Week 4: NumPy and Pandas

Numerical Computing and Data Manipulation

Dr. Eyuphan Koc

Bogazici University

Fall 2025

Based on "Python Data Science Handbook" by Jake VanderPlas
Chapter 2: Introduction to NumPy
Chapter 3: Data Manipulation with Pandas (Sections 3.0-3.6)
`https://github.com/jakevdp/PythonDataScienceHandbook`

This Week's Roadmap

NumPy Topics (Sections 1-9)

1. Introduction to NumPy
2. Array Fundamentals
3. Array Operations & Ufuncs
4. Aggregations & Statistics
5. Broadcasting
6. Boolean Operations
7. Advanced Indexing
8. Sorting
9. Structured Data

Pandas Topics (Sections 10-16)

10. Introduction to Pandas
11. Pandas Core Objects
12. Data Indexing & Selection
13. Operations in Pandas
14. Handling Missing Data
15. Hierarchical Indexing
16. Combining Datasets
17. Summary & Next Steps

What is NumPy?

NumPy = Numerical Python

- Fundamental package for scientific computing in Python
- Provides fast, efficient multi-dimensional arrays
- Foundation for Pandas, SciPy, Matplotlib, and more
- Written in C - blazing fast performance

Why NumPy?

- Efficient operations on large datasets
- Vectorized computations (no loops!)
- Broadcasting for operations on different shapes
- Essential for data science and scientific computing
- Foundation for the entire PyData ecosystem

[TOGETHER] Speed Comparison: Lists vs NumPy

Python Lists

- Flexible but slow
- Type checking at every operation
- Overhead for each element

```python
# Python list: slow
import time
L = list(range(1000000))
start = time.time()
result = [x**2 for x in L]
print(f"Time: {time.time()-start:.4f}s")
# Time: ~0.15s
```

NumPy Arrays

- Fixed type - no checking
- Contiguous memory block
- Vectorized C operations

```python
# NumPy array: fast!
import numpy as np
A = np.arange(1000000)
start = time.time()
result = A**2
print(f"Time: {time.time()-start:.4f}s")
# Time: ~0.002s  (75x faster!)
```

Key Takeaway

NumPy is 10-100x faster for numerical operations! Essential for large engineering datasets.

Installing and Importing NumPy

Installation

```
# Using pip
pip install numpy

# Using conda (recommended for data science)
conda install numpy
```

Standard Import Convention

```python
import numpy as np  # ALWAYS use this convention!

# Check version
print(np.__version__)  # e.g., '1.24.3'
```

Why np?

Universal convention in data science community - makes code readable to everyone

Creating Your First NumPy Arrays

```python
import numpy as np

# From Python list - 1D array
my_list = [1, 2, 3, 4, 5]
arr = np.array(my_list)
print(arr)  # [1 2 3 4 5]

# 2D array (matrix)
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])
print(matrix)
# [[1 2 3]
#  [4 5 6]
#  [7 8 9]]

# Unlike lists, all elements must be same type!
mixed = np.array([1, 2.5, 3])  # Will convert to float
print(mixed.dtype)  # dtype('float64')
```

Creating Arrays: Common Methods

```python
import numpy as np

# Zeros - initialize array
zeros = np.zeros(10)  # [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
zeros_matrix = np.zeros((3, 3))  # 3x3 matrix of zeros

# Ones
ones = np.ones(5)  # [1. 1. 1. 1. 1.]
fives = np.ones(5) * 5  # [5. 5. 5. 5. 5.]

# Sequential values
seq = np.arange(0, 11, 2)  # [0 2 4 6 8 10]  (start, stop, step)

# Evenly spaced values
linear = np.linspace(0, 1, 5)  # [0.   0.25  0.5  0.75  1.] (start, stop, count)

# Random values
random_vals = np.random.random(5)  # 5 random values between 0 and 1
random_ints = np.random.randint(0, 10, size=5)  # 5 random integers 0-9

# Identity matrix
I = np.eye(3)  # [[1. 0. 0.], [0. 1. 0.], [0. 0. 1.]]
```

Array Attributes: Understanding Your Data

```python
import numpy as np
np.random.seed(0)  # for reproducibility

# Create arrays of different dimensions
x1 = np.random.randint(10, size=6)  # 1D array
x2 = np.random.randint(10, size=(3, 4))  # 2D array
x3 = np.random.randint(10, size=(3, 4, 5))  # 3D array

print("x3 ndim: ", x3.ndim)  # 3
print("x3 shape:", x3.shape)  # (3, 4, 5)
print("x3 size: ", x3.size)  # 60

print("dtype:", x3.dtype)  # int64
print("itemsize:", x3.itemsize, "bytes")  # 8 bytes
print("nbytes:", x3.nbytes, "bytes")  # 480 bytes
```

Key Attributes

- **ndim**: number of dimensions
- **shape**: size of each dimension
- **size**: total number of elements
- **dtype**: data type of elements

Data Types in NumPy

Common Data Types

- `int32`, `int64`: Integers
- `float32`, `float64`: Floats
- `complex128`: Complex numbers
- `bool`: True/False

```python
# Specify dtype at creation
arr = np.array([1, 2, 3],
               dtype=np.float32)

# Convert dtype
arr_float64 = arr.astype(np.float64)

print(arr.dtype)          # float32
print(arr_float64.dtype)  # float64
```

Why It Matters

- Memory: `float32` uses half the space
- Precision: `float64` for high accuracy
- Speed: Smaller types = faster processing

```python
# Memory comparison
a32 = np.ones(1000000, dtype=np.float32)
a64 = np.ones(1000000, dtype=np.float64)

print(f"32-bit: {a32.nbytes/1e6} MB")
# 4.0 MB
print(f"64-bit: {a64.nbytes/1e6} MB")
# 8.0 MB
```

Indexing and Slicing: Accessing Array Elements

```python
import numpy as np

# 1D array - similar to Python lists
x = np.arange(10)  # [0 1 2 3 4 5 6 7 8 9]
print(x[0])       # 0 (first element)
print(x[-1])      # 9 (last element)
print(x[4:7])     # [4 5 6] (slice from index 4 to 6)
print(x[::2])     # [0 2 4 6 8] (every 2nd element)
print(x[::-1])    # [9 8 7 6 5 4 3 2 1 0] (reverse)

# 2D array - row, column indexing
x2 = np.array([[12, 5, 2, 4],
               [7, 6, 8, 8],
               [1, 6, 7, 7]])

print(x2[0, 0])    # 12 (element at row 0, col 0)
print(x2[2, -1])   # 7 (element at row 2, last column)
print(x2[0, :])    # [12  5  2  4] (first row)
print(x2[:, 1])    # [5 6 6] (second column)
print(x2[:2, :2])  # [[12  5], [ 7  6]] (2x2 subarray)
```

[TOGETHER] Array Views vs Copies

```python
import numpy as np

# Original array
original = np.array([1, 2, 3, 4, 5])

# SLICING creates a VIEW (not a copy!)
view = original[1:4]
view[0] = 999
print(original)  # [1 999 3 4 5]  <-- Original changed!

# To create independent copy, use .copy()
original = np.array([1, 2, 3, 4, 5])
independent = original[1:4].copy()
independent[0] = 999
print(original)  # [1 2 3 4 5]  <-- Original unchanged
print(independent)  # [999 3 4]
```

Critical for Engineering!

