Numpy and Pandas

Welcome!

What Are Python Libraries?

- Python libraries are:
- **Reusable code modules** that provide specific functionality
- **Packaged for easy distribution** (typically via PyPI - Python Package Index)
- **Installed using pip** (Python's package manager)
- **Organized by purpose** (data science, web development, gaming, etc.)

Why Use Libraries?

1. **Save development time** - Don't reinvent the wheel
2. **Reliability** - Well-tested code
3. **Performance** - Many are optimized (like NumPy)
4. **Community support** - Widely used libraries have good documentation
5. **Specialized functionality** - Access to domain-specific tools

Standard Library

- math – os – datetime - json - re
- Examples:

```
import webbrowser
webbrowser.open(f"https://en.wikipedia.org/wiki/Special:Random")
```

Third-Party Libraries

- PyQt, Kivy, NumPy, pandas, scikit-learn, Django, Flask, …
- Example:

```
import qrcode
qrcode.make("Hello, Python!").show()
```

Using pip

```
pip install package_name         # Install the latest version
pip install package_name==1.2.3 # Install a specific version
pip install -U package_name      # Upgrade a package
pip list                         # List all installed packages
pip uninstall package_name       # Uninstall a package
```

What is a Virtual Environment?

- A virtual environment is an isolated Python environment that:
- Has its own Python interpreter
- Maintains its own set of installed packages
- Is separate from your system-wide Python installation
- Is separate from other projects' environments

Why Use venv?

- Dependency Isolation: Different projects can require different versions of the same package
- Clean Workspace: Avoids polluting your system Python installation
- Reproducibility: Makes it easier to share projects with others
- No Admin Rights Needed: Install packages without system permissions

How to use?

- Create: python -m venv my_project_env
- Start:

my_project_env\Scripts\activate

source my_project_env/bin/activate

End:

- deactivate

Numpy! What is it?

Why we should use numpy?

- NumPy methods are simple, so we could write them ourselves. But why should we use NumPy?

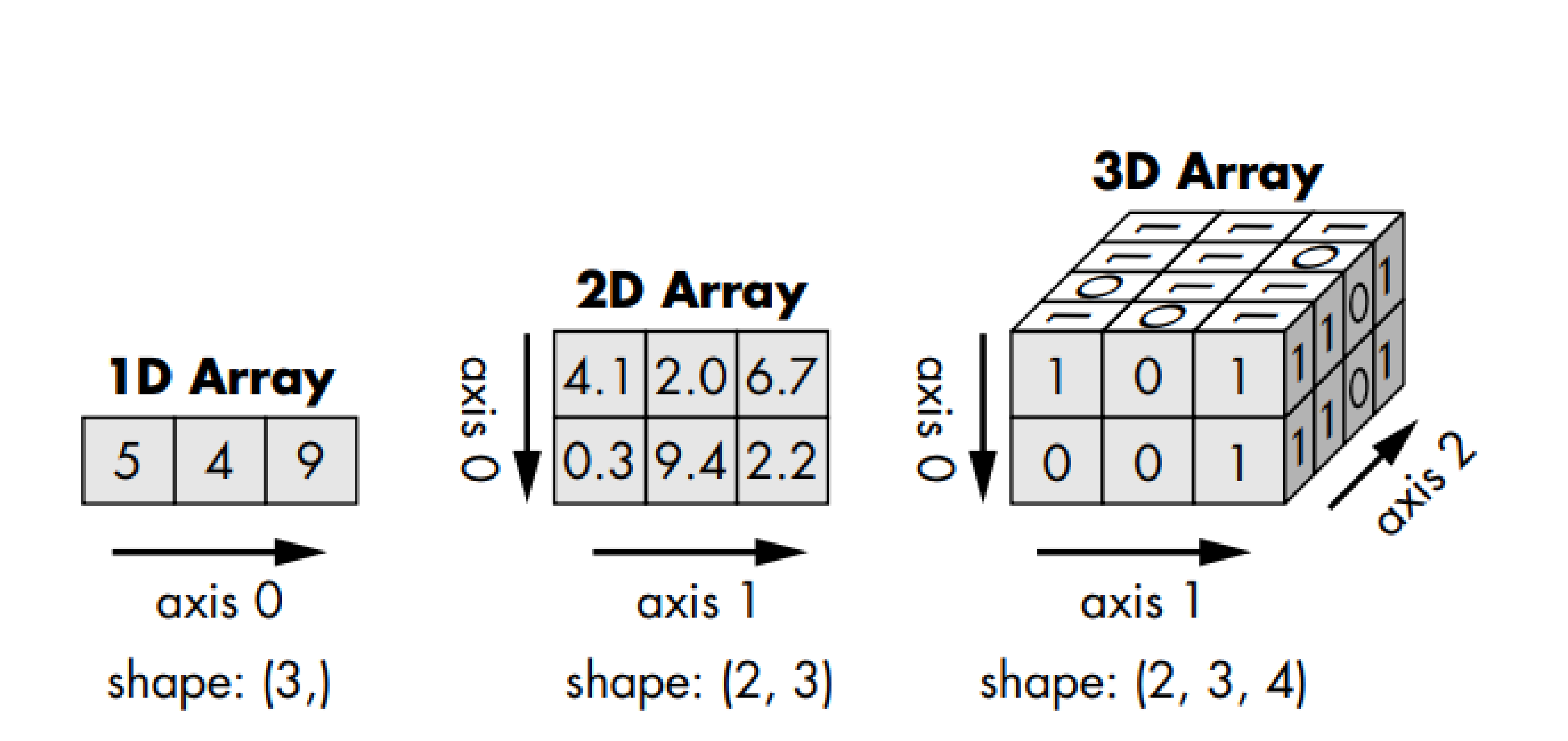
1. Coding with NumPy is easier and shorter.
2. NumPy is much faster!

