IMEN574/IDSC574 Programing for Data Science

8. **Numpy & Pandas**

Spring 2026

Instructor: **Sangwoo Mo**

What We'll Cover Today

You know basic Python. Now it's time to master the tools that make data science possible.

NumPy Fundamentals	Arrays, vectorization, broadcasting — why loops are slow	~10 min
NumPy Operations	Indexing, slicing, reshaping, and linear algebra	~15 min
NumPy Performance	Broadcasting rules, memory layout, and vectorized thinking	~10 min
Pandas Core	DataFrame, Series, indexing with loc/iloc, data loading	~15 min
Pandas Data Wrangling	Cleaning, missing values, GroupBy, merge/join	~25 min
Pandas Patterns	Method chaining, apply, performance tips, best practices	~15 min

1

NumPy Fundamentals

The Foundation of Scientific Computing in Python



Why NumPy? — Speed Matters

Python lists are flexible but slow for numerical computation. NumPy fixes this.

```python
# Python list: slow loop
data = list(range(1_000_000))
result = []
for x in data:
    result.append(x * 2)
# ~150ms on typical machine
```

```python
# NumPy array: vectorized
import numpy as np
data = np.arange(1_000_000)
result = data * 2
# ~1ms — 150x faster!
```

Python loop: ~150ms (interpreted, type checking each element)

NumPy: ~1ms (compiled C code, no type checking)

Why is NumPy so fast?

Contiguous memory layout (no pointer chasing) • **Fixed data types** (no type checking per element) • **SIMD/vectorized C code** (hardware-level parallelism)

Creating NumPy Arrays

```python
import numpy as np
# From Python list
a = np.array([1, 2, 3, 4, 5])
b = np.array([[1, 2], [3, 4]])  # 2D
# Common constructors
zeros = np.zeros((3, 4))       # 3x4 of 0s
ones  = np.ones((2, 3))        # 2x3 of 1s
eye   = np.eye(4)              # 4x4 identity
empty = np.empty((5, 5))       # uninitialized
# Sequences
r = np.arange(0, 10, 0.5)     # like range()
l = np.linspace(0, 1, 100)    # 100 even points
# Random
rng = np.random.default_rng(42)
X = rng.standard_normal((100, 5))  # 100x5
```

Key Attributes

`a.shape`	(5,) — dimensions
`a.ndim`	1 — number of axes
`a.dtype`	int64 — data type
`a.size`	5 — total elements
`a.itemsize`	8 — bytes per element

dtype Matters!

NumPy arrays have a single, fixed dtype.

`Common: float64, float32, int64, int32, bool`

Use float32 for ML to save memory (half the size of float64).

Indexing & Slicing — Accessing Data

```
arr = np.arange(10)  # [0,1,2,...,9]
# Basic slicing (same as Python lists)
arr[3]          # 3
arr[2:5]        # [2, 3, 4]
arr[::2]        # [0, 2, 4, 6, 8]  every 2nd
arr[-3:]        # [7, 8, 9]  last 3
# 2D indexing
mat = np.arange(12).reshape(3, 4)
mat[1, 2]       # row 1, col 2 → 6
mat[0, :]       # first row → [0,1,2,3]
mat[:, -1]      # last column → [3,7,11]
```

```
# Fancy indexing (NumPy special!)
arr[[1, 3, 5]]          # pick by indices
# Boolean indexing (most important!)
mask = arr > 5
arr[mask]               # [6, 7, 8, 9]
arr[arr > 5]            # same, shorthand
# Combine conditions with & | ~
arr[(arr > 2) & (arr < 7)]  # [3,4,5,6]
arr[~(arr == 5)]        # all except 5
# Boolean indexing on 2D
mat[mat > 5]            # flat array of matches
```



Boolean indexing is the foundation of data filtering in both NumPy and Pandas!

```
# CRITICAL: slicing returns a VIEW, not a copy!
view = arr[2:5]
view[0] = 999          # This modifies arr too!   arr → [0,1,999,3,4,...]
copy = arr[2:5].copy() # Use .copy() for independent copy
```

Reshaping & Dimension Manipulation

```
arr = np.arange(12)
# reshape: change dimensions (total elements must match)
mat = arr.reshape(3, 4)     # 12 → 3x4
mat = arr.reshape(2, 2, 3)  # 12 → 2x2x3
mat = arr.reshape(4, -1)    # -1 means 'figure it out' → 4x3
# Flatten back to 1D
flat = mat.ravel()          # Returns view (fast)
flat = mat.flatten()        # Returns copy (safe)
```

```
# Transpose
X = np.random.randn(100, 5)
X.T              # shape: (5, 100)
X.T @ X          # 5x5 covariance-like matrix
# Add dimension
col = arr.reshape(-1, 1)    # → (12,1) column
row = arr[np.newaxis, :]    # → (1,12) row
```

```
# Stack arrays
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
np.vstack([a, b])  # [[1,2,3],[4,5,6]]
np.hstack([a, b])  # [1,2,3,4,5,6]
np.column_stack([a, b])  # [[1,4],[2,5],[3,6]]
np.concatenate([a, b])   # [1,2,3,4,5,6]
```