Views save memory but can cause bugs. Always use `.copy()` when you need independent data.

Reshaping Arrays

```python
import numpy as np

# 1D to 2D
loads = np.arange(12)
print(loads)  # [ 0  1  2  3  4  5  6  7  8  9 10 11]

# Reshape to 3x4 matrix
matrix = loads.reshape(3, 4)
print(matrix)
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]

# Reshape to 4x3 matrix
matrix2 = loads.reshape(4, 3)
print(matrix2)
# [[ 0  1  2]
#  [ 3  4  5]
#  [ 6  7  8]
#  [ 9 10 11]]

# Use -1 to auto-calculate one dimension
matrix3 = loads.reshape(2, -1)  # 2 rows, auto-calculate columns
print(matrix3.shape)  # (2, 6)

# Flatten back to 1D
flat = matrix.flatten()  # or .ravel() for view
print(flat)  # [ 0  1  2  3  4  5  6  7  8  9 10 11]
```

Concatenating and Splitting Arrays

```python
import numpy as np

# Concatenate 1D arrays
dead_load = np.array([10, 15, 20])
live_load = np.array([5, 8, 10])
total_load = np.concatenate([dead_load, live_load])
print(total_load)  # [10 15 20  5  8 10]

# Stack vertically (vstack)
loads = np.vstack([dead_load, live_load])
print(loads)
# [[10 15 20]
#  [ 5  8 10]]

# Stack horizontally (hstack)
loads = np.hstack([dead_load, live_load])
print(loads)  # [10 15 20  5  8 10]

# Split array
split_loads = np.split(total_load, [3])  # Split at index 3
print(split_loads[0])  # [10 15 20]
print(split_loads[1])  # [ 5  8 10]
```

Vectorized Arithmetic: No Loops Needed!

```python
import numpy as np

# Create arrays
x = np.arange(4)
print("x =", x)  # [0 1 2 3]

# Element-wise operations (vectorized - FAST!)
print("x + 5 =", x + 5)     # [5 6 7 8]
print("x - 5 =", x - 5)     # [-5 -4 -3 -2]
print("x * 2 =", x * 2)     # [0 2 4 6]
print("x / 2 =", x / 2)     # [0.  0.5 1.  1.5]
print("x ** 2 =", x ** 2)   # [0 1 4 9]

# Multiple arrays
a = np.array([1, 2, 3, 4])
b = np.array([4, 3, 2, 1])
print("a + b =", a + b)     # [5 5 5 5]
print("a * b =", a * b)     # [4 6 6 4]

# Compare to Python list (requires loop!)
# result = [x**2 for x in my_list]  # Slow!
```

Universal Functions (ufuncs): Fast Operations

```python
import numpy as np

# Trigonometric functions
theta = np.linspace(0, np.pi, 3)
print("sin(theta) =", np.sin(theta))
print("cos(theta) =", np.cos(theta))
print("tan(theta) =", np.tan(theta))

# Exponential and logarithmic
x = [1, 2, 3]
print("e^x =", np.exp(x))         # [2.718  7.389  20.086]
print("2^x =", np.exp2(x))        # [2.  4.  8.]
print("log(x) =", np.log(x))     # [0.  0.693  1.099]

# Absolute value
x = np.array([-2, -1, 0, 1, 2])
print("abs(x) =", np.abs(x))     # [2 1 0 1 2]

# All much faster than Python loops!
```

[EXPLORE] Example: Computation on Arrays

```python
import numpy as np

# Compute values of sin(x) for many values
x = np.linspace(0, np.pi, 3)
print("x      =", x)
# [0.         1.57079633 3.14159265]

print("sin(x) =", np.sin(x))
# [0.0000000e+00 1.0000000e+00 1.2246468e-16]

# Compute a more complex operation
x = np.arange(5)
y = np.empty(5)
for i in range(5):
    y[i] = x[i] ** 2
print(y)  # [ 0.  1.  4.  9. 16.]

# Much better with vectorization:
x = np.arange(5)
y = x ** 2
print(y)  # [ 0  1  4  9 16]
```

Basic Aggregations: Summarizing Data

```python
import numpy as np

# Random data
L = np.random.random(100)

# Summary statistics
print(np.sum(L))       # Sum of all values
print(np.min(L))       # Minimum value
print(np.max(L))       # Maximum value
print(np.mean(L))      # Mean
print(np.std(L))       # Standard deviation
print(np.var(L))       # Variance

# These also work as array methods:
print(L.sum())
print(L.min())
print(L.max())
print(L.mean())
print(L.std())

# Percentiles
print(np.percentile(L, 25))  # 1st quartile
print(np.median(L))          # 50th percentile
print(np.percentile(L, 75))  # 3rd quartile
```

Multi-Dimensional Aggregations: The axis Parameter

```python
import numpy as np

# 2D array example
M = np.random.random((3, 4))
print(M)

# Aggregate along different axes
print("Shape:", M.shape)  # (3, 4)

# Sum all values
print(M.sum())

# Sum along axis 0 (collapse rows -> result has shape (4,))
print(M.sum(axis=0))

# Sum along axis 1 (collapse columns -> result has shape (3,))
print(M.sum(axis=1))

# Works with other functions too:
print(M.min(axis=0))  # Min of each column
print(M.max(axis=1))  # Max of each row
```