So let's race them!

Arrays: Key Differences from Python Lists

- **Homogeneous data types**: All elements must be same type (unlike Python lists)
- **Fixed size at creation**: Size can't change (unlike lists which are dynamic)
- **Vectorized operations**: Fast operations on entire arrays without Python loops
- **Memory efficiency**: More compact storage than Python lists
- **Built-in math functions**: Optimized linear algebra, Fourier transforms, random number generation

Creating Arrays



Array Attributes

- **Shape**: Tuple indicating size in each dimension
- **ndim**: Number of dimensions
- **size**: Total number of elements
- **dtype**: Data type of elements

Dtype:

- np.int8, np.int16, np.int32, np.int64
- np.uint8, np.uint16, np.uint32, np.uint64
- np.float16, np.float32, np.float64
- np.complex64, np.complex128
- np.bool_
- np.object_ (Python objects)
- np.string_, np.unicode_

Creating Array: 1 Arrays with Initial Values

```python
import numpy as np

#Creating array:

# 1-D array
arr1d = np.array([1, 2, 3, 4, 5])

# 2-D array (matrix)
arr2d = np.array([[1, 2, 3], [4, 5, 6]])

# 3-D array
arr3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
```

Creating Array: 2 Arrays with Initial Values

```
import numpy as np

#Creating array:

# Array of zeros
zeros = np.zeros((3, 4))  # 3x4 array filled with 0.0

# Array of ones
ones = np.ones((2, 3, 4), dtype=np.int16)  # 2x3x4 array of 1s

# Empty array (contains garbage values)
empty = np.empty((2, 3))  # Uninitialized array (fast creation)
```

Creating Array: 3 Range-like Arrays

```
import numpy as np

#Creating array:

# Similar to range() but returns array
arange = np.arange(10)  # 0 to 9
arange_step = np.arange(0, 10, 2)  # 0, 2, 4, 6, 8

# With floating-point steps
arange_float = np.arange(0, 1, 0.1)  # 0.0, 0.1, 0.2,...0.9

# Evenly spaced numbers over interval (inclusive)
linspace = np.linspace(0, 1, 5)  # [0.0, 0.25, 0.5, 0.75, 1.0]
```

Creating Array: 4 Random Arrays

```python
import numpy as np

# Set seed for reproducibility
np.random.seed(1404)

# Standard uniform [0, 1)
uniform = np.random.rand(2, 3)  # 2x3 array
# With custom range [low, high)
uniform_range = np.random.uniform(low=5, high=10, size=(3, 2))

# Standard normal (mean=0, std=1)
normal = np.random.randn(4)  # Shortcut
# Custom parameters (mean, std)
normal_custom = np.random.normal(loc=100, scale=15, size=100)

# Uniform integers [low, high)
integers = np.random.randint(low=0, high=10, size=5)
```

```python
# Binomial distribution
binomial = np.random.binomial(n=10, p=0.5, size=100)

# Poisson distribution
poisson = np.random.poisson(lam=5, size=100)

# Exponential distribution
exponential = np.random.exponential(scale=1.0, size=100)

# Beta distribution
beta = np.random.beta(a=0.5, b=0.5, size=100)

# Gamma distribution
gamma = np.random.gamma(shape=2, scale=2, size=100)
```

```python
# Random choice from array
choices = np.random.choice(['a', 'b', 'c'], size=10, p=[0.1, 0.3, 0.6])

# Shuffle array in-place
arr = np.arange(10)
np.random.shuffle(arr)

# Permutation (returns shuffled copy)
permuted = np.random.permutation(arr)
```

Accessing and Changing NumPy Arrays

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr[0])   # First element → 1
print(arr[-1])  # Last element → 5

arr[0] = 10     # Change first element to 10
arr[2:4] = 30   # Change elements at index 2 and 3 to 30

arr2d = np.array([[1, 2, 3], [4, 5, 6]])

print(arr2d[0, 1])   # First row, second column → 2
print(arr2d[:, 1])  # All rows, second column → array([2, 5])

arr2d[1, 2] = 60     # Change element at row 1, column 2
arr2d[:, 0] = 0      # Change first column of all rows to 0
```

ReShape

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6])
reshaped0 = arr.reshape(2, 3)  # 2 rows, 3 columns
# Result:
# array([[1, 2, 3],
#        [4, 5, 6]])


arr = np.arange(8)  # array([0, 1, 2, 3, 4, 5, 6, 7])
reshaped1 = arr.reshape(2, 2, 2)
# Result:
# array([[[0, 1],
#         [2, 3]],
#        [[4, 5],
#         [6, 7]]])

# Automatic dimension calculation (-1)
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
reshaped2 = arr.reshape(2, -1)  # -1 means "calculate automatically"
# Result is 2×4 array

copy = arr.copy()
```

append, insert, and delete !!!

```python
import numpy as np

arr = np.array([1, 2, 3])
new_arr = np.append(arr, 4)  # Append 4 at the end
print(new_arr)  # Output: [1 2 3 4]
#even in 2D!
arr = np.array([1, 2, 3])
new_arr = np.insert(arr, 1, 99)  # Insert 99 at index 1
print(new_arr)  # Output: [1 99 2 3]
#even in 2D!
arr = np.array([1, 2, 3, 4, 5])
new_arr = np.delete(arr, 1)  # Remove element at index 1
print(new_arr)  # Output: [1 3 4 5]
#even in 2D!
```

