọn) Nếu `True`, một bản sao của dữ liệu sẽ được trả về ngay cả khi không cần thiết. Nếu `False`, nó sẽ chỉ sao chép khi cần.
- Ví dụ

Mã lệnh	*Kết quả*

• *Chuyển* `Series` *sang* `NumPy array`*:*
`import pandas as pd`
`# Tạo một Series`
`s = pd.Series([10, 20, 30, 40, 50])`
`# Chuyển sang NumPy array`
`arr = s.to_numpy()`
`print(arr)`
	`[10 20 30 40 50]`

`print(type(arr))`	`<class 'numpy.ndarray'>`

• *Chuyển* `Series` *có kiểu dữ liệu cụ thể:*
`arr_float = s.to_numpy(dtype=float)`
`print(arr_float)`
	`[10.0 20.0 30.0 40.0 50.0]`

• *Ví dụ với dữ liệu chuỗi (*`string`*):*
`s_string = pd.Series(["apple", "banana", "cherry"])`
`arr_string = s_string.to_numpy()`
`print(arr_string)`
	`['apple' 'banana' 'cherry']`

• *Ví dụ với giá trị thiếu (*`NaN`*): Nếu* `Series` *chứa giá trị thiếu, mảng* `NumPy` *sẽ có kiểu* `object` *hoặc* `float` *(tùy thuộc vào dữ liệu):*
`s_missing = pd.Series([1, 2, None, 4])`
`arr_missing = s_missing.to_numpy()`
`print(arr_missing)`
	`[1.0 2.0 nan 4.0]`

- Lưu ý: Mặc dù `to_numpy()` rất tiện lợi, nhưng chỉ số (`index`) của `Series` không được giữ lại trong mảng `NumPy`. Nếu cần cả giá trị và chỉ số, nên sử dụng phương thức `.items()` hoặc `.reset_index()` trước khi chuyển đổi.

3.7.24. tolist()

- ***Giới thiệu***
  - Phương thức `.tolist()` thường được sử dụng với các đối tượng `pandas.Series` (đại diện cho một cột của `DataFrame`) để chuyển đổi một cột dữ liệu thành một list hoặc với mảng `NumPy`.
  - Để chuyển đổi toàn bộ `DataFrame` thành một cấu trúc danh sách các hàng (*list of list*), cần sử dụng `df.values.tolist()` hoặc tốt hơn là `df.to_numpy().tolist()`.
- ***Series.tolist()***
  - Khi có một đối tượng `pandas.Series` (ví dụ, một cột từ `DataFrame`), phương thức `.tolist()` sẽ chuyển đổi tất cả các giá trị trong `Series` đó thành một danh sách (`list`) Python tiêu chuẩn.
  - Ví dụ:

Mã lệnh	*Kết quả*

`import pandas as pd`
`# Tạo một DataFrame ví dụ`
`data = {'Tên': ['An', 'Bình', 'Cường', 'Dũng'],`
`'Tuổi': [22, 25, 21, 28],`
`'Điểm': [8.5, 9.0, 7.5, 8.8]}`
`df = pd.DataFrame(data)`
`print("DataFrame gốc:")`
`print(df)`
	`DataFrame gốc:`
	`Tên Tuổi Điểm`
	`0 An 22 8.5`
	`1 Bình 25 9.0`
	`2 Cường 21 7.5`
	`3 Dũng 28 8.8`

`# Lấy một cột (Series)`
`cot_ten = df['Tên']`
`print("Cột 'Tên' (là một pandas.Series):")`
`print(cot_ten)`
	`Cột 'Tên' (là một pandas.Series):`
	`0 An`
	`1 Bình`
	`2 Cường`
	`3 Dũng`
	`Name: Tên, dtype: object`

`print("Kiểu của cột 'Tên':", type(cot_ten))`	`Kiểu của cột 'Tên': <class 'pandas.core.series.Series'>`

`# Sử dụng .tolist() trên Series 'Tên'`
`list_ten = cot_ten.tolist()`
`print("Cột 'Tên' sau khi dùng .tolist():")`
`print(list_ten)`
	`Cột 'Tên' sau khi dùng .tolist():`
	`['An', 'Bình', 'Cường', 'Dũng']`

`print("Kiểu của list_ten:", type(list_ten))`	`Kiểu của list_ten: <class 'list'>`

`# Tương tự với cột 'Tuổi'`
`list_tuoi = df['Tuổi'].tolist()`
`print("Cột 'Tuổi' sau khi dùng .tolist():")`
`print(list_tuoi)`
	`Cột 'Tuổi' sau khi dùng .tolist():`
	`[22, 25, 21, 28]`

- **Chuyển đổi toàn bộ `DataFrame` thành danh sách các danh sách** (sử dụng `.values.tolist()` hoặc `.to_numpy().tolist()`)
  - Vì `DataFrame` không có phương thức `.tolist()` trực tiếp, nên nếu muốn chuyển đổi toàn bộ dữ liệu trong `DataFrame` thành một `list` với mỗi phần tử là một danh sách con (`sublist`) đại diện cho một hàng, khi đó cần thực hiện qua hai bước:
    - **Bước 1:** Lấy biểu diễn mảng `NumPy` của `DataFrame`.

- `df.values`: Trả về một mảng `NumPy`. (Cách cũ hơn, vẫn hoạt động)
- `df.to_numpy()`: Trả về một mảng NumPy. (Cách được khuyến nghị hiện nay vì rõ ràng hơn về hành vi).

▫ **Bước 2:** Gọi phương thức `.tolist()` trên mảng `NumPy` đó. Mảng `NumPy` có phương thức `.tolist()` riêng để chuyển đổi nó thành list Python (hoặc `list` các list nếu mảng là đa chiều).

- Ví dụ:

```
import pandas as pd
import numpy as np
# Tạo một DataFrame ví dụ
data = {'Tên': ['An', 'Bình', 'Cường'],
        'Tuổi': [22, 25, 21],
        'Điểm': [8.5, 9.0, 7.5]}
df = pd.DataFrame(data)
print("DataFrame gốc:")
print(df)
# Cách 1: Sử dụng .values.tolist()
list1 = df.values.tolist()
print("\nDataFrame chuyển thành list các list (dùng
                                          .values.tolist()):")
print(list1)
print("\nKiểu của list1:", type(list1))
if all(type(list1[i])==list for i in range(len(list1))):
    print("Kiểu của tất cả các phần tử con ĐỀU là list")
else:
    print("Trong list1 có phần tử con KHÔNG là list")
# Cách 2: Sử dụng .to_numpy().tolist() (khuyến nghị)
list2 = df.to_numpy().tolist()
print("\nDataFrame chuyển thành list các list (dùng
                                      .to_numpy().tolist()):")
print(list2)
```

Kết quả:

```
DataFrame gốc:
     Tên  Tuổi  Điểm
0     An    22   8.5
1   Bình    25   9.0
2  Cường    21   7.5

DataFrame chuyển thành list các list (dùng .values.tolist()):
[['An', 22, 8.5], ['Bình', 25, 9.0], ['Cường', 21, 7.5]]

Kiểu của list1: <class 'list'>
Kiểu của tất cả các phần tử con ĐỀU là list

DataFrame chuyển thành list các list (dùng .to_numpy().tolist()):
[['An', 22, 8.5], ['Bình', 25, 9.0], ['Cường', 21, 7.5]]
```

- **Mảng `NumPy`:** Thuộc tính `.values` của một `DataFrame` trả về một mảng `NumPy` đại diện cho dữ liệu. Mảng `NumPy` này có phương thức `.tolist()` để chuyển đổi

nó thành một danh sách các danh sách Python (trong đó mỗi danh sách con là một hàng của `DataFrame`).

```
import pandas as pd
data = {'col_A': [10, 20], 'col_B': ['x', 'y']}
df = pd.DataFrame(data)
print("DataFrame:")
print(df)
# Chuyển toàn bộ DataFrame thành list của các list (rows)
list_of_rows = df.values.tolist()
print("\nDataFrame dưới dạng list của các list:", list_of_rows)
# Output: [[10, 'x'], [20, 'y']]
```

3.7.25. unique()

- Trả về các giá trị duy nhất trong `Series`.
- `seriesName.unique()`

3.7.26. value_counts()

- *Giới thiệu:*
  - Được sử dụng để đếm số lần xuất hiện của mỗi giá trị duy nhất trong một đối tượng `Series`. Kết quả trả về cũng là một `Series`, trong đó:
    - `Index` (chỉ mục) của Series kết quả là các giá trị duy nhất có trong `Series` gốc.
    - `Values` (giá trị) của `Series` kết quả là số lần xuất hiện (tần suất) tương ứng của các giá trị duy nhất đó.
  - Mặc định, kết quả sẽ được sắp xếp theo số lần xuất hiện giảm dần, nghĩa là giá trị xuất hiện nhiều nhất sẽ đứng đầu.
- *Cú pháp*:

```
pandas.Series.value_counts(normalize=False, sort=True,
        ascending=False, bins=None, dropna=True)
```

trong đó:

- *normalize*:
  - Nếu *False* (mặc định): Trả về số lần xuất hiện (tần suất tuyệt đối) của mỗi giá trị.
  - Nếu *True*: Trả về tần suất tương đối (tỷ lệ phần trăm) của mỗi giá trị. Tức là số lần xuất hiện của mỗi giá trị sẽ được chia cho tổng số lượng phần tử (không tính *NaN* nếu *dropna=True*).
- *sort*:
  - Nếu *True* (mặc định): Sắp xếp kết quả theo số lần xuất hiện.
  - Nếu *False*: Không sắp xếp kết quả theo số lần xuất hiện (thứ tự có thể không đoán trước được hoặc theo thứ tự xuất hiện lần đầu của giá trị duy nhất).
- *ascending*: Chỉ có tác dụng khi *sort=True*.
  - Nếu *False* (mặc định): Sắp xếp theo thứ tự giảm dần (giá trị xuất hiện nhiều nhất đứng đầu).
  - Nếu *True*: Sắp xếp theo thứ tự tăng dần (giá trị xuất hiện ít nhất đứng đầu).

- `bins` (mặc định là `None`):
  - Tham số này dùng để chia các giá trị số thành các khoảng (`bins`) rời rạc trước khi đếm. Điều này rất hữu ích khi muốn xem phân phối tần suất của dữ liệu số liên tục.
  - Nếu là một số nguyên, nó xác định số lượng `bin` có độ rộng bằng nhau trong phạm vi của Series.
  - Nếu là một chuỗi các số, nó xác định các cạnh của `bin` (ví dụ: `[0, 10, 20, 30]` sẽ tạo ra các bin `(0, 10]`, `(10, 20]`, `(20, 30]`).
  - Khi `bins` được sử dụng, `index` của `Series` kết quả sẽ là các đối tượng `Interval` đại diện cho các khoảng đó.

- `dropna`:
  - Nếu `True` (mặc định): Không tính các giá trị bị thiếu (`NaN` - *Not a Number*) vào trong kết quả đếm.
  - Nếu `False`: Tính cả các giá trị `NaN` (nếu có) và hiển thị số lần xuất hiện của chúng.

- *Ứng dụng của* `value_counts()`*:*

  - ***Phân tích tần suất*:** Nhanh chóng hiểu được sự phân bổ của các giá trị khác nhau trong một cột dữ liệu. Ví dụ, trong một cột "Thành phố", có thể xem thành phố nào xuất hiện nhiều nhất.
  - ***Tìm các giá trị phổ biến nhất/ít phổ biến nhất*:** Dễ dàng xác định các mục hàng đầu hoặc các mục hiếm gặp.
  - ***Kiểm tra dữ liệu*:** Phát hiện các giá trị bất thường hoặc các vấn đề về chất lượng dữ liệu (ví dụ: một giá trị lẽ ra phải là duy nhất nhưng lại xuất hiện nhiều lần).
  - ***Chuẩn bị dữ liệu cho biểu đồ*:** Kết quả của `value_counts()` thường được dùng trực tiếp để vẽ biểu đồ cột (bar chart) thể hiện tần suất.
  - ***Phân tích dữ liệu phân loại*** *(Categorical Data)*: Đây là một trong những công cụ chính để khám phá dữ liệu dạng phân loại.