More Aggregation Functions

```python
import numpy as np

data = np.array([10, 15, 20, 25, 30, 35, 40])

# Basic stats
print(f"Sum: {np.sum(data)}")          # 175
print(f"Product: {np.prod(data)}")     # 3.15e9
print(f"Mean: {np.mean(data)}")        # 25.0
print(f"Std: {np.std(data)}")          # 10.0
print(f"Variance: {np.var(data)}")     # 100.0

# Min/Max
print(f"Min: {np.min(data)}")          # 10
print(f"Max: {np.max(data)}")          # 40
print(f"Argmin: {np.argmin(data)}")    # 0 (index of min)
print(f"Argmax: {np.argmax(data)}")    # 6 (index of max)

# Cumulative operations
cumsum = np.cumsum(data)  # [10 25 45 70 100 135 175]
cumprod = np.cumprod(data[:4])  # [10 150 3000 75000]

# Boolean operations
print(f"Any > 50: {np.any(data > 50)}")    # False
print(f"All > 5: {np.all(data > 5)}")      # True
```

[TOGETHER] Example: Analyzing Multi-Dimensional Data

```python
import numpy as np

# Precipitation data: 12 months x 5 years
precip_data = np.array([
    [3.2, 2.8, 3.5, 2.9, 3.1],  # January
    [2.5, 2.9, 2.3, 2.7, 2.6],  # February
    [3.8, 4.1, 3.6, 4.0, 3.9],  # March
    # ... (more months)
])

# Analysis
mean_per_month = np.mean(precip_data, axis=1)
mean_per_year = np.mean(precip_data, axis=0)
overall_mean = np.mean(precip_data)
overall_std = np.std(precip_data)

# Find extremes
max_precip = np.max(precip_data)
min_precip = np.min(precip_data)

print(f"Overall mean: {overall_mean:.2f}")
print(f"Std deviation: {overall_std:.2f}")
print(f"Range: {min_precip:.2f} - {max_precip:.2f}")
```

Broadcasting: Operating on Different Shapes

What is Broadcasting?

Broadcasting allows NumPy to perform operations on arrays of different shapes without explicitly replicating data.

Broadcasting Rules

1. If arrays have different dimensions, pad smaller shape with 1s on the left
2. If shapes don't match in a dimension, stretch dimension with size 1
3. If sizes disagree and neither is 1, raise error

Why It Matters

No loops needed! Operations are fast and memory-efficient. Essential for data science operations on large datasets.

Broadcasting Examples

```
import numpy as np

# Scalar + Array (broadcasts scalar to all elements)
a = np.array([0, 1, 2])
a + 5  # array([5, 6, 7])

# 1D + 1D
a = np.ones((3, 3))
b = np.arange(3)
a + b
# array([[1., 2., 3.],
#        [1., 2., 3.],
#        [1., 2., 3.]])

# Broadcasting with higher dimensions
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
print(a + b)
# [[0 1 2]
#  [1 2 3]
#  [2 3 4]]
```

Broadcasting in 2D: Centering Data

```python
import numpy as np

# Data matrix (10 observations x 3 features)
X = np.random.random((10, 3))

# Compute mean of each column (feature)
Xmean = X.mean(axis=0)

# Center the data (subtract mean from each column)
X_centered = X - Xmean

# Verify mean is now ~0 for each feature
print(X_centered.mean(axis=0))
# [~0. ~0. ~0.]

# This works because Xmean has shape (3,) which broadcasts
# to match X's shape (10, 3) by replicating across rows
```

[EXPLORE] Broadcasting: Plotting Functions

```python
import numpy as np
import matplotlib.pyplot as plt

# Create a 2D grid using broadcasting
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 50)[:, np.newaxis]

# Broadcasting: x is (50,), y is (50,1)
# Result z is (50, 50)
z = np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)

plt.imshow(z, origin='lower', extent=[0, 5, 0, 5],
           cmap='viridis')
plt.colorbar()
plt.title('Broadcasting Example')
plt.show()

# This creates a 2D function from 1D arrays!
```

```python
import numpy as np

# Data array
x = np.array([1, 2, 3, 4, 5])

# Comparison operators return boolean arrays
print(x < 3)
# [ True  True False False False]

print(x > 3)
# [False False False  True  True]

print(x == 3)
# [False False  True False False]

# Multiple comparisons (use & and |, not 'and' and 'or')
print((x > 1) & (x < 5))
# [False  True  True  True False]

# Count how many satisfy condition
print(np.sum(x > 2))  # 3 (True=1, False=0)

# Percentage
print(np.mean(x > 2))  # 0.6 (60%)
```

Boolean Indexing: Filtering Data

```python
import numpy as np

# Data array
x = np.array([1, 2, 3, 4, 5])

# Create boolean mask
mask = x < 3
print(mask)
# [ True  True False False False]

# Use mask to filter data
print(x[mask])  # [1 2]

# Can use directly without creating variable
print(x[x < 3])  # [1 2]

# More complex example with 2D
np.random.seed(0)
X = np.random.randint(10, size=(3, 4))
print(X)
# [[5 0 3 3]
#  [7 9 3 5]
#  [2 4 7 6]]

print(X[X < 5])  # [0 3 3 3 2 4] (flattened)
```

Boolean Operators: Combining Conditions

```python
import numpy as np

# Example: Rainy days analysis
rainfall_inches = np.array([0.2, 0.5, 0.0, 1.2, 0.8, 0.0, 0.3])

# Multiple criteria with & (AND) and | (OR)
print((rainfall_inches > 0) & (rainfall_inches < 1))
# [ True  True False False  True False  True]

print((rainfall_inches <= 0) | (rainfall_inches >= 1))
# [False False  True  True False  True False]

# Use np.sum() to count matches
print(np.sum((rainfall_inches > 0) & (rainfall_inches < 1)))  # 4

# IMPORTANT: Use & and | for arrays, not 'and' and 'or'!
# Also: always use parentheses around conditions

# Boolean operators
print(~(rainfall_inches > 0.5))  # NOT
# [ True False  True False False  True  True]
```

[TOGETHER] Example: Analyzing Weather Data

```python
import numpy as np

# Weather data
np.random.seed(1)
rainfall = np.random.random(365) * 2  # inches per day

# Analysis
rainy_days = np.sum(rainfall > 0.5)
dry_days = np.sum(rainfall < 0.1)
median_precip = np.median(rainfall)
mean_precip = np.mean(rainfall)

print(f"Rainy days (>0.5 in): {rainy_days}")
print(f"Dry days (<0.1 in): {dry_days}")
print(f"Median: {median_precip:.2f} in")
print(f"Mean: {mean_precip:.2f} in")

# Get all rainy day amounts
rainy = rainfall[rainfall > 0.5]
print(f"Average rainfall on rainy days: {rainy.mean():.2f} in")
```

```python
import numpy as np

# Simple array
x = np.array([51, 92, 14, 71, 60, 20, 82, 86, 74, 74])

# Select specific elements by index
ind = [3, 7, 4]
print(x[ind])  # [71 86 60]

# 2D indexing
X = np.arange(12).reshape((3, 4))
print(X)
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]

# Select specific rows and columns
row = np.array([0, 1, 2])
col = np.array([2, 1, 3])
print(X[row, col])  # [ 2  5 11]

# Select a subset of rows
print(X[[0, 2]])
# [[ 0  1  2  3]
#  [ 8  9 10 11]]
```