Arithmetic Operations

```
import numpy as np

a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Addition
print(a + b)  # [5 7 9]
print(np.add(a, b))  # equivalent

# Subtraction
print(b - a)  # [3 3 3]
print(np.subtract(b, a))

# Multiplication (element-wise, not matrix multiplication)
print(a * b)  # [4 10 18]
print(np.multiply(a, b))

# Division
print(b / a)  # [4.  2.5 2. ]
print(np.divide(b, a))

# Exponentiation
print(a ** 2)  # [1 4 9]
print(np.power(a, 2))

# Modulus/remainder
print(b % a)  # [0 1 0]
print(np.mod(b, a))
```

Matrix Operations

```
x = np.array([[1, 2], [3, 4]])
y = np.array([[5, 6], [7, 8]])

# Matrix multiplication
print(x @ y)  # [[19 22] [43 50]]
print(np.matmul(x, y))  # equivalent
print(x.dot(y))  # alternative

a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

print(np.inner(a, b))  # 32
print(np.outer(a, b))
'''
    [[ 4  5  6]
     [ 8 10 12]
     [12 15 18]]    '''

v1 = [1, 0, 0]
v2 = [0, 1, 0]
print(np.cross(v1, v2))  # [0 0 1]
```

Aggregation Operations

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

# Sum all elements
print(np.sum(arr))  # 21

# Sum along axis 0 (columns)
print(np.sum(arr, axis=0))  # [5 7 9]

# Sum along axis 1 (rows)
print(np.sum(arr, axis=1))  # [6 15]

print(np.mean(arr))  # 3.5
print(np.min(arr))  # 1
print(np.max(arr))  # 6
print(np.std(arr))  # standard deviation
print(np.var(arr))  # variance
print(np.prod(arr))  # product of all elements (720)
print(np.cumsum(arr))  # cumulative sum [1 3 6 10 15 21]
```

Comparison Operations

```
a = np.array([1, 2, 3])
b = np.array([2, 2, 2])

# Element-wise comparison
print(a == b)  # [False  True False]
print(a > b)   # [False False  True]

# Array-wise comparison
print(np.array_equal(a, b))  # False

# Any/All
print(np.any(a > 2))  # True
print(np.all(a < 4))  # True
```

Mathematical Functions

```
angles = np.array([0, np.pi/2, np.pi])

# Trigonometric functions
print(np.sin(angles))  # [0.0000000e+00 1.0000000e+00 1.2246468e-16]
print(np.cos(angles))
print(np.tan(angles))

# Rounding
arr = np.array([1.234, 2.567, 3.901])
print(np.round(arr, 1))  # [1.2 2.6 3.9]

# Floor and ceiling
print(np.floor(arr))  # [1. 2. 3.]
print(np.ceil(arr))   # [2. 3. 4.]

# Logarithms
print(np.log(arr))    # Natural log
print(np.log10(arr))  # Base-10 log
```

Linear Algebra Operations

```python
from numpy import linalg

m = np.array([[1, 2], [3, 4]])