- *Một số ví dụ:*

  - *Ví dụ 1*: Đếm giá trị trong một Series chứa chuỗi

Mã lệnh	*Kết quả*

`import pandas as pd`
`Sdata = pd.Series(['táo', 'cam', 'táo',`
`'chuối', 'cam', 'táo', 'lê'])`
`print("Series gốc:")`
`print(Sdata)`
 	
`counts_str = Sdata.value_counts()`
`print("\nĐếm giá trị (mặc định):")`
`print(counts_str)`
`print('-----')`
	`Đếm giá trị (mặc định):`
	`táo 3`
	`cam 2`
	`chuối 1`
	`lê 1`
	`Name: count, dtype: int64`
	`-----`

`counts_ frequency =`
`Sdata.value_counts(normalize=True)`
`print("Đếm giá trị (tần suất tương đối):")`
`print(counts_ frequency)`
	`Đếm giá trị (tần suất tương đối):`
	`táo 0.428571`
	`cam 0.285714`
	`chuối 0.142857`
	`lê 0.142857`
	`Name: proportion, dtype: float64`

`counts_ascending =`	`Đếm giá trị (sắp xếp tăng dần):` `chuối 1`

`Sdata.value_counts(ascending=True)`
`print("Đếm giá trị (sắp xếp tăng dần):")`
`print(counts_ascending)`
	`lê 1`
	`cam 2`
	`táo 3`
	`Name: count, dtype: int64`

- *Ví dụ 2*: Đếm giá trị trong một `Series` số và xử lý `NaN`

Mã lệnh	*Kết quả*

`import pandas as pd`
`Snum = pd.Series([1, 2, 2, 3, 3, 3,`
`np.nan, 4, np.nan, 2])`
`print("\nSeries số với NaN:")`
`print(Snum)`
	`Series số với NaN:`
	`0 1.0`
	`1 2.0`
	`2 2.0`
	`3 3.0`
	`4 3.0`
	`5 3.0`
	`6 NaN`
	`7 4.0`
	`8 NaN`
	`9 2.0`
	`dtype: float64`

`#Mặc định dropna=True`
`counts_dropna_True = Snum.value_counts()`
`print("\nĐếm giá trị số (bỏ qua NaN):")`
`print(counts_dropna_True)`
	`Đếm giá trị số (bỏ qua NaN):`
	`2.0 3`
	`3.0 3`
	`1.0 1`
	`4.0 1`
	`Name: count, dtype: int64`

`counts_dropna_False =`
`Snum.value_counts(dropna=False)`
`print("\nĐếm giá trị số (bao gồm NaN):")`
`print(counts_dropna_False)`
	`Đếm giá trị số (bao gồm NaN):`
	`2.0 3`
	`3.0 3`
	`NaN 2`
	`1.0 1`
	`4.0 1`
	`Name: count, dtype: int64`

- *Ví dụ 3*: Sử dụng tham số 'bins' cho dữ liệu số

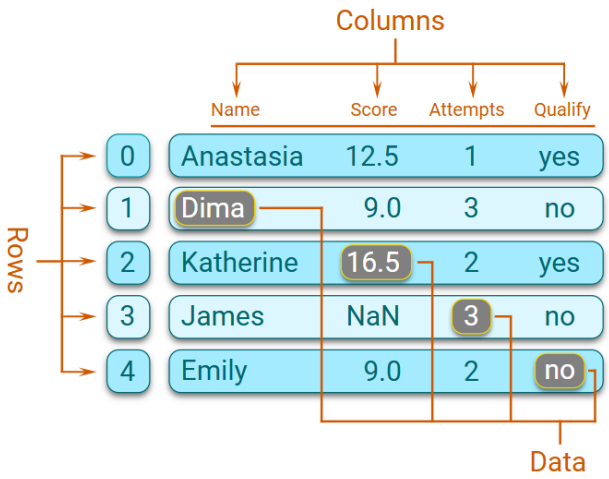
Mã lệnh	*Kết quả*

`import pandas as pd`
`ages = pd.Series([22, 25, 31, 45, 23,`
`35, 52, 28, 60, 33, 38, 41, 29])`
`print("\nSeries tuổi:")`
`print(ages)`
	`Series tuổi:`
	`0 22`
	`1 25`
	`2 31`
	`3 45`
	`4 23`
	`5 35`
	`6 52`
	`7 28`
	`8 60`
	`9 33`
	`10 38`
	`11 41`
	`12 29`
	`dtype: int64`

`# Chia thành 3 khoảng bằng nhau`
`age_bins_counts =`
`ages.value_counts(bins=3)`
`print("\nĐếm tuổi theo 3 khoảng`
`(bins=3):")`
`print(age_bins_counts)`
	`Đếm tuổi theo 3 khoảng (bins=3):`
	`(21.961, 34.667] 7`
	`(34.667, 47.333] 4`
	`(47.333, 60.0] 2`
	`Name: count, dtype: int64`

`custom_bins = [20, 30, 40, 50, 60, 70]`
`# sort=False để giữ thứ tự bins`
`age_custom_bins_counts =`
`ages.value_counts(bins=custom_bins,`
`sort=False)`
`print("\nĐếm tuổi theo các khoảng`
`tùy chỉnh:")`
`print(age_custom_bins_counts)`
	`Đếm tuổi theo các khoảng tùy chỉnh:`
	`(19.999, 30.0] 5`
	`(30.0, 40.0] 4`
	`(40.0, 50.0] 2`
	`(50.0, 60.0] 2`
	`(60.0, 70.0] 0`
	`Name: count, dtype: int64`

4. PANDAS DATAFRAME



Pandas DataFrame

4.1. Giới thiệu

pandas.DataFrame (gọi tắt là DataFrame) là một trong những cấu trúc dữ liệu cốt lõi và quan trọng nhất trong thư viện Pandas của Python. Nó được thiết kế để làm việc với dữ liệu có cấu trúc dạng bảng (tabular data) một cách hiệu quả và linh hoạt, tương tự như một bảng tính trong Excel, một bảng trong cơ sở dữ liệu SQL, hoặc một data.frame trong ngôn ngữ R.

Hãy hình dung DataFrame như một cái bảng hai chiều, có nhãn cho cả hàng và cột.

Mối quan hệ giữa Series và DataFrame: Điều quan trọng cần hiểu là một DataFrame thực chất là một tập hợp các Series có cùng một chỉ mục (index). Mỗi cột trong một DataFrame là một đối tượng Series. Khi chọn một cột duy nhất từ một DataFrame, kết quả thường là một Series.

4.2. Các đặc điểm chính của một DataFrame:

(i).- **Cấu trúc hai chiều (2D):** Dữ liệu được tổ chức thành các hàng (rows) và các cột (columns).

(ii).- **Nhãn (Labels):**

- **Index (Chỉ mục hàng):** Mỗi hàng có một nhãn định danh duy nhất, gọi là index. Index có thể là số nguyên (mặc định là từ 0, 1, 2,...), ngày tháng, chuỗi ký tự, hoặc các kiểu dữ liệu khác. Nó giúp truy cập và định vị các hàng một cách nhanh chóng.
- **Columns (Tên cột):** Mỗi cột cũng có một tên (nhãn) riêng.

(iii).- **Kiểu dữ liệu linh hoạt cho mỗi cột:** Mỗi cột trong DataFrame có thể chứa một kiểu dữ liệu khác nhau (int, float, string, boolean, datetime, …). Tuy nhiên, tất cả các giá trị trong cùng một cột thường có cùng kiểu dữ liệu.

(iv).- **Kích thước có thể thay đổi (Size-mutable):** có thể thêm hoặc xóa các cột, các hàng, khỏi DataFrame sau khi nó đã được tạo.

(v).- **Giá trị có thể thay đổi (Value-mutable):** có thể thay đổi các giá trị bên trong DataFrame.

(vi).- **Khả năng xử lý dữ liệu thiếu** *(Missing Data)*: Pandas cung cấp các công cụ mạnh mẽ để xử lý các giá trị bị thiếu (thường được biểu diễn bằng NaN - Not a Number).

(vii).- **Tích hợp nhiều chức năng:** DataFrame đi kèm với rất nhiều phương thức và thuộc tính để thực hiện các thao tác dữ liệu phổ biến như:

- Lựa chọn và lọc dữ liệu (selection, filtering)
- Sắp xếp dữ liệu (sorting)
- Nhóm dữ liệu (grouping by)
- Hợp nhất và nối các bảng (merging, joining, concatenating)
- Tính toán thống kê (statistical operations)
- Xử lý dữ liệu chuỗi, ngày tháng
- Đọc và ghi dữ liệu từ nhiều định dạng tệp khác nhau (CSV, Excel, SQL, JSON,...)

4.3. Cấu trúc bên trong (một cách trừu tượng):

Có thể coi một DataFrame như một tập hợp các đối tượng pandas.Series có cùng một index. Mỗi cột trong DataFrame thực chất là một pandas.Series.

```
          Cột A (Series)      Cột B (Series)      Cột C (Series)
Index 0      Giá trị A0          Giá trị B0          Giá trị C0
Index 1      Giá trị A1          Giá trị B1          Giá trị C1
Index 2      Giá trị A2          Giá trị B2          Giá trị C2
...                 ...                 ...                 ...
```

4.4. Cách tạo một DataFrame

Có nhiều cách để tạo một `DataFrame` trong Pandas:

4.4.1. Từ một Dictionary của các list hoặc NumPy array

- Các khóa (`keys`) của `dictionary` sẽ trở thành tên cột.
- Các giá trị (`values`) là các `list/array` sẽ trở thành dữ liệu của các cột tương ứng. Tất cả các `list/array` phải có cùng độ dài.
- Ví dụ:

```
import pandas as pd
import numpy as np

data_dict = {'col1': [1, 2, 3, 4],
             'col2': ['a', 'b', 'c', 'd'],
             'col3': np.array([10.1, 10.2, 10.3, 10.4])}
df_from_dict = pd.DataFrame(data_dict)
print(df_from_dict)
```

Kết quả:

```
   col1 col2  col3
0     1    a  10.1
1     2    b  10.2
2     3    c  10.3
3     4    d  10.4
```

4.4.2. Từ một list của các Dictionary

- Mỗi `dictionary` trong `list` đại diện cho một hàng.
- Các khóa của `dictionary` sẽ trở thành tên cột. Nếu một `dictionary` thiếu một khóa nào đó, giá trị `NaN` sẽ được điền vào ô tương ứng.
- Ví dụ:

```
data_list_of_dicts =[{'name':'Alice', 'age':30, 'city':'New York'},
                     {'name':'Bob', 'age':24, 'city':'Paris'},
                     {'name':'Charlie', 'age':35}] #thiếu 'city'
df_from_list_dicts = pd.DataFrame(data_list_of_dicts)
print(df_from_list_dicts)
```

Kết quả:

```
      name  age      city
0    Alice   30  New York
1      Bob   24     Paris
2  Charlie   35       NaN
```

4.4.3. Từ một NumPy array hai chiều

- Có thể cung cấp tên cột và index một cách tùy chọn.
- Ví dụ:

```
data_numpy = np.array([[1, 'a', 10.1],
                       [2, 'b', 10.2],
                       [3, 'c', 10.3]])
df_from_numpy = pd.DataFrame(data_numpy, columns=['ID', 'Category',
                         'Value'], index=['row1', 'row2', 'row3'])
print(df_from_numpy)
```

Kết quả:

```
      ID Category Value
row1   1        a  10.1
row2   2        b  10.2
row3   3        c  10.3
```

4.4.4. Từ một đối tượng pandas.Series

- Một Series sẽ trở thành một cột của DataFrame.
- Ví dụ:

```
s = pd.Series([1, 2, 3], name='MyColumn')
df_from_series = pd.DataFrame(s)
print(df_from_series)
```

Kết quả:

```
   MyColumn
0         1
1         2
2         3
```

4.4.5. Từ việc đọc file

- Đây là cách rất phổ biến:
  - `pd.read_csv('file.csv')`
  - `pd.read_excel('file.xlsx')`
  - `pd.read_sql(query, connection_object)`
  - `pd.read_json('file.json')`
  - `...`
- Ví dụ: Giả sử có file 'data.csv' với nội dung:

name	age	city
Alice	30	New York
Bob	24	Paris

```
df_from_csv = pd.read_csv('data.csv')
print(df_from_csv)
```

4.4.6. Một số thao tác cơ bản với DataFrame

Các nội dung này sẽ được giới thiệu chi tiết trong các phần thuộc tính, phương thức của DataFrame.

4.4.6.1. Xem dữ liệu

- `df.head(n)` : Xem n hàng đầu tiên (mặc định là 5).
- `df.tail(n)` : Xem n hàng cuối cùng (mặc định là 5).
- `df.info()` : Xem thông tin tổng quan về DataFrame (kiểu dữ liệu các cột, số lượng giá trị không thiếu, dung lượng bộ nhớ).
- `df.describe()` : Xem các thống kê mô tả cơ bản cho các cột số (như count, mean, std, min, max).
- `df.shape` : Trả về một tuple chứa số hàng và số cột (số_hàng, số_cột).
- `df.columns` : Trả về danh sách tên các cột.
- `df.index` : Trả về đối tượng Index chứa các nhãn hàng.
- `df.dtypes` : Trả về kiểu dữ liệu của mỗi cột.