Modifying Values with Fancy Indexing

```python
import numpy as np

# Start with array of zeros
x = np.zeros(10)
print(x)  # [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

# Set specific indices
ind = [0, 3, 5]
x[ind] = 99
print(x)  # [99.  0.  0. 99.  0. 99.  0.  0.  0.  0.]

# Increment specific values
x[ind] += 1
print(x)  # [100.   0.   0. 100.   0. 100.   0.   0.   0.   0.]

# Repeated indices - behavior is subtle!
x = np.zeros(5)
i = [0, 0, 0]
x[i] += 1
print(x)  # [1. 0. 0. 0. 0.] - only incremented once!
```

Combined Indexing: Mix and Match

```python
import numpy as np

# 2D array
X = np.arange(12).reshape((3, 4))
print(X)
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]

# Fancy indexing + slicing
# Select rows 0 and 2, columns 1 and 3
result = X[[0, 2]][:, [1, 3]]
print(result)
# [[ 1  3]
#  [ 9 11]]

# Boolean mask + fancy indexing
mask = X[:, 1] > 5
print(mask)  # [False False  True]
print(X[mask])
# [[ 8  9 10 11]]
```

Sorting Arrays

```python
import numpy as np

# Unsorted array
x = np.array([2, 1, 4, 3, 5])

# Sort (returns new sorted array)
print(np.sort(x))  # [1 2 3 4 5]

# argsort: returns indices that would sort the array
i = np.argsort(x)
print(i)  # [1 0 3 2 4]
print(x[i])  # [1 2 3 4 5]

# Sort in descending order
print(x[np.argsort(x)[::-1]])  # [5 4 3 2 1]

# Sort 2D array along axis
np.random.seed(42)
X = np.random.randint(0, 10, (4, 6))
print(X)

# Sort each column
print(np.sort(X, axis=0))

# Sort each row
print(np.sort(X, axis=1))
```

Practical Sorting: Finding Top N Elements

```python
import numpy as np

# Array of values
x = np.array([7, 2, 3, 1, 6, 5, 4])

# Partition: smallest 3 on left, rest on right
print(np.partition(x, 3))
# [2 1 3 4 6 5 7] (3 smallest values on left, not sorted)

# Get indices for partition
i = np.argpartition(x, 3)
print(x[i])
# [2 1 3 4 6 5 7]

# Find top K values efficiently
# Partition so K largest are on the right
K = 3
partitioned = np.partition(x, -K)
print(partitioned[-K:])  # [5 6 7] (not necessarily sorted)

# For sorted top K, use argsort
top_k_sorted = x[np.argsort(x)[-K:]]
print(top_k_sorted)  # [5 6 7]
```

Structured Arrays: Mixing Data Types

```python
import numpy as np

# Create structured array for person data
data = np.zeros(4, dtype={
    'names': ('name', 'age', 'weight'),
    'formats': ('U10', 'i4', 'f8')
})

# Fill data
data['name'] = ['Alice', 'Bob', 'Cathy', 'Doug']
data['age'] = [25, 45, 37, 19]
data['weight'] = [55.0, 85.5, 68.0, 61.5]

print(data)
# [('Alice', 25, 55. ) ('Bob', 45, 85.5)
#  ('Cathy', 37, 68. ) ('Doug', 19, 61.5)]

# Access by field name
print(data['name'])  # ['Alice' 'Bob' 'Cathy' 'Doug']
print(data['age'])   # [25 45 37 19]

# Filter
print(data[data['age'] < 30]['name'])  # ['Alice' 'Doug']
```

Note

For complex data, Pandas DataFrames are usually better! Let's learn about them now.

From NumPy to Pandas: The Next Step

What is Pandas?

- Built on top of NumPy
- Provides DataFrame: labeled, 2D data structure
- Like Excel or SQL tables, but in Python
- Industry standard for data manipulation
- Essential for real-world data analysis

Why Pandas After NumPy?

- NumPy: Fast arrays, but no labels or structure
- Pandas: Labels + missing data + heterogeneous types
- Access data by name, not just index position
- Built-in tools for reading CSV, Excel, SQL
- Better for messy, real-world data

Installing and Importing Pandas

Installation

```
# Using pip
pip install pandas

# Using conda (recommended for data science)
conda install pandas
```

Standard Import Convention

```python
import pandas as pd  # ALWAYS use this convention!
import numpy as np   # Often used together

# Check version
print(pd.__version__)  # e.g., '2.0.0'
```

Universal Convention

Like NumPy's `np`, always import Pandas as `pd`. This is the universal standard!

NumPy vs Pandas: A Quick Comparison

NumPy Array

- Access by integer index
- No column names
- Homogeneous types
- Fast but minimal structure

```
# NumPy array
data = np.array([[1, 2, 3],
                 [4, 5, 6]])
print(data[0, 1])  # 2
# Which column is this? Must remember!
```

Pandas DataFrame

- Access by label or index
- Named columns and rows
- Mixed types allowed
- More features, slight overhead

```
# Pandas DataFrame
df = pd.DataFrame([[1, 2, 3],
                   [4, 5, 6]],
                  columns=['A', 'B', 'C'])
print(df['B'][0])  # 2
# Clear what column 'B' means!
```

The Pandas Ecosystem

Core Data Structures

- **Series**: 1D labeled array (like a column)
- **DataFrame**: 2D labeled table (like a spreadsheet)
- **Index**: Row and column labels

Common Use Cases

- Loading and cleaning CSV/Excel data
- Time series analysis (stock prices, weather)
- Database-style operations (join, merge, group)
- Handling missing data
- Statistical analysis and visualization
- Data preprocessing for machine learning

Learning Path

Master NumPy first (✓), then Pandas builds naturally on top!