# Determinant
print(linalg.det(m))  # -2.0000000000000004

# Inverse
print(linalg.inv(m))  # [[-2.   1. ] [ 1.5 -0.5]]

# Eigenvalues and eigenvectors
eigenvalues, eigenvectors = linalg.eig(m)
```

Copy?

```python
import numpy as np

a = np.array([1, 2, 3, 4, 5])
b = a
b[0] = 99

# What will this print?
print("Array a:", a)
```

It is Not just this!

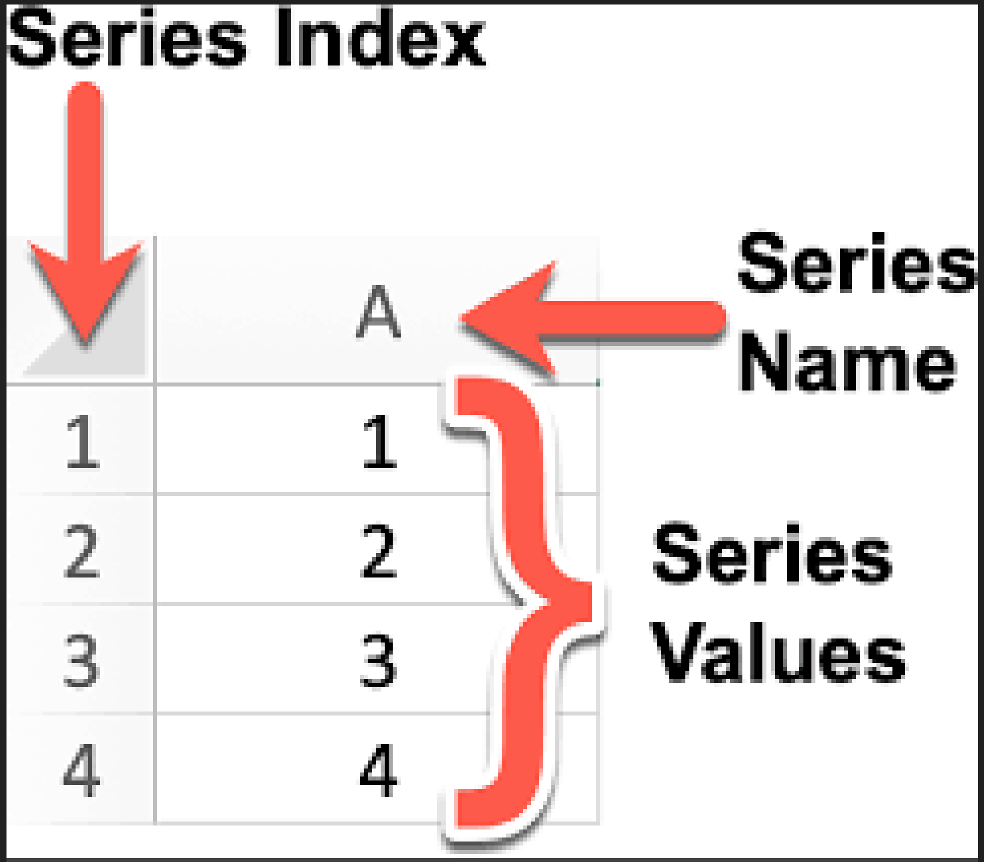
- numpy.fft
- numpy.polynomial
- numpy.lib
- numpy.ctypeslib
- numpy.testing
- numpy.distutils
- numpy.emath

Pandas

- *panel data*
- *not the animal !*

Attributes

- s.values
- s.index
- s.dtype
- s.size
- s.name



Creation

```
import pandas as pd

# 1. From a Python List
s1 = pd.Series([10, 20, 30, 40, 50])
# 2. From a List with Custom Index
s2 = pd.Series([10, 20, 30, 40], index=['a', 'b', 'c', 'd'])
# 3. From a NumPy Array
s3 = pd.Series(np.array([1.1, 2.2, 3.3]))
# 4. From a Dictionary (keys become index)
s4 = pd.Series({'a': 1, 'b': 2, 'c': 3})
# 5. From a Scalar Value (repeats value)
s5 = pd.Series(5, index=['a', 'b', 'c', 'd'])
# 6. From Another Series
s6 = pd.Series(s1)
# 7. Using Range Functions
s7 = pd.Series(range(5, 10))
# 8. From a Tuple
s8 = pd.Series((10, 20, 30, 40))
# 9. With Explicit Data Type
s9 = pd.Series([1, 2, 3], dtype='float64')
# 10. From a CSV File (single column)
s10 = pd.read_csv('data.csv', usecols=['column_name'], squeeze=True)
# 11. Create by Size
s11 = pd.Series([None]*10)          # Empty Series with nulls
```

```python
import pandas as pd

# Create initial Series
s = pd.Series([10, 20, 30, 40], index=['a', 'b', 'c', 'd'])
print(s)
```

i index	index	value
0	a	10
1	b	20
2	c	30
3	d	40

```
s['a'] = 100
print(s)
```

i index	index	value
0	a	10
1	b	20
2	c	30
3	d	40

i index	index	value
0	a	100
1	b	20
2	c	30
3	d	40

```
s[1] = 200
print(s)
```

i index	index	value
0	a	100
1	b	20
2	c	30
3	d	40

i index	index	value
0	a	100
1	b	200
2	c	30
3	d	40

FutureWarning: Series.__setitem__ treating keys as positions is deprecated. In a future version, integer keys will always be treated as labels (consistent with DataFrame behavior). To set a value by position, use `ser.iloc[pos] = value`
  s[1] = 200

```
s[['b', 'd']] = [150, 250]
print(s)
```

i index	index	value
0	a	100
1	b	200
2	c	30
3	d	40

i index	index	value
0	a	100
1	b	150
2	c	30
3	d	250

```
s[s > 150] = 999
print(s)
```

i index	index	value
0	a	100
1	b	150
2	c	30
3	d	250

i index	index	value
0	a	100
1	b	150
2	c	30
3	d	999

```python
s = s.where(s < 500, 80)
print(s)
```

i index	index	value
0	a	100
1	b	150
2	c	30
3	d	999

→

i index	index	value
0	a	100
1	b	150
2	c	30
3	d	80