4.4.6.2. Lựa chọn dữ liệu (Indexing and Selecting Data)

- Chọn một cột: `df['tên_cột']` (trả về một `Series`) hoặc `df.tên_cột` (nếu tên cột hợp lệ).
- Chọn nhiều cột: `df[['cột1', 'cột2']]` (trả về một `DataFrame` mới).
- Chọn hàng theo nhãn (index): `df.loc['nhãn_hàng']` hoặc `df.loc[['nhãn1', 'nhãn2']]`.
- Chọn hàng theo vị trí số nguyên: `df.iloc[vị_trí_hàng]` hoặc `df.iloc[[vị_trí1, vị_trí2]]`.
- Chọn cả hàng và cột
  - `df.loc['nhãn_hàng', 'tên_cột']`
  - `df.loc[['nhãn1', 'nhãn2'], ['cộtA', 'cộtB']]`
  - `df.iloc[vị_trí_hàng, vị_trí_cột]`
  - `df.iloc[[vị_trí_hàng1, vị_trí_hàng2], [vị_trí_cột1, vị_trí_cột2]]`
- Lựa chọn có điều kiện (Boolean indexing)

```
# Ví dụ chọn những người có tuổi >18:
df_adults = df[df['age'] > 18]
# Ví dụ chọn những người ở thành phố New York
df_specific_city = df[df['city'] == 'New York']
```

4.4.6.3. Thêm/Xóa cột

- Thêm cột mới : `df['tên_cột_mới'] = giá_trị_hoặc_Series`
- Xóa cột :

```
del df['tên_cột']
df.drop('tên_cột', axis=1, inplace=True)
```

Trong đó: `axis=1` cho cột, `inplace=True` để thay đổi trực tiếp `df` hoặc .

4.4.6.4. Xử lý dữ liệu thiếu

- `df.isnull()`: Kiểm tra các giá trị bị thiếu, trả về DataFrame boolean.
- `df.dropna()`: Loại bỏ các hàng/cột có giá trị thiếu.
- `df.fillna(giá_trị)`: Điền các giá trị thiếu bằng một giá trị cụ thể.

4.5. Công dụng phổ biến của DataFrame

- *Làm sạch và chuẩn bị dữ liệu* (*Data Cleaning and Preparation*): Xử lý dữ liệu thiếu, chuyển đổi kiểu dữ liệu, loại bỏ dữ liệu không cần thiết.
- *Phân tích dữ liệu khám phá* (*Exploratory Data Analysis - EDA*): Tính toán thống kê, trực quan hóa dữ liệu để hiểu rõ hơn về tập dữ liệu.
- *Trích xuất đặc trưng* (*Feature Engineering*): Tạo ra các cột mới (đặc trưng) từ dữ liệu hiện có để cải thiện mô hình học máy.
- *Làm đầu vào cho các thư viện học máy*: Nhiều thư viện học máy như Scikit-learn chấp nhận DataFrame làm đầu vào.

4.6. Một số thuộc tính thường dùng

4.6.1. Values

4.6.1.1. Giới thiệu

- Thuộc tính DataFrame.values giúp truy cập dữ liệu thô bên trong một DataFrame dưới dạng một mảng NumPy (NumPy array).
- Mục đích chính: DataFrame.values cung cấp một "khung nhìn" (view) hoặc một bản sao (copy) của dữ liệu trong DataFrame dưới dạng cấu trúc mảng đa chiều của NumPy. Điều này rất hữu ích khi cần thực hiện các phép toán tối ưu hóa của NumPy trên dữ liệu hoặc khi cần truyền dữ liệu sang các thư viện khác thường làm việc với mảng NumPy (như scikit-learn, TensorFlow, PyTorch).
- Đặc điểm quan trọng:
  - **Kiểu trả về:** Luôn là một numpy.ndarray.
    - Nếu DataFrame chỉ có một kiểu dữ liệu (ví dụ: tất cả các cột đều là số nguyên hoặc tất cả đều là số thực), mảng NumPy trả về sẽ có kiểu dữ liệu đó.
    - Nếu DataFrame có nhiều kiểu dữ liệu khác nhau (ví dụ: một cột là số, một cột là chuỗi, một cột là boolean), mảng NumPy trả về sẽ có kiểu dữ liệu "object". Điều này là do NumPy array yêu cầu tất cả các phần tử phải có cùng một kiểu dữ liệu. Kiểu object là kiểu chung nhất có thể chứa các kiểu Python khác nhau.
  - **Không bao gồm Index và Columns:** Mảng NumPy trả về bởi .values chỉ chứa dữ liệu của DataFrame, không bao gồm các nhãn của index (hàng) hay tên của cột.
  - **View vs. Copy (Lưu ý quan trọng):**
    - Trong các phiên bản Pandas cũ, .values thường trả về một "view" của dữ liệu nếu DataFrame đồng nhất về kiểu dữ liệu. Điều này có nghĩa là khi thay đổi

giá trị trong mảng NumPy, thay đổi đó cũng có thể ảnh hưởng đến DataFrame gốc và ngược lại.

- Tuy nhiên, hành vi này không được đảm bảo và có thể thay đổi. Đặc biệt, nếu DataFrame có nhiều kiểu dữ liệu (resulting in an object-dtype array), .values thường trả về một bản sao.
- Khuyến nghị hiện tại (từ Pandas 1.0 trở đi): Để đảm bảo nhận được một mảng NumPy một cách rõ ràng và tránh các cảnh báo hoặc hành vi không nhất quán trong tương lai, Pandas khuyến khích sử dụng phương thức DataFrame.to_numpy() thay vì thuộc tính .values. to_numpy() cho phép kiểm soát rõ ràng hơn việc tạo ra một view hay một copy và cách xử lý kiểu dữ liệu.

4.6.1.2. Ví dụ

(i).- Ví dụ 1: DataFrame với kiểu dữ liệu đồng nhất

```
import pandas as pd
import numpy as np
data1 = {'col1': [1, 2, 3],
         'col2': [4, 5, 6]}
df1 = pd.DataFrame(data1)

print("DataFrame 1:")
print(df1)
print("\nSử dụng thuộc tính df1.values:")
print(df1.values)
print("Kiểu của df1.values là:", type(df1.values))
print("Kiểu dữ liệu (dtype) của mảng NumPy:", df1.values.dtype)

# Output:
DataFrame 1:
   col1  col2
0     1     4
1     2     5
2     3     6
Sử dụng thuộc tính df1.values:
[[1 4]
 [2 5]
 [3 6]]
Kiểu của df1.values là: <class 'numpy.ndarray'>
Kiểu dữ liệu (dtype) của mảng NumPy: int64
```

(ii).- Ví dụ 2: DataFrame với kiểu dữ liệu hỗn hợp

```
import pandas as pd
import numpy as np
data2 = {'name': ['Alice', 'Bob', 'Charlie'],
         'age': [25, 30, 35],
         'is_student': [True, False, True]}
df2 = pd.DataFrame(data2)

print("DataFrame 2:")
print(df2)
```

```
print("\nSử dụng thuộc tính df2.values:")
print(df2.values)
print("Kiểu của df1.values là:", type(df2.values))
print("Kiểu dữ liệu (dtype) của mảng NumPy:", df2.values.dtype)
# Output:
DataFrame 2:
      name  age  is_student
0    Alice   25        True
1      Bob   30       False
2  Charlie   35        True

Sử dụng thuộc tính df2.values:
[['Alice' 25 True]
 ['Bob' 30 False]
 ['Charlie' 35 True]]
Kiểu của df1.values là: <class 'numpy.ndarray'>
Kiểu dữ liệu (dtype) của mảng NumPy: object
```

4.6.1.3. Khi nào nên sử dụng DataFrame.values (hoặc tốt hơn là DataFrame.to_numpy())

- Khi cần truyền dữ liệu vào các hàm của thư viện NumPy.
- Khi làm việc với các thư viện học máy như Scikit-learn, vì hầu hết các thuật toán trong đó đều mong đợi đầu vào là mảng NumPy.
- Khi muốn thực hiện các thao tác lặp hiệu suất cao trên dữ liệu mà không cần đến các chức năng của DataFrame (như nhãn - labels).

4.6.1.4. Về hiệu suất và kiểu dữ liệu

- Nếu DataFrame có nhiều kiểu dữ liệu, mảng NumPy kết quả sẽ có kiểu object. Các phép toán trên mảng NumPy kiểu object thường chậm hơn đáng kể so với các mảng có kiểu dữ liệu cụ thể (như int64 hay float64) vì chúng liên quan đến việc trỏ tới các đối tượng Python riêng lẻ.
- Nếu có thể, hãy cố gắng làm việc với các cột có kiểu dữ liệu đồng nhất hoặc chuyển đổi chúng trước khi gọi .values hoặc .to_numpy() nếu hiệu suất là yếu tố quan trọng.

4.6.2. shape

- Cho biết số lượng dòng và số lượng cột có trong DataFrame.
- Ví dụ:

Mã lệnh	*Kết quả*
`mydata.shape -- hoặc print(mydata.shape)`	`(1173, 16)`

4.7. Một số phương thức thường dùng

4.7.1. at()

4.7.1.1. Giới thiệu

DataFrame.at trong Pandas là một phương thức được sử dụng để truy cập một giá trị đơn lẻ trong DataFrame bằng cách sử dụng nhãn (label) của hàng và cột. Đây là một cách

rất nhanh để lấy hoặc thiết lập một giá trị cụ thể khi biết chính xác vị trí của nó dựa trên tên hàng và tên cột.

4.7.1.2. Đặc điểm chính của DataFrame.at

- **Truy cập theo nhãn (Label-based):** DataFrame.at yêu cầu cung cấp tên của hàng và tên của cột để xác định vị trí của giá trị.
- **Truy cập giá trị đơn lẻ (Scalar value):** DataFrame.at được tối ưu hóa để truy cập hoặc thiết lập duy nhất một giá trị tại một thời điểm. DataFrame.at không trả về một Series hay một DataFrame con.
- **Tốc độ:** DataFrame.at thường xử lý nhanh hơn so với DataFrame.loc khi chỉ cần truy cập một phần tử duy nhất, vì nó bỏ qua một số bước kiểm tra và xử lý phức tạp hơn của loc.
- **Chỉ chấp nhận nhãn:** Không giống như iloc (truy cập theo vị trí số nguyên), at chỉ làm việc với nhãn.

4.7.1.3. Cú pháp

- Để **truy cập** một giá trị:

```
ten_bien = df.at[nhan_hang, nhan_cot]
```

- Để **thiết lập** một giá trị mới:

```
df.at[nhan_hang, nhan_cot] = gia_tri_moi
```

Trong đó:

- *ten_bien*: tên của biến sẽ nhận giá trị trả về từ dataframe.at
- *df*: là tên của DataFrame.
- *nhan_hang*: là nhãn (tên) của hàng muốn truy cập.
- *nhan_cot*: là nhãn (tên) của cột muốn truy cập.
- *gia_tri_moi*: là giá trị muốn gán cho phần tử đó.

4.7.1.4. Ví dụ

Giả sử chúng ta có một DataFrame như sau:

Mã lệnh

```
data = {'Ten': ['An', 'Binh', 'Chi'],
        'Tuoi': [25, 30, 22],
        'ThanhPho': ['Hanoi', 'TP.HCM',
                                'Da Nang']}
df = pd.DataFrame(data, index=['nguoi1',
                      'nguoi2', 'nguoi3'])
print(df)
```

Kết quả

	Ten	*Tuoi*	*ThanhPho*
nguoi1	An	25	Hanoi
nguoi2	Binh	30	TP.HCM
nguoi3	Chi	22	DaNang

- **Sử dụng DataFrame.at để truy cập giá trị:**

```
# Lấy tuổi của 'Binh'
tuoi_binh = df.at['nguoi2', 'Tuoi']
print(f"Tuổi của Bình: {tuoi_binh}")
# Output: Tuổi của Bình: 30

# Lấy thành phố của 'An'
thanh_pho_an = df.at['nguoi1', 'ThanhPho']
print(f"Thành phố của An: {thanh_pho_an}")
```

```
# Output: Thành phố của An: Hanoi
```

- **Sử dụng DataFrame.at để thiết lập giá trị:**

Mã lệnh

```
# Cập nhật tuổi của 'Chi' thành 23
df.at['nguoi3', 'Tuoi'] = 28
print(df)
```

Kết quả

	Ten	Tuoi	ThanhPho
nguoi1	An	25	Hanoi
nguoi2	Binh	30	TP.HCM
nguoi3	Chi	**28**	DaNang

4.7.1.5. So sánh at với loc

- **at:**
  - Chỉ truy cập giá trị đơn lẻ.
  - Nhanh hơn cho việc truy cập một giá trị.
  - Chỉ chấp nhận nhãn.
- **loc:**
  - Có thể truy cập một giá trị đơn lẻ, một hàng, một cột, hoặc một "lát cắt" (slice) của DataFrame.
  - Linh hoạt hơn nhưng có thể chậm hơn một chút cho việc truy cập một giá trị đơn lẻ so với at.
  - Chấp nhận nhãn, danh sách nhãn, hoặc điều kiện boolean.

4.7.2. head()

- Trả về 5 dòng đầu tiên có trong DataFrame.

4.7.3. describe()

- Tính toán các số liệu thống kê khác nhau ngoại trừ NaN.
- Ví dụ

Mã lệnh

```
import pandas as pd
data =
 pd.read_csv(r"D:\\DataVietNam.csv")
print(data.describe())
```

Kết quả

	Time period	Observation Value
count	1173.000000	1.173000e+03
mean	2004.899403	6.820107e+05
std	5.341585	7.621509e+06
min	1987.000000	1.000000e-02
25%	2002.000000	1.470000e+01
50%	2006.000000	3.831000e+01
75%	2009.000000	9.700000e+01
max	2013.000000	1.385500e+08

4.7.4. drop()

4.7.4.1. Giới thiệu

DataFrame.drop được sử dụng để loại bỏ các hàng (rows) hoặc cột (columns) được chỉ định từ một DataFrame. Đây là một thao tác rất phổ biến trong quá trình làm sạch và chuẩn bị dữ liệu.