The Pandas Series: 1D Labeled Array

```python
import pandas as pd
import numpy as np

# Create Series from list
data = pd.Series([0.25, 0.5, 0.75, 1.0])
print(data)
# 0    0.25
# 1    0.50
# 2    0.75
# 3    1.00
# dtype: float64

# Access values and index
print(data.values)  # array([0.25, 0.5 , 0.75, 1.  ])
print(data.index)   # RangeIndex(start=0, stop=4, step=1)

# Access like array
print(data[1])      # 0.5
print(data[1:3])    # Series with indices 1, 2
```

Series with Custom Index

```python
import pandas as pd

# Series with string index (like a dictionary!)
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
print(data)
# a    0.25
# b    0.50
# c    0.75
# d    1.00

# Access by label
print(data['b'])  # 0.5

# Can use non-contiguous indices
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=[2, 5, 3, 7])
print(data[5])  # 0.5
```

Series from Dictionary

```python
import pandas as pd

# Create Series from dictionary
population_dict = {
    'California': 38332521,
    'Texas': 26448193,
    'New York': 19651127,
    'Florida': 19552860,
    'Illinois': 12882135
}
population = pd.Series(population_dict)
print(population)
# California    38332521
# Florida       19552860
# Illinois      12882135
# New York      19651127
# Texas         26448193

# Dictionary-style access
print(population['California'])  # 38332521

# Array-style slicing
print(population['California':'Illinois'])
```

The DataFrame: 2D Labeled Data Structure

```
import pandas as pd

# Create DataFrame from dictionary of Series
area_dict = {'California': 423967, 'Texas': 695662,
             'New York': 141297, 'Florida': 170312,
             'Illinois': 149995}
area = pd.Series(area_dict)

states = pd.DataFrame({'population': population,
                       'area': area})
print(states)
#               area  population
# California  423967    38332521
# Florida     170312    19552860
# Illinois    149995    12882135
# New York    141297    19651127
# Texas       695662    26448193

print(states.index)    # State names
print(states.columns)  # ['area', 'population']
```

[TOGETHER] Creating DataFrames: Multiple Ways

```python
import pandas as pd
import numpy as np

# From dictionary of lists
df1 = pd.DataFrame({'A': [1, 2, 3],
                    'B': [4, 5, 6]})

# From list of dictionaries
df2 = pd.DataFrame([{'a': 1, 'b': 2},
                    {'a': 3, 'b': 4, 'c': 5}])
print(df2)
#      a  b    c
# 0  1.0  2  NaN
# 1  3.0  4  5.0

# From NumPy array
df3 = pd.DataFrame(np.random.rand(3, 2),
                   columns=['foo', 'bar'],
                   index=['a', 'b', 'c'])

# Add new column
states['density'] = states['population'] / states['area']
```

DataFrame as Dictionary of Series

```python
import pandas as pd

# Access column (returns Series)
print(states['area'])
# California    423967
# Florida       170312
# ...

# Also works with attribute-style access
print(states.area)  # Same as states['area']

# Check if they're the same object
print(states.area is states['area'])  # True

# Add new column
states['density'] = states['population'] / states['area']
print(states)
#               area  population     density
# California  423967    38332521   90.413926
# Florida     170312    19552860  114.806121
# ...
```

The Index Object

```python
import pandas as pd

# Create Index
ind = pd.Index([2, 3, 5, 7, 11])
print(ind)  # Index([2, 3, 5, 7, 11], dtype='int64')

# Index as immutable array
print(ind[1])     # 3
print(ind[::2])  # Index([2, 5, 11])

# Index attributes (like NumPy arrays)
print(ind.size, ind.shape, ind.ndim, ind.dtype)
# 5 (5,) 1 int64

# Indices are IMMUTABLE
# ind[1] = 0  # This will raise TypeError!

# Index as ordered set
indA = pd.Index([1, 3, 5, 7, 9])
indB = pd.Index([2, 3, 5, 7, 11])
print(indA & indB)  # Intersection: [3, 5, 7]
print(indA | indB)  # Union: [1, 2, 3, 5, 7, 9, 11]
```

Key Takeaways: Pandas Objects

Series

- 1D labeled array = generalized NumPy array
- Also like a specialized dictionary
- Has both `values` (array) and `index` (labels)

DataFrame

- 2D labeled data structure = table with named columns
- Like a dictionary of Series (all sharing same index)
- Has `index` (rows), `columns`, and `values`
- Can contain heterogeneous types

Index

- Immutable array for row/column labels
- Supports set operations (union, intersection)
- Shared between Series/DataFrame for alignment

Series Indexing: Dictionary and Array Style

```python
import pandas as pd

data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])

# Dictionary-style indexing
print(data['b'])  # 0.5
print('a' in data)  # True

# Array-style slicing
print(data['a':'c'])  # Includes 'c'! (explicit index)
# a    0.25
# b    0.50
# c    0.75

print(data[0:2])  # Excludes index 2 (implicit index)
# a    0.25
# b    0.50

# Masking
print(data[(data > 0.3) & (data < 0.8)])
# Fancy indexing
print(data[['a', 'c']])
```

Confusion with Integer Indices

When Series has integer index, `data[1]` uses explicit index, but `data[1:3]` uses implicit. This can be confusing!

```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])

# loc: ALWAYS uses explicit index
print(data.loc[1])    # 'a'
print(data.loc[1:3])  # Indices 1, 3 (both included!)

# iloc: ALWAYS uses implicit Python-style index
print(data.iloc[1])    # 'b' (position 1)
print(data.iloc[1:3])  # Positions 1, 2 (excludes 3)
```

Best Practice

Always use `loc` and `iloc` explicitly! Makes code clearer and prevents bugs.

DataFrame Indexing: Columns Come First

```python
import pandas as pd

area = pd.Series({'California': 423967, 'Texas': 695662,
                  'New York': 141297, 'Florida': 170312})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                 'New York': 19651127, 'Florida': 19552860})
data = pd.DataFrame({'area':area, 'pop':pop})

# Access column (returns Series)
print(data['area'])  # Column 'area'

# Attribute-style access (if name doesn't conflict)
print(data.area)  # Same as data['area']

# Add new column
data['density'] = data['pop'] / data['area']

# Slicing accesses ROWS (different from columns!)
print(data['Florida':'New York'])  # Rows Florida to New York
```

[TOGETHER] DataFrame: loc and iloc

```python
import pandas as pd

# iloc: Python-style integer indexing
print(data.iloc[:3, :2])  # First 3 rows, first 2 columns
#                area       pop
# California  423967  38332521
# Florida     170312  19552860
# Illinois    149995  12882135

# loc: Label-based indexing
print(data.loc[:'Florida', :'pop'])  # Up to Florida, up to pop
#                area       pop
# California  423967  38332521
# Florida     170312  19552860

# Mixing both styles
print(data.loc[data.density > 100, ['pop', 'density']])
#                 pop     density
# Florida   19552860  114.806121
# New York  19651127  139.076746
```