```python
s = s.replace(80, 77)
print(s)
```

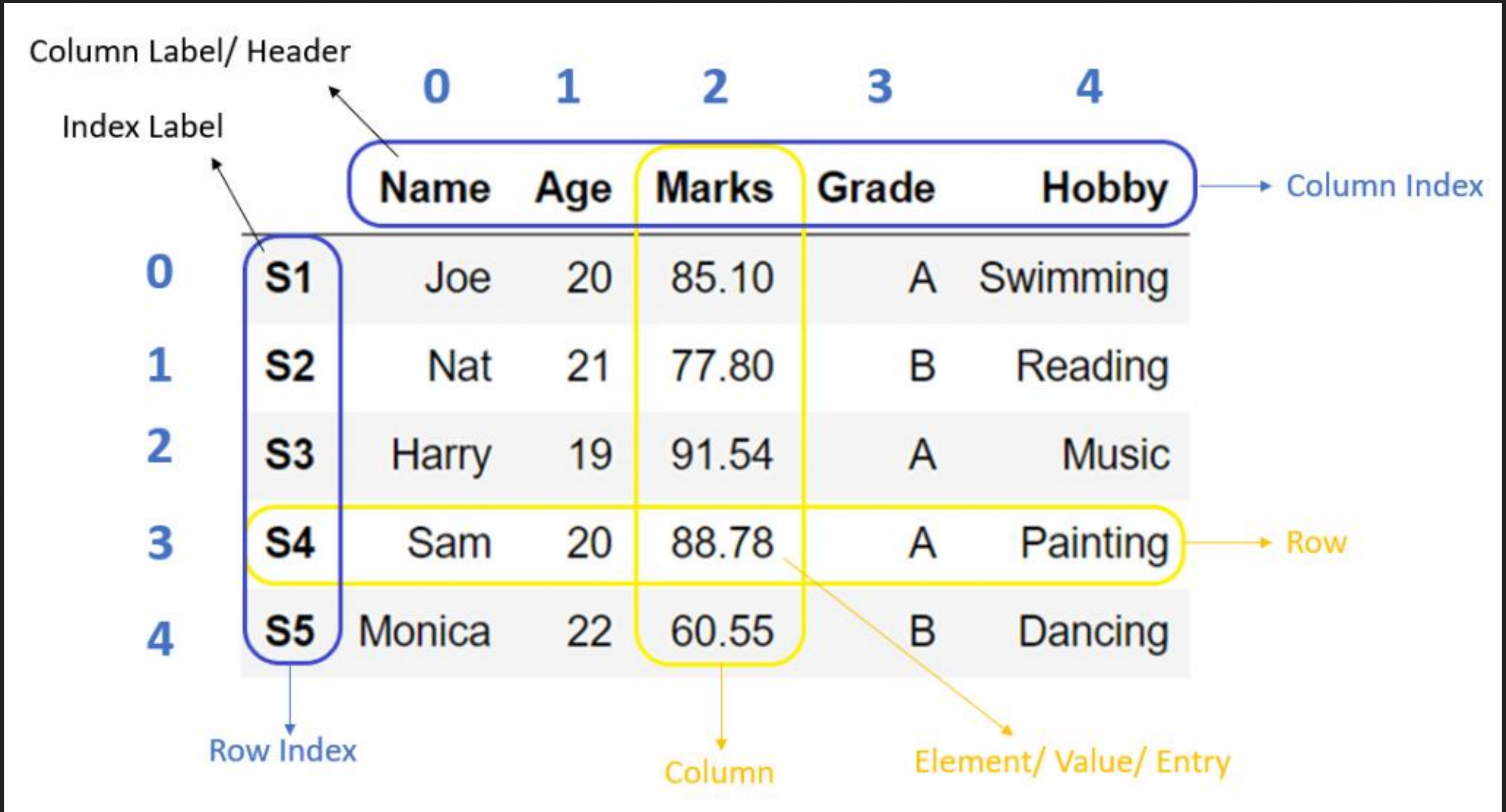
i index	index	value
0	a	100
1	b	150
2	c	30
3	d	80

i index	index	value
0	a	100
1	b	150
2	c	30
3	d	77

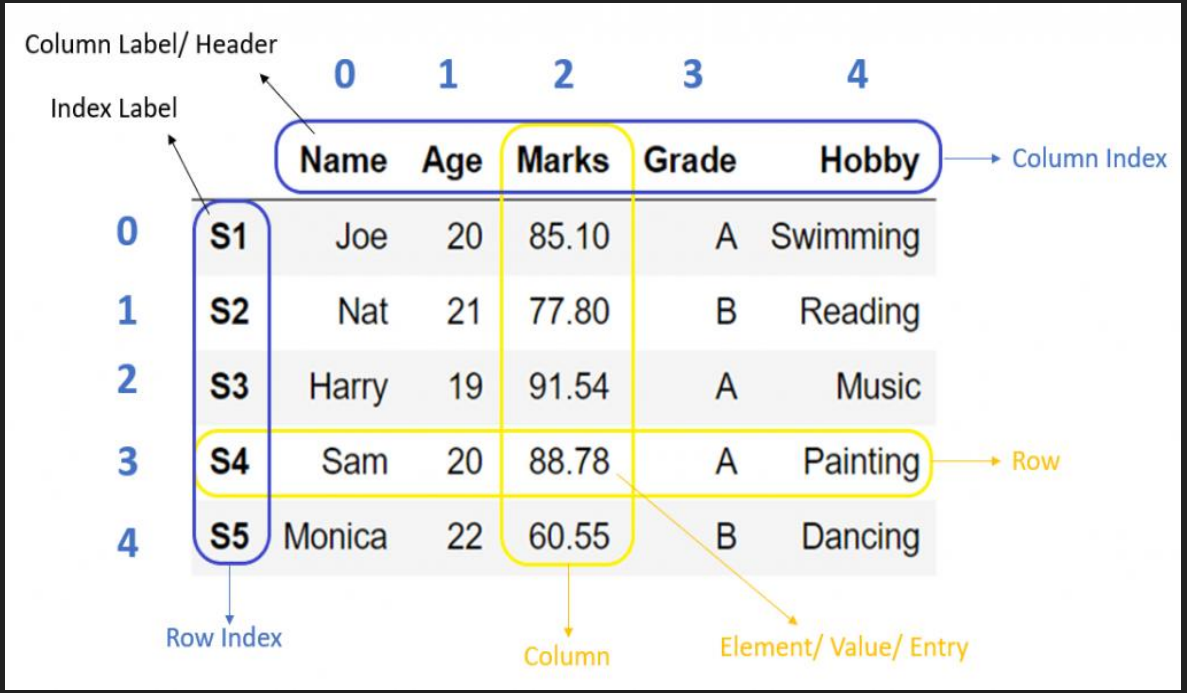
```python
import pandas as pd