4.7.4.2. Đặc điểm chính của DataFrame.drop

- **Loại bỏ theo nhãn (label):** chỉ định các hàng hoặc cột cần xóa bằng tên nhãn của chúng (tên index cho hàng, tên cột cho cột).

- **Xác định trục (axis):** cần chỉ rõ muốn xóa hàng (axis=0 hoặc axis='index') hay xóa cột (axis=1 hoặc axis='columns').
- **Không thay đổi DataFrame gốc (mặc định):** Theo mặc định, drop trả về một DataFrame mới đã được loại bỏ các hàng/cột chỉ định, còn DataFrame gốc không bị thay đổi. Có thể thay đổi hành vi này bằng tham số inplace=True.
- **Xử lý lỗi:** có thể kiểm soát cách phương thức xử lý nếu nhãn được cung cấp không tồn tại trong DataFrame bằng tham số errors.

4.7.4.3. Cú pháp

```
df.drop(labels=None, axis=0, index=None, columns=None,
                    level=None, inplace=False, errors='raise')
```

trong đó:

- `labels`: Nhãn đơn hoặc danh sách các nhãn cần xóa.
- `axis`: Trục để xóa.
  - `0` hoặc `'index'`: Xóa hàng (mặc định).
  - `1` hoặc `'columns'`: Xóa cột.
- `index`: Cách khác để chỉ định nhãn hàng cần xóa (tương đương với labels khi `axis=0`). Có thể cung cấp một nhãn đơn hoặc một danh sách các nhãn.
- `columns`: Cách khác để chỉ định nhãn cột cần xóa (tương đương với labels khi `axis=1`). Có thể cung cấp một nhãn đơn hoặc một danh sách các nhãn.
- `inplace`: Kiểu boolean, mặc định là False.
  - Nếu `False` (mặc định): Trả về một bản sao của DataFrame với các hàng/cột đã bị xóa. DataFrame gốc không thay đổi.
  - Nếu `True`: Thực hiện thao tác xóa trực tiếp trên DataFrame gốc và trả về `None`.
- `errors`: Mặc định là `'raise'`.
  - `'raise'`: Nếu một trong các nhãn không được tìm thấy, một lỗi `KeyError` sẽ được nêu ra.
  - `'ignore'`: Nếu một nhãn không được tìm thấy, nó sẽ bị bỏ qua và không có lỗi nào được nêu ra.

4.7.4.4. Ví dụ

Cho một DataFrame như sau:

	Ten	Tuoi	ThanhPho	Diem
SV01	An	25	Hanoi	8.5
SV02	Binh	30	TP.HCM	9.0
SV03	Chi	22	Hue	7.5
SV04	Dung	35	Hanoi	8.0

```
import pandas as pd
du_lieu = {'Ten': ['An', 'Binh', 'Chi', 'Dung'],
           'Tuoi': [25, 30, 22, 35],
           'ThanhPho': ['Hanoi', 'TP.HCM', 'Hue', 'Hanoi'],
           'Diem': [8.5, 9.0, 7.5, 8.0]}
df = pd.DataFrame(du_lieu, index=['SV01', 'SV02', 'SV03', 'SV04'])
print("DataFrame gốc:")
print(df)
```

i. Xóa hàng (rows)

- ***Xóa một hàng theo nhãn index:***

```
df_xoa_hang = df.drop(labels='SV02') # Mặc định axis=0
print("\nSau khi xóa hàng SV02 (mặc định trả về DataFrame mới):")
print(df_xoa_hang)
print("\nDataFrame gốc không thay đổi:")
print(df)
```

- ***Xóa nhiều hàng theo danh sách nhãn index:***

```
df_xoa_nhieu_hang = df.drop(index=['SV01', 'SV04'])
print("\nSau khi xóa hàng SV01 và SV04:")
print(df_xoa_nhieu_hang)
```

- ***Xóa hàng và thay đổi trực tiếp DataFrame gốc:***

```
df.drop(labels='SV03', inplace=True) # axis=0 là mặc định
print("\nSau khi xóa hàng SV03 với inplace=True, dataframe gốc là:")
print(df)
'''Đến đây df đã bị thay đổi. Để chạy các ví dụ tiếp theo, ta nên tạo
lại df hoặc làm việc trên bản sao.
Khởi tạo lại df cho các ví dụ tiếp theo'''
df = pd.DataFrame(du_lieu, index=['SV01', 'SV02', 'SV03', 'SV04'])
```

ii. Xóa cột (columns)

- ***Xóa một cột theo tên cột***

```
df_xoa_cot = df.drop(labels='ThanhPho', axis=1)
print("\nSau khi xóa cột 'ThanhPho':")
print(df_xoa_cot)
```

- ***Xóa một cột bằng tham số columns***

```
df_xoa_cot_2 = df.drop(columns='Tuoi')
print("\nSau khi xóa cột 'Tuoi' (dùng tham số columns):")
print(df_xoa_cot_2)
```

- ***Xóa nhiều cột theo danh sách tên cột***

```
df_xoa_nhieu_cot = df.drop(labels=['Tuoi', 'Diem'], axis=1)
print("\nSau khi xóa cột 'Tuoi' và 'Diem':")
print(df_xoa_nhieu_cot)
```

- ***Xóa cột và thay đổi trực tiếp DataFrame gốc:***

```
df_copy = df.copy() # Tạo bản sao để dùng cho các ví dụ khác
df_copy.drop(columns=['Diem'], inplace=True)
print("\nSau khi xóa cột 'Diem' (inplace=True) trên bản sao:")
print(df_copy)
```

iii. Xử lý lỗi khi nhãn không tồn tại

- ***Mặc định (errors='raise'):***

```
try:
    df.drop(labels='CotKhongTonTai', axis=1)
except KeyError as e:
    print(f"\nLỗi khi xóa cột không tồn tại (errors='raise'): {e}")
```

- ***Bỏ qua lỗi (errors='ignore'):***

```
df_ignore_error = df.drop(labels='CotKhongTonTai', axis=1,
                                                   errors='ignore')
```

```
print("\nKhi xóa cột không tồn tại (errors='ignore'):")
print(df_ignore_error) '''DataFrame không thay đổi vì cột không tồn
                                                    tại để xóa'''
```

4.7.4.5. Lưu ý quan trọng

- Việc sử dụng `inplace=True` có thể tiện lợi, nhưng hãy cẩn thận vì nó sẽ thay đổi DataFrame gốc. Nếu cần giữ lại DataFrame gốc cho các thao tác khác, tốt hơn hết là gán kết quả của drop cho một biến DataFrame mới (hành vi mặc định khi `inplace=False`).
- `DataFrame.drop` hữu ích khi biết chính xác tên (nhãn) của các hàng hoặc cột muốn loại bỏ. Nếu muốn xóa dựa trên vị trí số nguyên, có thể cần kết hợp với `df.index` hoặc `df.columns` để lấy nhãn tương ứng, hoặc sử dụng các phương pháp khác như `slicing` với `iloc`.

4.7.5. dropna()

Giả sử đang có DataFrame như sau:

	A	B	C
0	1.0	2.0	3.0
1	4.0	NaN	NaN
2	NaN	NaN	NaN
3	5.0	6.0	7.0
4	8.0	NaN	9.0

- Xóa các dòng chứa giá trị NaN: df.
- Cú pháp:

```
DataFrameName.dropna(axis=0,how='any', thresh=None,
                                    subset=None, inplace=False)
```

trong đó:

- ***axis***: trục lấy giá trị int hoặc chuỗi cho hàng/cột. Đầu vào có thể là 0 hoặc 1 đối với Số nguyên và 'index' hoặc 'columns' đối với Chuỗi.
- ***how***: how chỉ lấy giá trị chuỗi có hai loại ('any' hoặc 'all'). 'any' giảm hàng/cột nếu BẤT CỨ giá trị nào là Null và 'all' chỉ xóa dòng nếu TẤT CẢ các giá trị là null.
- ***thresh***: lấy giá trị số nguyên ***n***, cho biết lượng dòng chứa giá trị ***NaN*** tối thiểu sẽ giảm xuống. Khi ***n*** > số lượng dòng có giá trị ***NaN***, sẽ xóa toàn bộ các dòng có trong DataFrame.
- ***subset***: Đó là một mảng giới hạn quá trình loại bỏ các hàng/cột được chuyển qua danh sách.
- ***inplace***: Đây là một boolean thực hiện các thay đổi trong *DataFrame* nếu được gán là `True`.

- Một số ví dụ:
  - Xóa tất cả các dòng có giá trị bị thiếu:

Mã lệnh

```
df.dropna()   #hoặc df.dropna(axis=0)
```

Kết quả

	A	B	C
0	1.0	2.0	3.0
3	5.0	6.0	7.0

- Xóa tất cả các cột có giá trị bị thiếu:

```
df.dropna(axis=1)
```

- Xóa tất cả các dòng có giá trị bị thiếu:

Mã lệnh

```
df.dropna(axis=1)
```

Kết quả

0
1
2
3
4

- Chỉ định số lượng dòng chứa giá trị NaN cần xóa. Khi đó sẽ ưu tiên xóa các dòng chứa nhiều giá NaN nhất.

Mã lệnh

```
df.dropna(thesh=2)
```

Kết quả

	A	B	C
0	1.0	2.0	3.0
3	5.0	6.0	7.0
4	8.0	NaN	9.0

```
df.dropna(thesh=4)
```

	A	B	C

- Chỉ xóa những dòng chứa toàn giá trị NaN:

Mã lệnh

```
df.dropna(axis=0, how='all', inplace=True)
```

Kết quả

	A	B	C
0	1.0	2.0	3.0
1	4.0	NaN	NaN
3	5.0	6.0	7.0
4	8.0	NaN	9.0

4.7.6. fillna()

- Chỉ định giá trị thay thế cho các giá trị NaN
- Cú pháp: `Df.fillna(value = giá trị)`
- Ví dụ: điền giá trị mean (tính theo từng cột) cho các giá trị NaN.

```
df.fillna(values=df.mean())
```

	A	B	C
0	1.0	2.0	3.000000
1	4.0	4.0	6.333333
2	4.5	4.0	6.333333
3	5.0	6.0	7.000000
4	8.0	4.0	9.000000

4.7.7. groupby()

- Được sử dụng để nhóm dữ liệu theo một hoặc nhiều khóa (keys) và thực hiện các phép toán tổng hợp (như sum, mean, count, ...), thống kê hoặc chuyển đổi trên từng nhóm. Đây là một công cụ cực kỳ hữu ích trong phân tích và xử lý dữ liệu.
- Quá trình hoạt động của groupby theo 3 bước chính:

  Bước 1: *Chia nhóm*: Chia dữ liệu theo các khóa đã chỉ định.

  Bước 2: *Áp dụng hàm*: Thực hiện tính toán hoặc phép toán tổng hợp trên từng nhóm.

  Bước 3: *Kết hợp kết quả*: Trả về một đối tượng pandas (Series, DataFrame, ...) chứa kết quả.

- Cú pháp

```
DataFrame.groupby(by=None, axis=0, level=None, as_index=True,
      sort=True, group_keys=True, observed=False, dropna=True)
```

  trong đó

  - *by* : Tên cột (hoặc danh sách tên cột) dùng để nhóm dữ liệu.
  - *axis* : Trục nhóm (0 cho hàng, 1 cho cột).
  - *as_index* : Nếu True, các cột nhóm trở thành chỉ số. Nếu False, các cột nhóm vẫn là một phần của DataFrame.
  - *level* : Dùng để nhóm dữ liệu theo các mức của MultiIndex.
  - *sort* : Sắp xếp các nhóm (True mặc định).