Boolean Masking in DataFrames

```python
import pandas as pd

# Boolean mask on rows
high_density = data.density > 100
print(data[high_density])
#               area       pop     density
# Florida   170312  19552860  114.806121
# New York  141297  19651127  139.076746

# Combine with loc for specific columns
print(data.loc[high_density, ['pop', 'density']])

# Boolean operations
# Use & (AND), | (OR), ~ (NOT), not 'and', 'or', 'not'!
mask = (data['density'] > 50) & (data['density'] < 120)
print(data[mask])

# Fancy indexing: select specific rows and columns
print(data.loc[['California', 'Texas'], ['pop', 'area']])
```

Indexing Conventions: Summary

Series Indexing

- `data[key]`: Dictionary-style access by explicit index
- `data[i:j]`: Array-style slicing by implicit index
- `data.loc[key]`: Explicit indexing
- `data.iloc[i]`: Implicit integer indexing

DataFrame Indexing

- `data['col']`: Access column
- `data.iloc[i, j]`: Integer row/column indexing
- `data.loc[label, col]`: Label-based row/column indexing
- `data[mask]`: Boolean masking on rows

Key Rule

Columns are primary in DataFrames! `data['col']` gets column, not row.

Ufuncs: Index Preservation

```python
import pandas as pd
import numpy as np

rng = np.random.RandomState(42)
ser = pd.Series(rng.randint(0, 10, 4))
print(ser)
# 0    6
# 1    3
# 2    7
# 3    4

# NumPy ufuncs preserve index!
print(np.exp(ser))
# 0     403.428793
# 1      20.085537
# 2    1096.633158
# 3      54.598150

# Works with DataFrames too
df = pd.DataFrame(rng.randint(0, 10, (3, 4)),
                  columns=['A', 'B', 'C', 'D'])
print(np.sin(df * np.pi / 4))  # Preserves row/column labels!
```

[TOGETHER] Index Alignment in Operations

```python
import pandas as pd

# Top 3 states by area
area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
                  'California': 423967})
# Top 3 states by population
population = pd.Series({'California': 38332521, 'Texas': 26448193,
                        'New York': 19651127})

# Division aligns indices automatically!
density = population / area
print(density)
# Alaska              NaN  (no population data)
# California    90.413926
# New York            NaN  (no area data)
# Texas         38.018740

# Result contains UNION of indices
# Missing values filled with NaN
```

Index Alignment: Controlling Missing Values

```python
import pandas as pd

A = pd.Series([2, 4, 6], index=[0, 1, 2])
B = pd.Series([1, 3, 5], index=[1, 2, 3])

# Default: union of indices, fill with NaN
print(A + B)
# 0    NaN
# 1    5.0
# 2    9.0
# 3    NaN

# Use .add() method with fill_value
print(A.add(B, fill_value=0))
# 0    2.0  (2 + 0)
# 1    5.0  (4 + 1)
# 2    9.0  (6 + 3)
# 3    5.0  (0 + 5)
```

Python Operators and Pandas Methods

Python Operator	Pandas Method
`+`	`add()`
`-`	`sub()`, `subtract()`
`*`	`mul()`, `multiply()`
`/`	`truediv()`, `div()`, `divide()`
`//`	`floordiv()`
`%`	`mod()`
`**`	`pow()`

Why Use Methods?

Methods allow you to specify `fill_value` for missing data and control alignment behavior.

Operations Between DataFrame and Series

```python
import pandas as pd
import numpy as np

# Create DataFrame
A = np.array([[3, 8, 2, 4],
              [2, 6, 4, 8],
              [6, 1, 3, 8]])
df = pd.DataFrame(A, columns=list('QRST'))

# Subtract first row (broadcasts row-wise by default)
print(df - df.iloc[0])
#     Q  R  S  T
# 0  0  0  0  0
# 1 -1 -2  2  4
# 2  3 -7  1  4

# For column-wise, specify axis
print(df.subtract(df['R'], axis=0))
#     Q  R  S  T
# 0 -5  0 -6 -4
# 1 -4  0 -2  2
# 2  5  0  2  7
```

Key Advantages of Pandas Operations

Automatic Index Alignment

- Operations automatically align on matching indices
- No need to manually match row/column labels
- Prevents errors from misaligned data

Index Preservation

- Labels are maintained through operations
- Results keep meaningful row/column names
- Data context is never lost

Compare to NumPy

NumPy arrays lose label information. Pandas keeps everything organized and labeled!

Missing Data: A Reality of Real-World Datasets

The Problem

Real-world data is rarely clean! Missing values are common in:

- Sensor data (equipment failures)
- Survey responses (unanswered questions)
- Database joins (unmatched records)
- Data entry errors

Pandas Approach: Two Sentinels

- **None**: Python object for missing data
- **NaN**: IEEE floating-point "Not a Number"
- Pandas treats them (nearly) interchangeably

None vs NaN

```python
import numpy as np
import pandas as pd

# None: Python object (slow, object dtype)
vals1 = np.array([1, None, 3, 4])
print(vals1.dtype)  # object
# Operations are slow! Uses Python loops, not C

# NaN: Floating-point (fast, native type)
vals2 = np.array([1, np.nan, 3, 4])
print(vals2.dtype)  # float64
# Fast! Uses compiled C code

# NaN is "contagious"
print(1 + np.nan)    # nan
print(0 * np.nan)    # nan

# Pandas converts between them automatically
print(pd.Series([1, np.nan, 2, None]))
# 0    1.0
# 1    NaN
# 2    2.0
# 3    NaN
```

Detecting Missing Data

```
import pandas as pd
import numpy as np

data = pd.Series([1, np.nan, 'hello', None])

# isnull(): returns Boolean mask
print(data.isnull())
# 0    False
# 1     True
# 2    False
# 3     True

# notnull(): opposite of isnull()
print(data.notnull())
# 0     True
# 1    False
# 2     True
# 3    False

# Use Boolean indexing to filter
print(data[data.notnull()])
# 0        1
# 2    hello
```

[TOGETHER] Dropping Missing Data

```python
import pandas as pd
import numpy as np

# Series: dropna() removes NaN values
data = pd.Series([1, np.nan, 2, None, 3])
print(data.dropna())
# 0    1.0
# 2    2.0
# 4    3.0

# DataFrame: more options!
df = pd.DataFrame([[1,     np.nan, 2],
                   [2,     3,      5],
                   [np.nan, 4,     6]])

# Drop rows with ANY NaN
print(df.dropna())  # Only row 1 remains

# Drop columns with ANY NaN
print(df.dropna(axis='columns'))  # Only column 2 remains

# Drop only if ALL values are NaN
print(df.dropna(how='all'))
```