s = pd.Series([10, 20, 30, 40], index=['a', 'b', 'c', 'd'])
print(s)

# 1. Complete index replacement
s.index = ['w', 'x', 'y', 'z']
print(s)

# 2. Rename specific indexes
s = s.rename({'x': 'xx', 'y': 'yy'})
print(s)

# 3. Reset index
s = s.reset_index()
print(s)

# 4. Set new index from values
s = pd.Series([10, 20, 30])
s.index = ['a', 'b', 'c']
print(s)

# 5. Delete elements
s = s.drop('b')
print(s)

# 6. Reindexing
s = s.reindex(['a', 'c', 'd', 'e'], fill_value=0)
print(s)

# 7. Adds new element
s['d'] = 400
print(s)
```

```
a    10
b    20
c    30
d    40
dtype: int64
w    10
x    20
y    30
z    40
dtype: int64
w     10
xx    20
yy    30
z     40
dtype: int64
  index  0
0    w  10
1   xx  20
3 2   yy  30
   z  40
a    10
b    20
c    30
dtype: int64
a    10
c    30
dtype: int64
c    30
a    10
d     0
e     0
dtype: int64
c     30
a     10
d      0
e      0
f    400
dtype: int64
```

DataFrame



Column Label/ Header
Index Label
0 1 2 3 4
Name Age Marks Grade Hobby
Column Index
0 S1 Joe 20 85.10 A Swimming
1 S2 Nat 21 77.80 B Reading
2 S3 Harry 19 91.54 A Music
3 S4 Sam 20 88.78 A Painting
Row
4 S5 Monica 22 60.55 B Dancing
Row Index
Column
Element/ Value/ Entry




```python
import pandas as pd

data = {
    'Name': ['Joe', 'Nat', 'Harry', 'Sam', 'Monica'],
    'Age': [20, 21, 19, 20, 22],
    'Marks': [85.10, 77.80, 91.54, 88.78, 60.55],
    'Grade': ['A', 'B', 'A', 'A', 'B'],
    'Hobby': ['Swimming', 'Reading', 'Music', 'Painting', 'Dancing']
}
df = pd.DataFrame(data, index=['S1', 'S2', 'S3', 'S4', 'S5'])

print("Complete DataFrame:")
print(df)

print("1. Shape (rows, columns):", df.shape)

print("2. Columns:")
print(df.columns)

# 3. Index attribute
print("\n3. Index:")
print(df.index)

print("\n4. Data Types:")
print(df.dtypes)


print("\n5. NumPy array of values:")
print(df.values)


print("\n6. Is DataFrame empty?", df.empty)

print("\n7. Memory usage:")
print(df.memory_usage())

print("\n8. Axes (row index and columns):")
print(df.axes)

print("\n9. Total elements (size):", df.size)

print("\n10. Number of dimensions (ndim):", df.ndim)

print("\n11. Numeric columns only:")
print(df.select_dtypes(include='number'))
```

```
Complete DataFrame:
     Name  Age  Marks Grade     Hobby
S1    Joe   20  85.10     A  Swimming
S2    Nat   21  77.80     B   Reading
S3  Harry   19  91.54     A     Music
S4    Sam   20  88.78     A  Painting
S5 Monica   22  60.55     B   Dancing
1. Shape (rows, columns): (5, 5)
2. Columns:
Index(['Name', 'Age', 'Marks', 'Grade', 'Hobby'], dtype='object')

3. Index:
Index(['S1', 'S2', 'S3', 'S4', 'S5'], dtype='object')

4. Data Types:
Name      object
Age        int64
Marks    float64
Grade     object
Hobby     object
dtype: object

5. NumPy array of values:
[['Joe' 20 85.1 'A' 'Swimming']
 ['Nat' 21 77.8 'B' 'Reading']
 ['Harry' 19 91.54 'A' 'Music']
 ['Sam' 20 88.78 'A' 'Painting']
 ['Monica' 22 60.55 'B' 'Dancing']]

6. Is DataFrame empty? False

7. Memory usage:
Index    40
Name     40
Age      40
Marks    40
Grade    40
Hobby    40
dtype: int64

8. Axes (row index and columns):
[Index(['S1', 'S2', 'S3', 'S4', 'S5'], dtype='object'), Index(
['Name', 'Age', 'Marks', 'Grade', 'Hobby'], dtype='object')]

9. Total elements (size): 25

10. Number of dimensions (ndim): 2

11. Numeric columns only:
    Age  Marks
S1   20  85.10
S2   21  77.80
S3   19  91.54
S4   20  88.78
S5   22  60.55
```

Creation