- Ví dụ
  - ***Nhóm và tính tổng giá trị:***

```
import pandas as pd
# Tạo DataFrame mẫu
data = {'Category': ['A', 'B', 'A', 'B', 'A'],
        'Values': [10, 20, 30, 40, 50]}
df = pd.DataFrame(data)
# Nhóm theo 'Category' và tính tổng
result = df.groupby('Category')['Values'].sum()
print(result)
# Kết quả:
# Category
# A    90
# B    60
# Name: Values, dtype: int64
```

  - ***Nhóm và tính giá trị trung bình:***

```
# Nhóm theo 'Category' và tính trung bình
```

```
result = df.groupby('Category')['Values'].mean()
print(result)
# Kết quả:
# Category
# A    30.0
# B    30.0
# Name: Values, dtype: float64
```

- ***Nhóm dữ liệu với as_index. Kết quả trả về dạng DataFrame:***

```
result = df.groupby('Category', as_index=False)['Values'].sum()
print(result)
# Kết quả:
#   Category  Values
# 0        A      90
# 1        B      60
result = df.groupby('Category', as_index=True)['Values'].sum()
print(result)
# Kết quả:
# Category
# A    90
# B    60
Name: Values, dtype: int64
```

- ***Nhóm theo nhiều cột:***

```
data = {'Category': ['A', 'B', 'A', 'B', 'A'],
        'Type': ['X', 'X', 'Y', 'Y', 'X'],
        'Values': [10, 20, 30, 40, 50]}
df = pd.DataFrame(data)

# Nhóm theo 'Category' và 'Type', tính tổng
result = df.groupby(['Category', 'Type'])['Values'].sum()
print(result)
# Kết quả:
# Category  Type
# A         X       60
#           Y       30
# B         X       20
#           Y       40
# Name: Values, dtype: int64
```

- ***Nhóm và đếm số lượng phần tử trong mỗi nhóm:***

```
# Đếm số lượng phần tử trong mỗi nhóm
result = df.groupby('Category')['Values'].count()
print(result)
# Kết quả:
# Category
# A    3
# B    2
# Name: Values, dtype: int64
```

- Lưu ý
  - *Xử lý dữ liệu thiếu (NaN)*: Dữ liệu có giá trị NaN trong cột dùng để nhóm sẽ bị loại bỏ mặc định. Có thể sử dụng tham số **`dropna=False`** để giữ lại.
  - *Kết hợp nhiều phép toán*: Có thể sử dụng **`.agg()`** để áp dụng nhiều phép toán khác nhau trên các cột.
- Hàm agg
  - Được sử dụng để áp dụng một hoặc nhiều hàm tổng hợp (aggregation functions như sum, mean, max, min), hoặc các hàm của người dùng khác cho từng nhóm dữ liệu được tạo bởi DataFrame.groupby(). Đây là một công cụ mạnh mẽ, giúp thực hiện nhiều phép toán khác nhau trên dữ liệu nhóm, và có thể tùy chỉnh kết quả theo từng cột.

- Tính linh hoạt của hàm agg: có thể:
  - Áp dụng một hàm tổng hợp cho tất cả các cột.
  - Áp dụng các hàm khác nhau cho từng cột.
  - Sử dụng các hàm của người dùng tự định nghĩa.
- Cú pháp

`DataFrame.groupby(...).agg(func)`

Trong đó, func: Có thể là:

  - Một hàm (vd: sum, mean, min, ...)
  - Danh sách các hàm (vd: ['sum', 'mean'])
  - dictionary ánh xạ tên từng cột với các hàm riêng biệt (vd: {'column1': 'sum', 'column2': ['mean', 'max']}).
- Ví dụ
  - *Áp dụng một hàm tổng hợp cho tất cả các cột:*

```
import pandas as pd
# Tạo DataFrame mẫu
data = {'Category': ['A', 'A', 'B', 'B', 'C'],
        'Values1': [10, 20, 30, 40, 50],
        'Values2': [1, 2, 3, 4, 5]}
df = pd.DataFrame(data)
# Nhóm theo 'Category' và tính tổng cho tất cả các cột
result = df.groupby('Category').agg('sum')
print(result)
# Kết quả:
#           Values1  Values2
# Category
# A              30        3
# B              70        7
# C              50        5
```

  - *Áp dụng nhiều hàm cho một cột:*

```
# Nhóm theo 'Category' và áp dụng nhiều hàm cho 'Values1'
result = df.groupby('Category')['Values1']
                                    .agg(['sum', 'mean', 'max'])
print(result)
# Kết quả:
#           sum  mean  max
# Category
# A          30  15.0   20
# B          70  35.0   40
# C          50  50.0   50
```

  - *Áp dụng các hàm khác nhau cho từng cột:*

```
# Áp dụng hàm sum cho 'Values1' và hàm mean cho 'Values2'
result = df.groupby('Category').agg({'Values1': 'sum',
                                     'Values2': 'mean'})
print(result)
# Kết quả:
#           Values1  Values2
# Category
# A              30      1.5
# B              70      3.5
# C              50      5.0
```

  - *Áp dụng danh sách hàm cho một cột cụ thể:*

```
# Áp dụng nhiều hàm cho cả hai cột
result = df.groupby('Category').agg({'Values1': ['sum', 'max'],
                                     'Values2': ['mean', 'min']})
print(result)
```

```
# Kết quả:
#            Values1          Values2
#               sum max       mean  min
# Category
# A              30  20        1.5    1
# B              70  40        3.5    3
# C              50  50        5.0    5
```

- *Sử dụng hàm của người dùng:*

```
# Định nghĩa hàm của người dùng: Tính độ lệch chuẩn
def range_func(x):
    return x.max() - x.min()

# Áp dụng hàm của người dùng cho 'Values'
result = df.groupby('Category')['Values'].agg(range_func)
print(result)
# Kết quả:
# Category
# A    10
# B    10
# C     0
# Name: Values, dtype: int64
# Áp dụng hàm của người dùng cho 'Values'
result = df.groupby('Category', as_index=False)['Values']
                                                .agg(range_func)
print(result)
# Kết quả:
#   Category  Values
# 0        A      40
# 1        B      20
```

- Lưu ý: Thay đổi chỉ số: Sử dụng as_index=False trong groupby để giữ cột nhóm trong kết quả thay vì chuyển thành chỉ số.

4.7.8. iat()

4.7.8.1. Giới thiệu

Phương thức DataFrame.iat (viết tắt của "integer at") trong Pandas là một công cụ rất hiệu quả để truy cập một giá trị đơn lẻ (scalar value) trong DataFrame bằng cách sử dụng vị trí số nguyên (integer position) của hàng và cột.

4.7.8.2. Đặc điểm chính của DataFrame.iat

- **Truy cập dựa trên vị trí số nguyên:** dựa trên chỉ số hàng và chỉ số cột (đều bắt đầu từ 0) được cung cấp để xác định vị trí của giá trị cần lấy hoặc thiết lập.
- **Chỉ dành cho giá trị đơn lẻ (Scalar Access):** DataFrame.iat được thiết kế đặc biệt để lấy hoặc đặt *một giá trị duy nhất* tại một thời điểm. Nó không thể dùng để lấy một lát cắt (*slice*) hay nhiều hàng/cột cùng lúc như .iloc.
- **Tốc độ rất nhanh:** Vì được tối ưu hóa cho việc truy cập giá trị đơn lẻ dựa trên số nguyên, `DataFrame.iat` thường nhanh hơn đáng kể so với các phương thức truy cập khác như .iloc hoặc .loc.
- **Không kiểm tra biên (Bound Checking):** DataFrame.iat thực hiện ít kiểm tra hơn so với `.iloc`, điều này góp phần vào tốc độ của nó nhưng ngược lại nếu được cung cấp vị trí nằm ngoài giới hạn của DataFrame, phương thức có thể gây ra lỗi hoặc hành vi không mong muốn nhanh hơn.

4.7.8.3. Cú pháp

- **Để truy cập (lấy) một giá trị:**

```
ten_bien = df.iat[vi_tri_hang, vi_tri_cot]
```

- **Để thiết lập (gán) một giá trị mới:**

```
df.iat[vi_tri_hang, vi_tri_cot] = gia_tri_moi
```

trong đó:

- *ten_bien*: tên của biến sẽ nhận giá trị.
- *df*: là tên của DataFrame.
- *vi_tri_hang*: Là chỉ số số nguyên của hàng (ví dụ: 0 cho hàng đầu tiên, 1 cho hàng thứ hai, v.v.).
- *vi_tri_cot*: Là chỉ số số nguyên của cột (ví dụ: 0 cho cột đầu tiên, 1 cho cột thứ hai, v.v.).
- *gia_tri_moi*: Giá trị muốn gán cho ô đó.

4.7.8.4. Ví dụ

```
import pandas as pd
# Tạo một DataFrame ví dụ
data = {'Tên': ['Tý', 'Sửu', 'Dần', 'Mẹo'],
        'Tuổi': [22, 25, 21, 28],
        'Điểm': [8.5, 9.0, 7.5, 8.8]}
df = pd.DataFrame(data, index=['SV01', 'SV02', 'SV03', 'SV04'])
print("DataFrame gốc:")
print(df)

''' 1. Truy cập giá trị bằng .iat
Lấy giá trị ở hàng đầu tiên (vị trí 0), cột thứ hai (vị trí 1 - cột
'Tuổi') '''
tuoi_an = df.iat[0, 1]
print(f"Tuổi của người đầu tiên ({df.iat[0, 0]}) dùng .iat[0, 1]:
{tuoi_an}")

''' Lấy giá trị ở hàng thứ ba (vị trí 2), cột cuối cùng (vị trí 2 -
cột 'Điểm') '''
diem_cuong = df.iat[2, 2]
print(f"Điểm của người thứ ba ({df.iat[2, 0]}) dùng .iat[2, 2]:
{diem_cuong}")

# 2. Thiết lập giá trị bằng .iat
print("DataFrame trước khi thay đổi:")
print(df)

''' Thay đổi tuổi của người thứ hai (Sửu - vị trí hàng 1) từ 25 thành
26 '''
df.iat[1, 1] = 26
print(f"\nDataFrame sau khi thay đổi tuổi của {df.iat[1, 0]}:")
print(df)

''' Thay đổi điểm của người cuối cùng (Mẹo - vị trí hàng 3) từ 8.8
thành 9.2 '''
```

```
df.iat[3, 2] = 9.2
print(f"\nDataFrame sau khi thay đổi điểm của {df.iat[3, 0]}:")
print(df)
```

4.7.8.5. So sánh DataFrame.iat với DataFrame.iloc và DataFrame.at:

- **`DataFrame.iat[row_pos, col_pos]`:**
  - Truy cập bằng vị trí số nguyên.
  - Chỉ cho giá trị đơn lẻ.
  - Nhanh nhất cho truy cập giá trị đơn lẻ.
- **`DataFrame.at[row_label, col_label]`:**
  - Truy cập bằng nhãn (label) của hàng và cột.
  - Chỉ cho giá trị đơn lẻ.
  - Rất nhanh cho truy cập giá trị đơn lẻ dựa trên nhãn.
- **`DataFrame.iloc[row_pos, col_pos]`:**
  - Truy cập bằng vị trí số nguyên.
  - Có thể truy cập giá trị đơn lẻ, một hàng, một cột, hoặc một lát cắt (slice) của DataFrame.
  - Nhanh, nhưng có thể chậm hơn .iat một chút cho việc truy cập giá trị đơn lẻ do có nhiều chức năng hơn.

4.7.8.6. Khi nào nên sử dụng DataFrame.iat

- Khi cần hiệu suất tối đa để đọc hoặc ghi một giá trị duy nhất trong DataFrame.
- Khi biết chính xác vị trí số nguyên (không phải nhãn) của hàng và cột muốn thao tác.
- Trong các vòng lặp hoặc các đoạn mã nhạy cảm về hiệu suất, nơi việc truy cập nhanh vào từng ô dữ liệu là quan trọng.

4.7.9. iterrows()

4.7.9.1. Giới thiệu

- `DataFrame.iterrows()` là một phương thức của đối tượng `DataFrame` trong thư viện Pandas của Python. Nó được sử dụng để lặp qua các hàng của `DataFrame` dưới dạng các cặp (`index`, `Series`).
- Điều này có nghĩa là trong mỗi vòng lặp, `iterrows()` sẽ trả về:
  - **`Index` của hàng đó:** Đây là nhãn chỉ mục của hàng (có thể là số nguyên, chuỗi, hoặc các kiểu dữ liệu khác tùy thuộc vào cách `DataFrame` được tạo).
  - **Một đối tượng `Series`:** Đối tượng `Series` này chứa tất cả dữ liệu của hàng đó, với `index` của `Series` là tên các cột của `DataFrame`.

4.7.9.2. Cú pháp

```
for index, row in df.iterrows():
    # Xử lý index và row ở đây
    print(f"Index: {index}")
    print(f"Row data:\n{row}\n")
    ''' Có thể truy cập từng giá trị trong hàng bằng cách sử
                                                dụng tên cột'''
```

```
    # value_of_column_A = row['column_A_name']
    # value_of_column_B = row['column_B_name']
```

trong đó:

- `df`: Là `DataFrame` cần lặp qua.
- `index`: Biến này sẽ nhận giá trị `index` của hàng hiện tại trong mỗi vòng lặp.
- `row`: Biến này sẽ nhận một đối tượng `pd.Series` chứa dữ liệu của hàng hiện tại.

4.7.9.3. Mục đích sử dụng

`iterrows()` thường được sử dụng khi cần thực hiện các thao tác trên từng hàng của `DataFrame` một cách riêng lẻ và cần truy cập vào cả `index` lẫn dữ liệu của hàng đó. Một số trường hợp sử dụng phổ biến bao gồm:

- *Truy cập và xử lý dữ liệu từng hàng*: Ví dụ, muốn kiểm tra một điều kiện nào đó trên mỗi hàng, hoặc thực hiện một phép tính dựa trên các giá trị của hàng đó.
- *Tạo dữ liệu mới dựa trên dữ liệu hàng hiện tại*: hoặc một cấu trúc dữ liệu khác dựa trên thông tin từ mỗi hàng.
- *Gọi các hàm hoặc API bên ngoài cho mỗi hàng*: khi cần gửi dữ liệu của từng hàng đến một hàm hoặc một dịch vụ bên ngoài để xử lý.