Filling Missing Data

```python
import pandas as pd
import numpy as np

data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))

# Fill with constant value
print(data.fillna(0))
# a    1.0
# b    0.0  <- filled
# c    2.0
# d    0.0  <- filled
# e    3.0

# Forward fill: propagate previous value
print(data.fillna(method='ffill'))
# b    1.0  <- filled with 'a'
# d    2.0  <- filled with 'c'

# Back fill: propagate next value
print(data.fillna(method='bfill'))
# b    2.0  <- filled with 'c'
# d    3.0  <- filled with 'e'
```

DataFrame: Filling with axis Parameter

```python
import pandas as pd
import numpy as np

df = pd.DataFrame([[1,      np.nan, 2],
                   [2,      3,      5],
                   [np.nan, 4,      6]])

# Forward fill along columns (row-wise)
print(df.fillna(method='ffill', axis=1))
#      0    1    2
# 0  1.0  1.0  2.0
# 1  2.0  3.0  5.0
# 2  NaN  4.0  6.0  <- No previous value in row

# Forward fill along rows (column-wise)
print(df.fillna(method='ffill', axis=0))
#      0    1  2
# 0  1.0  NaN  2
# 1  2.0  3.0  5
# 2  2.0  4.0  6  <- Filled from row 1
```

Missing Data: Summary

Detection

- `isnull()`: Boolean mask of missing values
- `notnull()`: Boolean mask of valid values
- Use for filtering or counting: `data[data.notnull()]`

Removal

- `dropna()`: Remove NaN values
- Series: drops NaN entries
- DataFrame: drops rows or columns (specify `axis`)
- Control with `how='all'` or `thresh=n`

Filling

- `fillna(value)`: Replace with constant
- `fillna(method='ffill')`: Forward fill
- `fillna(method='bfill')`: Backward fill
- Specify `axis` for direction in DataFrame

MultiIndex: Higher-Dimensional Data

The Challenge

Often need to work with data indexed by more than one or two keys:

- Data by (state, year)
- Measurements by (subject, visit, test)
- Stock prices by (date, ticker)

Solution: MultiIndex

- Multiple index levels within a single index
- Store higher-dimensional data in 1D Series or 2D DataFrame
- More flexible than Panel (3D) or Panel4D (4D)
- Efficient and intuitive for complex data

Creating a MultiIndex Series

```python
import pandas as pd

# State population data for multiple years
index = pd.MultiIndex.from_tuples([
    ('California', 2000), ('California', 2010),
    ('New York', 2000), ('New York', 2010),
    ('Texas', 2000), ('Texas', 2010)
])
populations = [33871648, 37253956, 18976457,
               19378102, 20851820, 25145561]
pop = pd.Series(populations, index=index)
print(pop)
# (California, 2000)    33871648
# (California, 2010)    37253956
# (New York, 2000)      18976457
# (New York, 2010)      19378102
# (Texas, 2000)         20851820
# (Texas, 2010)         25145561

# Name the index levels
pop.index.names = ['state', 'year']
```

[TOGETHER] Indexing with MultiIndex

```python
import pandas as pd

# Access all data for year 2010
print(pop[:, 2010])
# state
# California     37253956
# New York       19378102
# Texas          25145561

# Access all data for California
print(pop['California'])
# year
# 2000    33871648
# 2010    37253956

# Access specific element
print(pop['California', 2010])  # 37253956

# Slicing works too!
print(pop['California':'New York'])
```

```python
import pandas as pd

# Add another column to our MultiIndex data
pop_df = pd.DataFrame({
    'total': pop,
    'under18': [9267089, 9284094, 4687374,
                4318033, 5906301, 6879014]
})
print(pop_df)
#                      total   under18
# state      year
# California 2000   33871648   9267089
#            2010   37253956   9284094
# New York   2000   18976457   4687374
#            2010   19378102   4318033
# Texas      2000   20851820   5906301
#            2010   25145561   6879014

# Calculate fraction under 18
f_u18 = pop_df['under18'] / pop_df['total']
print(f_u18.unstack())  # Convert to regular DataFrame
```

Creating MultiIndex: Multiple Methods

```python
import pandas as pd

# From arrays
pd.MultiIndex.from_arrays([
    ['a', 'a', 'b', 'b'],
    [1, 2, 1, 2]
])

# From tuples
pd.MultiIndex.from_tuples([
    ('a', 1), ('a', 2), ('b', 1), ('b', 2)
])

# From product (Cartesian product)
pd.MultiIndex.from_product([
    ['a', 'b'],  # Level 0
    [1, 2]       # Level 1
])

# All create the same MultiIndex!
# MultiIndex(levels=[['a', 'b'], [1, 2]],
#            codes=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

Stack and Unstack

```
import pandas as pd

# Unstack: convert MultiIndex to regular DataFrame
print(pop.unstack())
# year          2000      2010
# state
# California  33871648  37253956
# New York    18976457  19378102
# Texas       20851820  25145561

# Stack: convert back to MultiIndex Series
print(pop.unstack().stack())
# state       year
# California  2000    33871648
#             2010    37253956
# New York    2000    18976457
#             2010    19378102
# Texas       2000    20851820
#             2010    25145561

# Can specify level to unstack
print(pop.unstack(level=0))  # Unstack states instead
```

MultiIndex: Key Concepts

Why Use MultiIndex?

- Represent higher-dimensional data compactly
- More flexible than Panel/Panel4D
- Efficient for sparse multi-dimensional data
- Intuitive slicing and indexing

Key Operations

- `MultiIndex.from_tuples()`, `from_arrays()`, `from_product()`
- `pop['California']`: Partial indexing
- `pop[:, 2010]`: Slicing on second level
- `unstack()`: MultiIndex → DataFrame
- `stack()`: DataFrame → MultiIndex

Important

MultiIndex must be sorted for partial slicing! Use `sort_index()` if needed.

Why Combine Data?

Common Scenarios

- Combining data from multiple sensors/sources
- Appending new observations to existing dataset
- Merging different measurements of same subjects
- Concatenating time series data from different periods

Pandas Tools

- **pd.concat()**: General concatenation
- **df.append()**: Quick row append (less efficient)
- **pd.merge()** and **pd.join()**: Database-style joins (next lecture!)

Simple Concatenation with pd.concat()

```python
import pandas as pd

# Concatenate Series
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
print(pd.concat([ser1, ser2]))
# 1    A
# 2    B
# 3    C
# 4    D
# 5    E
# 6    F