```python
import pandas as pd
import numpy as np
from sqlalchemy import create_engine
import requests

# 1. From a Dictionary
dict_data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'London', 'Paris']
}
df1 = pd.DataFrame(dict_data)

# 2. From a List of Lists
list_data = [
    ['Alice', 25, 'New York'],
    ['Bob', 30, 'London'],
    ['Charlie', 35, 'Paris']
]
df2 = pd.DataFrame(list_data, columns=['Name', 'Age', 'City'])

# 3. From a List of Dictionaries
dict_list = [
    {'Name': 'Alice', 'Age': 25, 'City': 'New York'},
    {'Name': 'Bob', 'Age': 30, 'City': 'London'},
    {'Name': 'Charlie', 'Age': 35, 'City': 'Paris'}
]
df3 = pd.DataFrame(dict_list)

# 4. From a NumPy Array
array = np.array([
    ['Alice', 25, 'New York'],
    ['Bob', 30, 'London'],
    ['Charlie', 35, 'Paris']
])
df4 = pd.DataFrame(array, columns=['Name', 'Age', 'City'])

# 5. From CSV (example - would need actual file)
df5 = pd.read_csv('data.csv')

# 6. From Excel (example - would need actual file)
df6 = pd.read_excel('data.xlsx', sheet_name='Sheet1')

# 7. From SQL Database (example - would need actual DB)
engine = create_engine('sqlite:///database.db')
df7 = pd.read_sql('SELECT * FROM table_name', engine)

# 8. From JSON
json_data = '[{"Name": "Alice", "Age": 25, "City": "New York"}, \
            {"Name": "Bob", "Age": 30, "City": "London"}, \
            {"Name": "Charlie", "Age": 35, "City": "Paris"}]'
df8 = pd.read_json(json_data)

# 9. From Series
names = pd.Series(['Alice', 'Bob', 'Charlie'])
ages = pd.Series([25, 30, 35])
df9 = pd.DataFrame({'Name': names, 'Age': ages})

# 10. Empty DataFrame
df10 = pd.DataFrame(columns=['Name', 'Age', 'City'])

# 11. From Dictionary of Series
df11 = pd.DataFrame({
    'Name': pd.Series(['Alice', 'Bob', 'Charlie']),
    'Age': pd.Series([25, 30, 35])
})

# 12. From Record Array
record_data = np.rec.array([
    ('Alice', 25, 'New York'),
    ('Bob', 30, 'London'),
    ('Charlie', 35, 'Paris')
], dtype=[('Name', 'U10'), ('Age', 'i4'), ('City', 'U10')])
df12 = pd.DataFrame(record_data)

# Display the first DataFrame as an example
print("DataFrame created from dictionary:")
print(df1)
```

Access, change, add!

```python
import pandas as pd

# 1. Create DataFrame
df = pd.DataFrame({
    'Price': [1.20, 0.50, 3.00, 2.50],
    'Stock': [50, 120, 15, 80],
    'Color': ['Red', 'Yellow', 'Dark Red', 'Brown']
}, index=['Apple', 'Banana', 'Cherry', 'Date'])  # Product names as index

print("Original DataFrame:")
print(df)
print("\n" + "="*40 + "\n")

# 2. Access data in different ways
# By label (loc)
print("Price of Banana:", df.loc['Banana', 'Price'])
print("All info for Cherry:\n", df.loc[['Cherry']])

# By position (iloc)
print("First product details:", df.iloc[0].to_dict())
print("Last item's color:", df.iloc[-1]['Color'])

# Conditional access
print("\nRed-colored fruits:")
print(df[df['Color'] == 'Dark Red'])

print("\n" + "="*40 + "\n")

# 3. Modify data
# Change Banana's price (using label)
df.loc['Banana', 'Price'] = 0.55

# Change all red fruits stock (using condition)
df.loc[df['Color'].str.contains('Red'), 'Stock'] += 10

print("After modifications:")
print(df)
print("\n" + "="*40 + "\n")

# 4. Add new data
# Add new column
df['OnSale'] = [False, True, False, True]

# Add new fruit (Elderberry)
df.loc['Elderberry'] = [4.20, 25, 'Purple', False]

print("Final DataFrame:")
print(df)
```

```
Original DataFrame:
        Price  Stock     Color
Apple     1.2     50       Red
Banana    0.5    120    Yellow
Cherry    3.0     15  Dark Red
Date      2.5     80     Brown

========================================

Price of Banana: 0.5
All info for Cherry:
        Price  Stock     Color
Cherry    3.0     15  Dark Red
First product details: {'Price': 1.2, 'Stock': 50, 'Color': 'Red'}
Last item's color: Brown

Red-colored fruits:
        Price  Stock     Color
Cherry    3.0     15  Dark Red

========================================

After modifications:
        Price  Stock     Color
Apple    1.20     60       Red
Banana   0.55    120    Yellow
Cherry   3.00     25  Dark Red
Date     2.50     80     Brown

========================================

Final DataFrame:
            Price  Stock     Color  OnSale
Apple        1.20     60       Red   False
Banana       0.55    120    Yellow    True
Cherry       3.00     25  Dark Red   False
Date         2.50     80     Brown    True
Elderberry   4.20     25    Purple   False
```

Delete