4.7.9.4. Ví dụ

```
import pandas as pd
data = {'Tên': ['An', 'Bình', 'Cường'],
        'Tuổi': [25, 30, 22],
        'Thành phố': ['Hà Nội', 'TP.HCM', 'Đà Nẵng']}
df = pd.DataFrame(data)
print("DataFrame gốc:")
print(df)
print("\nLặp qua các hàng sử dụng iterrows():")

for index, row in df.iterrows():
    print(f"\n--- Hàng thứ {index} ---")
    print(f"Index của hàng: {index}")
    print(f"Tên: {row['Tên']}")
    print(f"Tuổi: {row['Tuổi']}")
    # Kiểm tra xem người đó có trên 25 tuổi không
    if row['Tuổi'] > 25:
        print(f"{row['Tên']} > 25 tuổi.")
    else:
        print(f"{row['Tên']} <= 25 tuổi.")
    print(f"Thành phố: {row['Thành phố']}")
    # Cũng có thể xem toàn bộ đối tượng Series của hàng:
    print(f"---\nDữ liệu hàng (Series):\n{row}")
```

Kết quả:

```
DataFrame gốc:
     Tên  Tuổi Thành phố
0     An    25    Hà Nội
1   Bình    30    TP.HCM
2  Cường    22   Đà Nẵng

Lặp qua các hàng sử dụng iterrows():

--- Hàng thứ 0 ---
Index của hàng: 0
```

```
Tên: An
Tuổi: 25
An <= 25 tuổi.
Thành phố: Hà Nội
---
Dữ liệu hàng (Series):
Tên              An
Tuổi             25
Thành phố    Hà Nội
Name: 0, dtype: object

--- Hàng thứ 1 ---
Index của hàng: 1
Tên: Bình
Tuổi: 30
Bình > 25 tuổi.
Thành phố: TP.HCM
---
Dữ liệu hàng (Series):
Tên            Bình
Tuổi             30
Thành phố    TP.HCM
Name: 1, dtype: object

--- Hàng thứ 2 ---
Index của hàng: 2
Tên: Cường
Tuổi: 22
Cường <= 25 tuổi.
Thành phố: Đà Nẵng
---
Dữ liệu hàng (Series):
Tên            Cường
Tuổi              22
Thành phố    Đà Nẵng
Name: 2, dtype: object
```

4.7.9.5. Những lưu ý về iterrows()

- ***Hiệu suất (Performance)***: iterrows() không phải là cách hiệu quả nhất để lặp qua các hàng trong DataFrame, đặc biệt là với các DataFrame lớn vì iterrows() tạo ra một đối tượng Series mới cho mỗi hàng, điều này có thể tốn kém về mặt thời gian và bộ nhớ.
  - Nếu cần hiệu suất cao, hãy xem xét các phương pháp vector hóa của Pandas (thực hiện thao tác trên toàn bộ cột hoặc Series cùng một lúc) hoặc sử dụng df.apply() hoặc df.itertuples().
  - df.itertuples() thường nhanh hơn iterrows() vì nó trả về các namedtuples (tuple được đặt tên) thay vì Series.
- ***Kiểu dữ liệu (Data Types)***: Khi iterrows() tạo ra một Series cho mỗi hàng, nó có thể cố gắng "upcast" kiểu dữ liệu. Điều này có nghĩa là nếu một hàng có nhiều kiểu dữ liệu khác nhau (ví dụ: số nguyên và chuỗi), tất cả các giá trị trong Series kết quả có thể được chuyển đổi thành một kiểu dữ liệu chung hơn (thường là object), điều này có thể không phải lúc nào cũng là điều người lập trình mong đợi.
- ***Không nên sửa đổi DataFrame trong khi lặp***: iterrows() trả về một bản sao (copy) của dữ liệu cho mỗi hàng, không phải là một "view" (khung nhìn) trực tiếp vào

DataFrame. Do đó, việc cố gắng sửa đổi row bên trong vòng lặp sẽ không làm thay đổi DataFrame gốc.

- ***Nếu cần phải sửa đổi DataFrame***: hãy sử dụng các phương thức như df.loc[] hoặc df.iloc[] để truy cập và gán lại giá trị. Tuy nhiên, việc sửa đổi DataFrame trong khi lặp thường không được khuyến khích vì có thể dẫn đến hành vi không đoán trước được. Tốt hơn là thu thập các thay đổi cần thiết và áp dụng chúng sau vòng lặp.

4.7.9.6. Khi nào nên sử dụng iterrows():

- Khi DataFrame tương đối nhỏ.
- Khi logic xử lý cho mỗi hàng phức tạp và khó vector hóa.
- Khi tính dễ đọc của mã quan trọng hơn hiệu suất tối đa.
- Khi thực sự cần đối tượng Series cho mỗi hàng để sử dụng các phương thức của Series.

4.7.9.7. Các lựa chọn thay thế cho iterrows() (thường hiệu quả hơn):

- ***Vector hóa (Vectorization):*** Sử dụng các phép toán trực tiếp trên cột Series của Pandas. Đây là cách nhanh nhất. Ví dụ:

```
df['Tuổi mới'] = df['Tuổi'] + 1 # Ví dụ vector hóa
```

- ***DataFrame.apply()***: Áp dụng một hàm dọc theo một trục của DataFrame (thường là các hàng hoặc cột). Ví dụ:

```
def categorize_age(age):
    if age < 18:
        return "Trẻ em"
    elif age < 60:
        return "Người lớn"
    else:
        return "Người cao tuổi"
df['Nhóm tuổi'] = df['Tuổi'].apply(categorize_age)
```

- ***DataFrame.itertuples()***: Lặp qua các hàng dưới dạng các namedtuples. Thường nhanh hơn iterrows(). Ví dụ:

```
# index=True để bao gồm index, name để đặt tên cho tuple
for mytuple in df.itertuples(index=True, name='PandasRow'):
    print(f"Index: {row_tuple.Index}, Tên: {row_tuple.Tên},
                                        Tuổi: {row_tuple.Tuổi}")
```

- ***Chuyển đổi sang danh sách các dictionary hoặc list***: Sử dụng df.to_dict('records') hoặc df.values.tolist() rồi lặp qua chúng nếu phù hợp.

4.7.10. rank()

- Thực hiện sắp xếp trên cột tương ứng, sau đó trả về thứ hạng của các giá trị dựa trên kết quả sắp xếp (không xếp hạng cho các giá trị NaN).
- Cú pháp:

```
DataFrame.rank(axis=0, method='average', numeric_only=None,
                    na_option='keep', ascending=True, pct=False)
```

trong đó:

- `axis`: 0 hoặc 'index' cho hàng và 1 hoặc 'column' cho Cột.
- `method`: Lấy một chuỗi đầu vào ('average', 'min', 'max', 'first', 'dense') để cho pandas biết phải làm gì với cùng một giá trị. Mặc định là 'average', có nghĩa là gán mức xếp hạng trung bình cho các giá trị tương tự.
- `numeric_only`: Nhận giá trị boolean và hàm rank chỉ hoạt động trên giá trị không phải số nếu giá trị đó là False.
- `na_option`: nhận 1 trong 3 giá trị ('keep', 'top', 'bottom') để đặt vị trí của các giá trị Null nếu có trong series đã truyền.
- `ascending`: Giá trị Boolean xếp theo thứ tự tăng dần nếu được gán là True.
- `pct`: Giá trị Boolean xếp hạng phần trăm nếu được gán là True.
- Kiểu trả về: Chuỗi cho biết thứ hạng.

- Ví dụ: xếp hạng dựa trên giá trị của cột A

```
df['A_rank'] = df['A'].rank()
print(df)
```

	A	B	C	A_rank
0	1.0	2.0	3.0	1.0
1	4.0	NaN	NaN	2.0
2	NaN	NaN	NaN	NaN
3	5.0	6.0	7.0	3.0
4	8.0	NaN	9.0	4.0

4.7.11. rename()

4.7.11.1. Giới thiệu

DataFrame.rename được sử dụng để thay đổi tên của các nhãn trục (index labels hoặc column names) của một DataFrame. Đây là một thao tác rất phổ biến khi cần làm cho tên cột hoặc tên hàng trở nên dễ hiểu hơn, tuân theo một quy ước đặt tên nhất định, hoặc sửa lỗi chính tả.

4.7.11.2. Đặc điểm chính của DataFrame.rename

- **Đổi tên linh hoạt:** Có thể đổi tên một hoặc nhiều nhãn của index, cột, hoặc cả hai cùng lúc.
- **Sử dụng dictionary hoặc function:** có thể cung cấp một dictionary để ánh xạ tên cũ sang tên mới, hoặc một hàm để áp dụng logic đổi tên phức tạp hơn cho tất cả các nhãn.
- **Không thay đổi DataFrame gốc** (mặc định): Giống như nhiều phương thức khác trong Pandas, rename theo mặc định sẽ trả về một DataFrame mới đã được đổi tên, DataFrame gốc không bị ảnh hưởng. Có thể thay đổi hành vi này bằng tham số inplace=True.
- **Chỉ định trục:** cần chỉ rõ muốn đổi tên nhãn của index (axis=0 hoặc axis='index') hay nhãn của cột (axis=1 hoặc axis='columns').

4.7.11.3. Cú pháp

```
df.rename(mapper=None, index=None, columns=None, axis=None,
    copy=True, inplace=False, level=None, errors='ignore')
```

trong đó:

- `mapper`: Một đối tượng dạng dictionary hoặc một hàm. Dùng để cung cấp logic ánh xạ cho việc đổi tên.
  - Nếu là `dictionary`: {'tên_cũ_1': 'tên_mới_1', 'tên_cũ_2': 'tên_mới_2'}.
  - Nếu là `hàm`: Hàm này sẽ nhận tên nhãn hiện tại làm đối số và trả về tên nhãn mới.
- `index`: Dictionary hoặc hàm để đổi tên các nhãn của index. Tương đương với việc truyền mapper và đặt `axis='index'`.
- `columns`: Dictionary hoặc hàm để đổi tên các nhãn của cột. Tương đương với việc truyền mapper và đặt axis='columns'.
- `axis`: Xác định trục để áp dụng mapper.
  - `0` hoặc `'index'`: Đổi tên nhãn index (hàng).
  - `1` hoặc `'columns'`: Đổi tên nhãn cột.
- `copy`: (Không còn được sử dụng phổ biến, chủ yếu để tương thích ngược. `inplace` kiểm soát việc này).
- `inplace`: Kiểu boolean, mặc định là `False`.
  - Nếu `False` (mặc định): Trả về một bản sao của DataFrame với các nhãn đã được đổi tên. DataFrame gốc không thay đổi.
  - Nếu `True`: Thực hiện thao tác đổi tên trực tiếp trên DataFrame gốc và trả về None.
- `level`: Số nguyên hoặc tên của level nếu DataFrame có `MultiIndex`. Chỉ đổi tên các nhãn trong level được chỉ định.
- `errors`: Mặc định là `'ignore'`.
  - `'ignore'`: Nếu `mapper` là `dictionary` và chứa các khóa không tồn tại trong các nhãn của trục, chúng sẽ bị bỏ qua.
  - `'raise'`: Nếu `mapper` là `dictionary` và chứa các khóa không tồn tại, một lỗi `KeyError` sẽ được nêu ra.