# Concatenate DataFrames (row-wise by default)
df1 = pd.DataFrame({'A': ['A1', 'A2'], 'B': ['B1', 'B2']})
df2 = pd.DataFrame({'A': ['A3', 'A4'], 'B': ['B3', 'B4']})
print(pd.concat([df1, df2]))
#      A   B
# 0   A1  B1
# 1   A2  B2
# 0   A3  B3  <- Index repeated!
# 1   A4  B4
```

Concatenation Along Different Axes

```python
import pandas as pd

df1 = pd.DataFrame({'A': ['A0', 'A1'],
                    'B': ['B0', 'B1']})
df2 = pd.DataFrame({'C': ['C0', 'C1'],
                    'D': ['D0', 'D1']})

# Concatenate along columns (axis=1)
print(pd.concat([df1, df2], axis=1))
#     A   B   C   D
# 0  A0  B0  C0  D0
# 1  A1  B1  C1  D1

# Concatenate along rows (axis=0, default)
print(pd.concat([df1, df2], axis=0))
#       A    B    C    D
# 0    A0   B0  NaN  NaN
# 1    A1   B1  NaN  NaN
# 0  NaN  NaN   C0   D0
# 1  NaN  NaN   C1   D1
```

[TOGETHER] Handling Duplicate Indices

```python
import pandas as pd

df1 = pd.DataFrame({'A': ['A0', 'A1'], 'B': ['B0', 'B1']})
df2 = pd.DataFrame({'A': ['A2', 'A3'], 'B': ['B2', 'B3']})

# Option 1: Ignore old index, create new one
print(pd.concat([df1, df2], ignore_index=True))
#      A    B
# 0   A0   B0
# 1   A1   B1
# 2   A2   B2
# 3   A3   B3

# Option 2: Add hierarchical index with keys
print(pd.concat([df1, df2], keys=['x', 'y']))
#        A    B
# x 0   A0   B0
#   1   A1   B1
# y 0   A2   B2
#   1   A3   B3

# Option 3: Verify integrity (raise error if duplicates)
# pd.concat([df1, df2], verify_integrity=True)  # Raises error!
```

Concatenation with Joins

```python
import pandas as pd

df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                    'B': ['B0', 'B1', 'B2']})
df2 = pd.DataFrame({'B': ['B3', 'B4', 'B5'],
                    'C': ['C3', 'C4', 'C5']})

# Default: outer join (union of columns)
print(pd.concat([df1, df2]))
#       A    B    C
# 0    A0   B0  NaN
# 1    A1   B1  NaN
# 2    A2   B2  NaN
# 0   NaN   B3   C3
# 1   NaN   B4   C4
# 2   NaN   B5   C5

# Inner join: intersection of columns
print(pd.concat([df1, df2], join='inner'))
#      B
# 0   B0
# 1   B1
# 2   B2
# 0   B3
# 1   B4
# 2   B5
```

The append() Method

```python
import pandas as pd

df1 = pd.DataFrame({'A': ['A1', 'A2'], 'B': ['B1', 'B2']})
df2 = pd.DataFrame({'A': ['A3', 'A4'], 'B': ['B3', 'B4']})

# append() is shorthand for pd.concat([df1, df2])
print(df1.append(df2))
#      A   B
# 0   A1  B1
# 1   A2  B2
# 0   A3  B3
# 1   A4  B4
```

Important Note

- `append()` does NOT modify original DataFrame
- It creates a new DataFrame (like all Pandas operations)
- Less efficient than `pd.concat()` for multiple appends
- For many appends: collect in list, then use `pd.concat()` once

Combining Data: Summary

pd.concat()

- General-purpose concatenation
- Works on Series and DataFrame
- `axis=0`: concatenate rows (default)
- `axis=1`: concatenate columns
- `join='outer'` (default) or `join='inner'`
- `ignore_index=True`: reset index
- `keys=['x', 'y']`: add hierarchical index

append()

- Shorthand for row concatenation
- Returns new DataFrame (doesn't modify original)
- Less efficient for multiple operations

Coming Soon

`pd.merge()` and `pd.join()` for database-style joins (inner, outer, left, right)

Key Concepts Mastered: NumPy

Array Fundamentals

- Creating and manipulating arrays
- Indexing, slicing, reshaping
- Data types and memory
- Broadcasting rules

Operations

- Vectorized arithmetic
- Universal functions (ufuncs)
- Mathematical operations
- Boolean operations

Data Analysis

- Aggregations and statistics
- Boolean masking
- Fancy indexing
- Sorting and searching

Why NumPy?

- 10-100x faster than Python lists
- Foundation for all scientific Python
- Memory efficient
- Integrates with C/Fortran code

Key Concepts Mastered: Pandas

Core Structures

- Series: 1D labeled arrays
- DataFrame: 2D labeled tables
- Index: Row and column labels
- MultiIndex: Hierarchical indices

Data Selection

- loc/iloc indexing
- Boolean masking
- Fancy indexing
- Slicing and filtering

Operations

- Index alignment
- Missing data handling
- Concatenation and append
- Element-wise operations

Why Pandas?

- Named rows and columns
- Built-in missing data support
- Built on NumPy (fast!)
- Industry standard for data analysis

NumPy Ecosystem: What's Built on Top

Scientific Computing Stack

- **SciPy**: Scientific algorithms (optimization, integration, linear algebra)
- **Pandas**: Structured data analysis (DataFrames) - NEXT WEEK!
- **Matplotlib**: Visualization and plotting
- **scikit-learn**: Machine learning
- **TensorFlow/PyTorch**: Deep learning

Domain-Specific Libraries

- **Astropy**: Astronomy and astrophysics
- **Biopython**: Bioinformatics
- **QuantLib**: Financial modeling
- All built on NumPy's fast array operations!

Next Week: More Pandas & Data Visualization

Advanced Pandas Topics

Building on this week's foundation:

- Merge and Join: Database-style data combination
- GroupBy: Split-apply-combine operations
- Pivot Tables: Excel-style data summarization
- Time Series: Working with dates and times
- Reading/Writing Data: CSV, Excel, SQL, JSON
- String Operations: Text processing in Pandas

Introduction to Matplotlib

- Basic plotting with matplotlib
- Line plots, scatter plots, histograms
- Customizing plots (labels, legends, styles)
- Integration with Pandas DataFrames
- Creating publication-quality figures

Thank You!

Questions?

Dr. Eyuphan Koc
eyuphan.koc@bogazici.edu.tr