```python
import pandas as pd

# Create sample DataFrame
df = pd.DataFrame({
    'Price': [1.20, 0.50, 3.00, 2.50],
    'Stock': [50, 120, 15, 80],
    'Color': ['Red', 'Yellow', 'Dark Red',
'Brown']
}, index=['Apple', 'Banana', 'Cherry',
'Date'])

print("Original DataFrame:")
print(df)

# Delete the 'Color' column
df.drop('Color', axis=1, inplace=True)
print("\nAfter deleting 'Color' column:")
print(df)

# Delete 'Banana' and 'Date' rows
df.drop(['Banana', 'Date'], inplace=True)
print("\nAfter deleting Banana and Date
rows:")
print(df)

# Conditional deletion - remove items
with stock < 20
df = df[df['Stock'] >= 20]
print("\nAfter removing low-stock
items:")
print(df)
```

```
Original DataFrame:
        Price  Stock     Color
Apple     1.2     50       Red
Banana    0.5    120    Yellow
Cherry    3.0     15  Dark Red
Date      2.5     80     Brown

After deleting 'Color' column:
        Price  Stock
Apple     1.2     50
Banana    0.5    120
Cherry    3.0     15
Date      2.5     80

After deleting Banana and Date rows:
        Price  Stock
Apple     1.2     50
Cherry    3.0     15

After removing low-stock items:
       Price  Stock
Apple    1.2     50
```

Search: Basic Boolean Indexing

- # Find all products with price > 2
- expensive = df[df['Price'] > 2]

- # Find products with stock between 50-100
- medium_stock = df[(df['Stock'] >= 50) & (df['Stock'] <= 100)]

- # Find red-colored products
- red_items = df[df['Color'] == 'Red']

String Operations

- # Find products containing 'erry' in their name
- erry_products = df[df.index.str.contains('erry')]

- # Find colors starting with 'Y'
- yellowish = df[df['Color'].str.startswith('Y')]

- # Case-insensitive search
- red_items = df[df['Color'].str.lower().str.contains('red')]

isin() Method

- # Find specific products
- selected = df[df.index.isin(['Apple', 'Banana'])]

- # Find multiple colors
- color_filter = df[df['Color'].isin(['Red', 'Yellow'])]

query() Method (SQL-like syntax)

- # Simple query
- result = df.query('Price > 2')

- # Multiple conditions
- result = df.query('Price > 1 and Stock < 100')

- # Using variables in query
- min_price = 1.5
- result = df.query('Price > @min_price')

loc/iloc with Conditions

- # Get specific columns for matching rows
- result = df.loc[df['Price'] > 2, ['Price', 'Stock']]

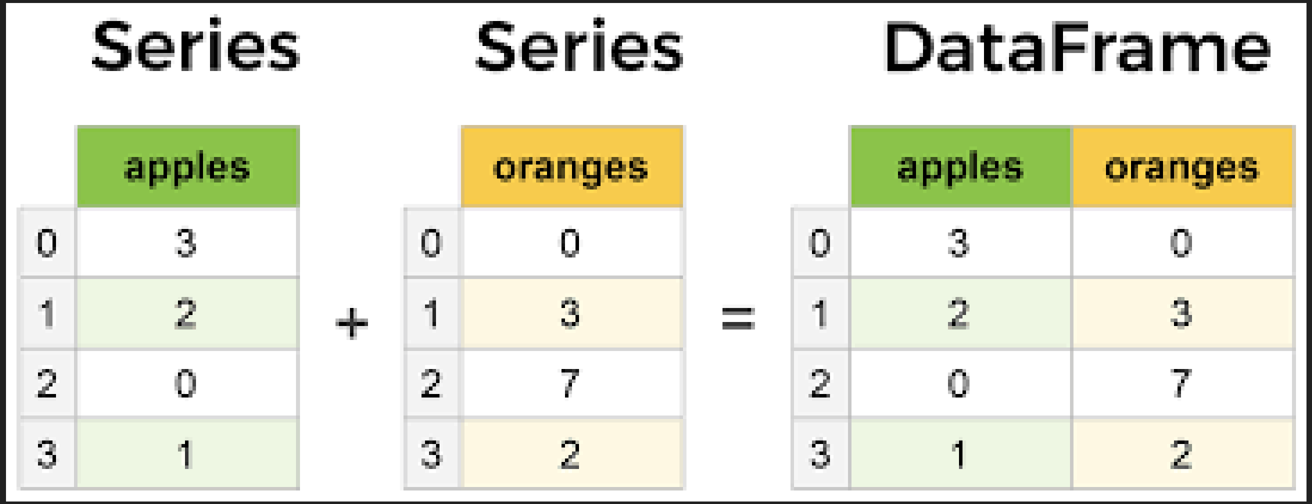
- # Get first 2 matching rows
- result = df[df['Price'] > 1].iloc[:2]

where() Method (keeps structure)

- # Shows NaN for non-matching entries
- filtered = df.where(df['Price'] > 1)

- # With custom replacement
- filtered = df.where(df['Stock'] > 50, 'Low Stock')

Logic


Series
Series
DataFrame
apples
oranges
apples
oranges
0 3
1 2
2 0
3 1
+
0 0
1 3
2 7
3 2
=
0 3 0
1 2 3
2 0 7
3 1 2

The End