4.7.11.4. Ví dụ

Cho một DataFrame như sau:

	Họ và Tên	NĂM SINH	điểm_tb
SV1	Nguyễn Văn Tý	1990	8.5
SV2	Trần Thị Sửu	1995	9.0
SV3	Lê Văn Dần	1998	7.5

```
import pandas as pd
data = {'Họ và Tên': ['Nguyễn Văn Tý', 'Trần Thị Sửu', 'Lê Văn Dần'],
        'NĂM SINH': [1990, 1995, 1988],
        'điểm_tb': [8.5, 9.0, 7.5]}
df = pd.DataFrame(data, index=['SV1', 'SV2', 'SV3'])
```

i. Đổi tên cột sử dụng dictionary

```
'''Đổi tên cột 'Họ và Tên' thành 'HoTen' và 'NĂM SINH' thành 'NamSinh'
'''
df_renamed_cols = df.rename(columns={'Họ và Tên': 'HoTen',
                                     'NĂM SINH': 'NamSinh'})
print("\nSau khi đổi tên cột (trả về DataFrame mới):")
print(df_renamed_cols)
print("\nDataFrame gốc không thay đổi:")
print(df)
```

ii. Đổi tên index sử dụng dictionary

```
df_renamed_index = df.rename(index={'hs1': 'HocSinh1',
```

```
                                                    'hs3': 'HocSinh3'})
print("\nSau khi đổi tên index:")
print(df_renamed_index)
```

iii. Đổi tên cột sử dụng hàm

```
''' Đổi tất cả tên cột thành chữ thường và thay thế khoảng trắng bằng dấu gạch dưới '''
def chuan_hoa_ten_cot(ten_cot):
    return ten_cot.lower().replace(' ', '_').replace('đ', 'd')

df_renamed_cols_func = df.rename(columns=chuan_hoa_ten_cot)
print("\nSau khi đổi tên cột bằng hàm:")
print(df_renamed_cols_func)
```

iv. Đổi tên index sử dụng hàm

```
''' Thêm tiền tố 'ID_' vào mỗi nhãn index '''
df_renamed_index_func = df.rename(index=lambda x: 'ID_' + x.upper())
print("\nSau khi đổi tên index bằng hàm:")
print(df_renamed_index_func)
```

v. Đổi tên và thay đổi trực tiếp DataFrame gốc (inplace=True)

```
''' Tạo bản sao để không ảnh hưởng df gốc của các ví dụ khác '''
df_copy = df.copy()
print("\nDataFrame copy trước khi inplace rename:")
print(df_copy)

df_copy.rename(columns={'điểm_tb': 'DiemTrungBinh'}, inplace=True)
print("\nSau khi đổi tên cột 'điểm_tb' (inplace=True) trên bản sao:")
print(df_copy)
# đến đây df_copy đã bị thay đổi
```

vi. Sử dụng tham số axis

Thay vì dùng `index=` hoặc `columns=`, có thể dùng `mapper=` kết hợp với `axis`.

```
# Đổi tên cột sử dụng mapper và axis
df_renamed_axis_cols = df.rename(mapper={'Họ và Tên': 'FullName'},
                                                        axis=1)
''' hoặc df_renamed_axis_cols = df.rename(mapper={'Họ và Tên': 'FullName'}, axis='columns') '''
print("\nĐổi tên cột dùng mapper và axis=1:")
print(df_renamed_axis_cols)

# Đổi tên index sử dụng mapper và axis
df_renamed_axis_index = df.rename(mapper={'hs1': 'Student1'}, axis=0)
''' hoặc df_renamed_axis_index = df.rename(mapper={'hs1': 'Student1'}, axis='index') '''
print("\nĐổi tên index dùng mapper và axis=0:")
print(df_renamed_axis_index)
```

vii. Xử lý lỗi với errors='raise'

```
try:
    df.rename(columns={'CotKhongTonTai': 'TenMoi'}, errors='raise')
except KeyError as e:
    print(f"\nLỗi khi đổi tên cột không tồn tại (errors='raise'): {e}")

# Mặc định errors='ignore' nên không có lỗi nếu cột không tồn tại
df_ignore_error = df.rename(columns={'CotKhongTonTai': 'TenMoi'})
print("\nKhi đổi tên cột không tồn tại (errors='ignore' - mặc định):")
print(df_ignore_error) # DataFrame không thay đổi gì ở cột này
```

4.7.12. sample()

4.7.12.1. Giới thiệu

- DataFrame.sample trong Pandas là một phương thức tiện dụng cho phép lấy một số lượng mẫu ngẫu nhiên các mục (hàng hoặc cột) từ một DataFrame.
- Đây là một công cụ hữu ích trong nhiều tình huống, chẳng hạn như:
  - **Trong phân tích thăm dò dữ liệu (Exploratory Data Analysis - EDA):** Khi làm việc với tập dữ liệu lớn, việc lấy mẫu ngẫu nhiên giúp nhanh chóng có cái nhìn tổng quan về dữ liệu mà không cần xử lý toàn bộ.
  - **Trong Học máy (Machine Learning):** Tạo tập huấn luyện (train set) và tập kiểm tra (test set) ngẫu nhiên từ dữ liệu gốc.
  - **Trong Kiểm thử (Testing):** Lấy mẫu để kiểm tra một phần chức năng hoặc thuật toán trên một tập dữ liệu con.
  - **Trong Mô phỏng (Simulation):** Sử dụng mẫu để chạy các kịch bản mô phỏng.
  - Để giảm kích thước dữ liệu cho việc thử nghiệm hoặc phát triển nhanh.
  - Khi cần đảm bảo tính ngẫu nhiên có thể tái lập được (qua random_state).

4.7.12.2. Đặc điểm chính của DataFrame.sample

- **Lấy mẫu ngẫu nhiên:** Các mục được chọn một cách ngẫu nhiên.
- **Lấy mẫu hàng hoặc cột:** có thể chỉ định lấy mẫu theo hàng (mặc định) hoặc theo cột.
- **Số lượng hoặc tỷ lệ:** có thể chỉ định số lượng mục cụ thể cần lấy hoặc một tỷ lệ phần trăm của tổng số mục.
- **Lấy mẫu có lặp lại hoặc không lặp lại:** Cho phép chọn cùng một mục nhiều lần (có lặp lại - *with replacement*) hoặc mỗi mục chỉ được chọn một lần (không lặp lại - *without replacement*).
- **Tính tái lập** (*Reproducibility*): Có thể sử dụng tham số `random_state` để đảm bảo rằng việc lấy mẫu ngẫu nhiên sẽ cho ra cùng một kết quả mỗi khi chạy lại mã, rất quan trọng cho việc gỡ lỗi và chia sẻ kết quả.

4.7.12.3. Cú pháp

```
df.sample(n=None, frac=None, replace=False, weights=None,
    random_state=None, axis=None, ignore_index=False)
```

trong đó:

- `n`: Số lượng mục cần lấy mẫu. Không thể sử dụng cùng lúc với `frac`.
- `frac`: Tỷ lệ phần trăm (từ 0 đến 1) của các mục cần lấy mẫu. Ví dụ: 0.5 nghĩa là lấy 50% số mục. Không thể sử dụng cùng lúc với `n`. Khi frac=1, các hàng dữ liệu trong dataframe sẽ được xáo trộn thứ tự.
- `replace`: Kiểu boolean, mặc định là `False`.
  - `False`: Lấy mẫu không lặp lại (mỗi mục chỉ được chọn một lần). Nếu `n` lớn hơn số lượng mục có sẵn (hoặc `frac > 1`), sẽ gây lỗi trừ khi `replace=True`.
  - `True`: Lấy mẫu có lặp lại (một mục có thể được chọn nhiều lần).
- `weights`: Một đối tượng Series, mảng numpy, hoặc tên cột trong DataFrame. Xác định trọng số (xác suất) để chọn các mục. Các mục có trọng số cao hơn sẽ có khả năng được chọn cao hơn. Nếu không được cung cấp, tất cả các mục có cơ hội được chọn như nhau.

- *random_state*: Số nguyên hoặc một đối tượng *numpy.random.RandomState*. Nếu là số nguyên, nó sẽ được sử dụng làm hạt giống (seed) cho bộ tạo số ngẫu nhiên. Điều này đảm bảo rằng việc lấy mẫu sẽ cho ra kết quả giống nhau mỗi khi mã lệnh được chạy với cùng một *random_state*. Rất hữu ích cho việc tái lập kết quả.
- *axis*: Trục để lấy mẫu.
  - *0* hoặc *'index'*: Lấy mẫu các hàng (mặc định).
  - *1* hoặc *'columns'*: Lấy mẫu các cột.
- *ignore_index*: Kiểu boolean, mặc định là False.
  - Nếu *True*: Index của DataFrame kết quả sẽ được đặt lại thành *RangeIndex (0, 1, 2, ...)*.
  - Nếu *False*: Index của DataFrame kết quả sẽ giữ nguyên từ DataFrame gốc.

4.7.12.4. Ví dụ

Cho một DataFrame như sau:

Mã lệnh

```
import numpy as np
data = {'col_A': range(10),
        'col_B': np.random.rand(10),
        'col_C': ['A','B','C','D',
                  'E','F','G','H','I','J']}
df = pd.DataFrame(data)
print("DataFrame gốc:")
print(df)
```

Kết quả

	col_A	col_B	col_C
0	0	0.996832	A
1	1	0.631072	B
2	2	0.216001	C
3	3	0.891800	D
4	4	0.819657	E
5	5	0.556562	F
6	6	0.735328	G
7	7	0.480252	H
8	8	0.518690	I
9	9	0.899518	J

i. Lấy mẫu một số lượng hàng cố định (n)

```
# Lấy 3 hàng ngẫu nhiên, với random_state để kết quả lặp lại
sample_n_rows = df.sample(n=3, random_state=42)
print("\nLấy 3 hàng ngẫu nhiên (random_state=42):")
print(sample_n_rows)
```

ii. Lấy mẫu một tỷ lệ phần trăm hàng (frac)

```
# Lấy 50% số hàng ngẫu nhiên
sample_frac_rows = df.sample(frac=0.5, random_state=1)
print("\nLấy 50% số hàng ngẫu nhiên (random_state=1):")
print(sample_frac_rows)
```

iii. Lấy mẫu có lặp lại (replace=True)

```
# Lấy 5 hàng ngẫu nhiên, có thể có hàng được chọn nhiều lần
sample_replace_rows = df.sample(n=5, replace=True, random_state=10)
print("\nLấy 5 hàng ngẫu nhiên có lặp lại (random_state=10):")
print(sample_replace_rows)
```

iv. Lấy mẫu cột (axis=1)

```
# Lấy 2 cột ngẫu nhiên
```

```
sample_columns = df.sample(n=2, axis=1, random_state=7)
print("\nLấy 2 cột ngẫu nhiên (random_state=7):")
print(sample_columns)
```

v. Lấy mẫu với trọng số (weights)

Giả sử muốn các hàng có giá trị tại col_A lớn hơn có khả năng được chọn cao hơn.

```
# Tạo trọng số: các hàng có col_A lớn hơn sẽ có trọng số cao hơn
# (Ví dụ đơn giản, có thể phức tạp hơn)
weights_for_rows = df['col_A'] + 1 # +1 để tránh trọng số 0
sample_weighted_rows = df.sample(n=3, weights=weights_for_rows,
                                                    random_state=21)
print("\nLấy 3 hàng ngẫu nhiên với trọng số (dựa trên col_A,
                                                  random_state=21):")
print(sample_weighted_rows)
```

vi. Reset index của DataFrame kết quả (ignore_index=True)

```
sample_ignore_idx = df.sample(n=3, random_state=5, ignore_index=True)
print("\nLấy 3 hàng ngẫu nhiên và reset index (random_state=5):")
print(sample_ignore_idx)
```

4.8. Duyệt DataFrame theo hàng

4.8.1. Lưu ý về hiệu suất và mục đích của việc duyệt DataFrame theo hàng

- Nói chung, nên cố gắng tránh duyệt qua các hàng trong Pandas. Pandas được thiết kế để thực hiện các phép toán vector hóa (*vectorized operations*), tức là áp dụng một thao tác lên toàn bộ cột hoặc `DataFrame` cùng một lúc. Các phép toán vector hóa thường nhanh hơn rất nhiều so với việc lặp qua từng hàng.
- Khi nào nên duyệt hàng trong `DataFrame`:
  - Khi cần thực hiện các logic phức tạp trên mỗi hàng mà khó biểu diễn bằng các phép toán vector hóa.
  - Khi cần truyền dữ liệu từng hàng vào một hàm bên ngoài không được thiết kế để làm việc với `Series`/`DataFrame` của Pandas.
- Nếu phải duyệt hàng, `df.itertuples()` thường là lựa chọn tốt nhất về mặt hiệu suất.

4.8.2. Các cách duyệt DataFrame theo hàng

4.8.2.1. Cách 1: Sử dụng DataFrame.iterrows()

- Phương thức này trả về một `iterator`, mỗi lần lặp sẽ cung cấp một cặp (`index`, `Series`), trong đó index là nhãn chỉ mục của hàng và `Series` là dữ liệu của hàng đó.
- Ưu điểm: Dễ hiểu, cho phép truy cập cả nhãn chỉ mục và dữ liệu hàng dưới dạng `Series`.
- Nhược điểm:
  - Không hiệu quả về hiệu suất đối với các `DataFrame` lớn vì cần phải tạo một đối tượng `Series` cho mỗi hàng.
  - Kiểu dữ liệu (`dtype`) có thể không được giữ nguyên qua các hàng nếu `DataFrame` có các kiểu dữ liệu hỗn hợp trong các cột khác nhau. `Series` của hàng thường sẽ có dtype là `object`.

- Không nên sửa đổi `DataFrame` trong khi đang duyệt bằng `iterrows()`.

- Ví dụ

```
import pandas as pd
# Tạo DataFrame mẫu
data = {'Tên': ['An', 'Bình', 'Cúc'],
        'Tuổi': [22, 25, 21],
        'Thành phố': ['Hà Nội', 'TP.HCM', 'Đà Nẵng']}
df = pd.DataFrame(data, index=['ID1', 'ID2', 'ID3'])
print("DataFrame gốc:")
print(df)

for rowIndex, rowData in df.iterrows():
    print(f"Đang ở hàng có chỉ mục (index): { rowIndex }")
    print("Dữ liệu hàng (dưới dạng Series):")
    print(rowData)
    print(f"Tên: {rowData['Tên']}, Tuổi: {rowData['Tuổi']},
                                    Thành phố: {rowData['Thành phố']}")
    print("---")
```

kết quả:

```
DataFrame gốc:
      Tên  Tuổi Thành phố
ID1    An    22    Hà Nội
ID2  Bình    25    TP.HCM
ID3   Cúc    21   Đà Nẵng

Đang ở hàng có chỉ mục (index): ID1
Dữ liệu hàng (dưới dạng Series):
Tên              An
Tuổi             22
Thành phố    Hà Nội
Name: ID1, dtype: object
Tên: An, Tuổi: 22, Thành phố: Hà Nội
---
Đang ở hàng có chỉ mục (index): ID2
Dữ liệu hàng (dưới dạng Series):
Tên            Bình
Tuổi             25
Thành phố    TP.HCM
Name: ID2, dtype: object
Tên: Bình, Tuổi: 25, Thành phố: TP.HCM
---
Đang ở hàng có chỉ mục (index): ID3
Dữ liệu hàng (dưới dạng Series):
Tên              Cúc
Tuổi              21
Thành phố    Đà Nẵng
Name: ID3, dtype: object
Tên: Cúc, Tuổi: 21, Thành phố: Đà Nẵng
---
```

4.8.3. Cách 2: Sử dụng DataFrame.itertuples()

- Phương thức này trả về một iterator, mỗi lần lặp sẽ cung cấp một `namedtuple` (hoặc `tuple` thông thường nếu `name=None`) cho mỗi hàng. `namedtuple` này có các field tương ứng với tên cột, và phần tử đầu tiên thường là chỉ mục của hàng (nếu `index=True`, là mặc định).
- Ưu điểm: Hiệu quả hơn nhiều so với `iterrows()` vì nó không tạo đối tượng `Series` cho mỗi hàng. Kiểu dữ liệu thường được giữ nguyên tốt hơn.
- Nhược điểm: Việc truy cập dữ liệu thông qua thuộc tính của `namedtuple` hoặc chỉ số của `tuple`. Không nên sửa đổi `DataFrame` trong khi đang duyệt.

- Ví dụ

```
import pandas as pd
# Tạo DataFrame mẫu
data = {'Tên': ['An', 'Bình', 'Cúc'],
        'Tuổi': [22, 25, 21],
        'Thànhphố': ['Hà Nội', 'TP.HCM', 'Đà Nẵng']}
df = pd.DataFrame(data, index=['ID1', 'ID2', 'ID3'])
print("DataFrame gốc:")
print(df)
print("-----")
''' - df.itertuples trả về 1 tuple và sử dụng
        + index=True (mặc định): bao gồm cả chỉ mục hàng là phần tử
đầu tiên của tuple
        + name: có thể đặt tên hoặc gán bằng None
     - Khi tên cột có khoảng trắng hoặc ký tự đặc biệt, cần dùng
getattr(hang, 'Tên Cột')
'''
for hang in df.itertuples(index=True, name='MyTuple'):
    print(f"Dữ liệu của biến hang là: {tuple(hang)}")
    print(f"Chỉ mục hàng: {hang.Index}") # Truy cập chỉ mục qua thuộc
tính 'Index'
    print(f"Tên:   {hang.Tên},   Tuổi:   {hang.Tuổi},   Thành   phố:
{getattr(hang, 'Thànhphố')}")
    print("-----")
```

kết quả:

```
DataFrame gốc:
      Tên  Tuổi Thànhphố
ID1    An    22   Hà Nội
ID2  Bình    25   TP.HCM
ID3   Cúc    21  Đà Nẵng
-----
An
Dữ liệu của biến hang là: ('ID1', 'An', 22, 'Hà Nội')
Chỉ mục hàng: ID1
An: An, Tuổi: 22, Thành phố: Hà Nội
-----
Dữ liệu của biến hang là: ('ID2', 'Bình', 25, 'TP.HCM')
Chỉ mục hàng: ID2
An: Bình, Tuổi: 25, Thành phố: TP.HCM
-----
Dữ liệu của biến hang là: ('ID3', 'Cúc', 21, 'Đà Nẵng')
Chỉ mục hàng: ID3
An: Cúc, Tuổi: 21, Thành phố: Đà Nẵng
-----
```

4.8.3.1.1. Cách 3: Lặp qua chỉ mục (index) và sử dụng df.loc[] hoặc df.iloc[]

- Có thể lặp qua các nhãn chỉ mục của `DataFrame` (sử dụng `df.index`) hoặc qua một dãy số (sử dụng `range(len(df))`) rồi dùng `.loc` hoặc `.iloc` để truy cập từng hàng.
- Ưu điểm: Trực quan, cho phép truy cập hàng dưới dạng `Series`.
- Nhược điểm: Tương tự như `iterrows()`, việc tạo Series cho mỗi hàng có thể không hiệu quả với `DataFrame` lớn.
- Ví dụ

```
import pandas as pd
# Tạo DataFrame mẫu
data = {'Tên': ['An', 'Bình', 'Cúc'],
        'Tuổi': [22, 25, 21],
        'Thànhphố': ['Hà Nội', 'TP.HCM', 'Đà Nẵng']}
df = pd.DataFrame(data, index=['ID1', 'ID2', 'ID3'])
print(df)
```

```
print("-----")
print("Cách 3A: Lặp qua df.index và dùng df.loc[]")
for chi_muc in df.index:
    # Lấy hàng dưới dạng Series bằng nhãn chỉ mục
    Series_hang = df.loc[chi_muc]
    print(f"Đang ở hàng có chỉ mục: {chi_muc}")
    print(f"Tên: {Series_hang['Tên']}, Tuổi: {Series_hang['Tuổi']},
                                Thành phố: {Series_hang['Thànhphố']}")
    print("-----")


print("\nCách 3B (biến thể): Lặp qua range(len(df)) và dùng
df.iloc[]")
for i in range(len(df)):
    # Lấy hàng dưới dạng Series bằng vị trí số nguyên
    Series_hang = df.iloc[i]
    print(f"Đang ở hàng có vị trí số nguyên: {i}")
    print(f"Tên: {Series_hang['Tên']}, Tuổi: {Series_hang['Tuổi']},
                                Thành phố: {Series_hang['Thànhphố']}")
    print("-----")
```

kết quả:

```
       Tên  Tuổi Thànhphố
ID1     An    22    Hà Nội
ID2   Bình    25    TP.HCM
ID3    Cúc    21   Đà Nẵng
-----
Cách 3A: Lặp qua df.index và dùng df.loc[]
Đang ở hàng có chỉ mục: ID1
Tên: An, Tuổi: 22, Thành phố: Hà Nội
-----
Đang ở hàng có chỉ mục: ID2
Tên: Bình, Tuổi: 25, Thành phố: TP.HCM
-----
Đang ở hàng có chỉ mục: ID3
Tên: Cúc, Tuổi: 21, Thành phố: Đà Nẵng
-----

Cách 3B (biến thể): Lặp qua range(len(df)) và dùng df.iloc[]
Đang ở hàng có vị trí số nguyên: 0
Tên: An, Tuổi: 22, Thành phố: Hà Nội
-----
Đang ở hàng có vị trí số nguyên: 1
Tên: Bình, Tuổi: 25, Thành phố: TP.HCM
-----
Đang ở hàng có vị trí số nguyên: 2
Tên: Cúc, Tuổi: 21, Thành phố: Đà Nẵng
-----
```

4.9. Duyệt DataFrame theo cột

4.9.1. Cách 1: Duyệt trực tiếp qua df.columns

(Đây là cách đơn giản, rõ ràng và được dùng phổ biến nhất)

- Thuộc tính *df.columns* trả về một đối tượng *Index* chứa tất cả các tên cột. Có thể duyệt qua đối tượng này như duyệt qua một danh sách.
- Ví dụ

```
import pandas as pd
data = {'Tên': ['An', 'Bình', 'Cúc'],
        'Tuổi': [22, 25, 21],
        'Thành phố': ['Hà Nội', 'TP.HCM', 'Đà Nẵng']}
df = pd.DataFrame(data, index=['ID1', 'ID2', 'ID3'])
print(df)
print('Tên các cột có trong DataFrame:')
for ten_cot in df.columns:
```

```
    print(ten_cot)
```

kết quả:

```
        Tên  Tuổi Thành phố
ID1     An    22    Hà Nội
ID2   Bình    25    TP.HCM
ID3    Cúc    21   Đà Nẵng
Tên các cột có trong DataFrame:
Tên
Tuổi
Thành phố
```

4.9.2. Cách 2: Sử dụng vòng lặp for với range và chỉ số

- Cũng có thể sử dụng một vòng lặp *for* truyền thống với chỉ số để truy cập tên cột.
- Ví dụ

```
import pandas as pd
data = {'Tên': ['An', 'Bình', 'Cúc'],
        'Tuổi': [22, 25, 21],
        'Thành phố': ['Hà Nội', 'TP.HCM', 'Đà Nẵng']}
df = pd.DataFrame(data, index=['ID1', 'ID2', 'ID3'])
print(df)
print('Tên các cột có trong DataFrame:')
for i in range(len(df.columns)):
    print(df.columns[i])
```

kết quả:

```
        Tên  Tuổi Thành phố
ID1     An    22    Hà Nội
ID2   Bình    25    TP.HCM
ID3    Cúc    21   Đà Nẵng
Tên các cột có trong DataFrame:
Tên
Tuổi
Thành phố
```

4.9.3. Cách 3: Chuyển tên cột thành danh sách rồi duyệt

- Có thể chuyển *df.columns* thành một danh sách Python rồi duyệt qua danh sách đó.
- Ví dụ

```
import pandas as pd
data = {'Tên': ['An', 'Bình', 'Cúc'],
        'Tuổi': [22, 25, 21],
        'Thành phố': ['Hà Nội', 'TP.HCM', 'Đà Nẵng']}
df = pd.DataFrame(data, index=['ID1', 'ID2', 'ID3'])
print(df)
print('Tên các cột có trong DataFrame:')
danh_sach_ten_cot = df.columns.tolist()
for ten_cot in danh_sach_ten_cot:
    print(ten_cot)
```

4.9.4. Cách 4: Sử dụng df.items() (Duyệt qua (tên cột, Series dữ liệu cột))

- Nếu muốn duyệt qua cả tên cột và dữ liệu của cột đó (dưới dạng một *Series*), có thể dùng *df.items()*.
- Ví dụ

```
import pandas as pd
data = {'Tên': ['An', 'Bình', 'Cúc'],
        'Tuổi': [22, 25, 21],
        'Thành phố': ['Hà Nội', 'TP.HCM', 'Đà Nẵng']}
df = pd.DataFrame(data, index=['ID1', 'ID2', 'ID3'])
print(df)
print('Các cột có trong DataFrame:')
for ten_cot, du_lieu_cot in df.items():
    print(f"Tên cột: {ten_cot}")
```

```
print("Dữ liệu của cột:")
print(du_lieu_cot)
print("-----")
```

---

5. MỘT SỐ THƯ VIỆN KHÁC TRONG PYTHON

5.1. dateutil

5.1.1. Hàm dateutil.parser.parse

- Được sử dụng để phân tích và chuyển đổi một chuỗi (`string`) biểu diễn thời gian không chuẩn hóa (hoặc có nhiều định dạng khác nhau) thành đối tượng `datetime.datetime`.
- Cú pháp:
  `dateutil.parser.parse(timestr, default=None, ignoretz=False, tzinfos=None, **kwargs)`
  trong đó
  - **`timestr`**: Chuỗi biểu diễn thời gian cần phân tích.
  - **`default`**: Một đối tượng datetime được sử dụng làm giá trị mặc định nếu chuỗi không cung cấp đầy đủ thông tin.
  - **`ignoretz`**: Nếu True, bỏ qua thông tin múi giờ trong chuỗi.
  - **`tzinfos`**: Từ điển hoặc hàm để xử lý thông tin múi giờ tùy chỉnh trong chuỗi.
- Ví dụ

Mã lệnh	*Kết quả*

• *Phân tích chuỗi thời gian đơn giản:*
`from dateutil.parser import parse`
`# Chuỗi thời gian cơ bản`
`dt = parse("2023-03-29")`
`print(dt)`
	`2023-03-29 00:00:00`

• *Phân tích chuỗi thời gian có giờ phút:*
`dt = parse("29 March 2023 14:30:00")`
`print(dt)`
	`2023-03-29 14:30:00`

• *Xử lý thời gian không chuẩn hóa:*
`dt = parse("March 29th, 2023 at 2:30 PM")`
`print(dt)`
	`2023-03-29 14:30:00`

• *Chuỗi thời gian có múi giờ:*
`dt = parse("2023-03-29T14:30:00+09:00")`
`print(dt)`
	`2023-03-29 14:30:00+09:00`

• *Bỏ qua múi giờ bằng ignoretz:*
`dt = parse("2023-03-29T14:30:00+09:00", ignoretz=True)`
`print(dt)`
	`2023-03-29 14:30:00`

• *Sử dụng giá trị mặc định khi chuỗi không đầy đủ:*
`from datetime import datetime`
`# Sử dụng ngày mặc định`
`default_date = datetime(2023, 3, 1)`
`dt = parse("14:30", default=default_date)`
`print(dt)`
	`2023-03-01 14:30:00`

LINK THAM KHẢO

[1].-Pandas Data Series: Exercises, Practice, Solution (w3 resource)

[2].-Pandas Standard Library Functions | Programiz

[3].-General functions — pandas 2.2.3 documentation

[4].-Pandas cheat sheet: Top 35 commands and operations

[5].-https://www.w3resource.com/python-exercises/pandas/index-dataframe.php