Broadcasting — NumPy's Superpower

Broadcasting lets NumPy operate on arrays of different shapes without copying data.

```
# Scalar broadcast
arr = np.array([1, 2, 3, 4])
arr * 3     # [3, 6, 9, 12]
# Vector + matrix broadcast
X = np.ones((3, 4))   # shape (3,4)
bias = np.array([1, 2, 3, 4])  # shape (4,)
X + bias    # bias added to each row!
```

```
# Real-world: standardize columns
X = np.random.randn(100, 5)
mean = X.mean(axis=0)  # shape (5,)
std  = X.std(axis=0)   # shape (5,)
X_scaled = (X - mean) / std
# Broadcasts (5,) across (100,5)!
```

Broadcasting Rules (compare from trailing dimensions)

Shape A	Shape B	Result	Why
(3, 4)	(4,)	(3, 4)	4 matches 4, row broadcast
(3, 4)	(3, 1)	(3, 4)	3 matches, 1 stretches to 4
(3, 4)	(3,)	ERROR	3 ≠ 4 and neither is 1
(5, 1, 3)	(1, 4, 3)	(5, 4, 3)	1s stretch to match

Vectorized Operations & Universal Functions

```
X = np.random.randn(1000, 10)
# Element-wise operations (ufuncs)
np.sqrt(np.abs(X))    # sqrt of absolute values
np.exp(X)             # e^x for each element
np.log(X[X > 0])      # log of positive values
np.clip(X, -3, 3)     # clamp to [-3, 3]
# Comparison (returns boolean array)
mask = X > 0           # True/False for each element
np.where(X > 0, X, 0) # ReLU activation!
```

```
# Aggregation along axes
X.mean()         # global mean (scalar)
X.mean(axis=0)   # column means (shape: 10,)
X.mean(axis=1)   # row means (shape: 1000,)
# Other aggregations
X.sum(axis=0)    # column sums
X.std(axis=0)    # column stds
X.min(axis=1)    # row minimums
X.argmax(axis=1) # index of max per row
np.percentile(X, 75, axis=0)  # Q3 per col
```



axis=0 means 'across rows' (collapse rows → column result). axis=1 means 'across columns' (collapse cols → row result).

```
# Practical: compute R² score without loops
y_true = np.random.randn(100)
y_pred = y_true + np.random.randn(100) * 0.1
ss_res = np.sum((y_true - y_pred) ** 2)
ss_tot = np.sum((y_true - y_true.mean()) ** 2)
r2 = 1 - ss_res / ss_tot   # All vectorized, no loops!
```

Linear Algebra with NumPy

```
# Matrix operations
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
A @ B           # Matrix multiplication
np.dot(A, B)    # Same thing
A * B           # Element-wise (NOT matrix mul!)
A.T             # Transpose
np.linalg.inv(A)   # Inverse
np.linalg.det(A)   # Determinant
np.trace(A)         # Trace (sum of diagonal)
```

```
# Decompositions
eigenvals, eigenvecs = np.linalg.eig(A)
U, S, Vt = np.linalg.svd(A)  # SVD
# Solve linear system: Ax = b
b = np.array([1, 2])
x = np.linalg.solve(A, b)  # Much better than
inv!
# Norms
np.linalg.norm(A)          # Frobenius norm
np.linalg.norm(A, axis=1)  # Row L2 norms
np.linalg.norm(A, ord=1)   # L1 norm
```



Use np.linalg.solve(A, b) instead of np.linalg.inv(A) @ b — it's faster and more numerically stable.

```
# Practical: OLS regression in one line!
X_b = np.c_[np.ones((100, 1)), X]                  # Add bias column
beta = np.linalg.lstsq(X_b, y, rcond=None)[0]      # Least squares solution
```

2

NumPy Advanced Patterns

Performance, Memory, and Vectorized Thinking



Think in Vectors, Not Loops

The key mental shift: replace explicit loops with array operations.

```
# BAD: Python loop
distances = []
for i in range(len(X)):
    d = np.sqrt(np.sum(
        (X[i] - center) ** 2))
    distances.append(d)
# Slow! 100ms for 100K points
```

```
# GOOD: Vectorized
distances = np.sqrt(
    np.sum((X - center) ** 2, axis=1)
)
# Fast! <1ms for 100K points
```

```
# BAD: Conditional loop
result = np.zeros_like(X)
for i in range(len(X)):
    for j in range(X.shape[1]):
        if X[i,j] > 0:
            result[i,j] = X[i,j]
# Nested loop... very slow
```

```
# GOOD: Vectorized
result = np.where(X > 0, X, 0)
# Or even simpler:
result = np.maximum(X, 0)  # ReLU!
# One line, 100x+ faster
```

NumPy Practical Patterns for DS

```
# Pattern 1: Feature engineering
X = np.random.randn(1000, 5)
# Polynomial features
X_poly = np.column_stack([
    X,
    X ** 2,            # Squared features
    X[:, 0:1] * X[:, 1:2]  # Interaction
])
# One-hot encoding with NumPy
labels = np.array([0, 2, 1, 0, 2])
one_hot = np.eye(3)[labels]  # Clever trick!
```

```
# Pattern 2: Train/test split
n = len(X)
indices = np.arange(n)
np.random.shuffle(indices)
split = int(0.8 * n)
train_idx = indices[:split]
test_idx  = indices[split:]
X_train, X_test = X[train_idx], X[test_idx]
# Pattern 3: Mini-batch iteration
for start in range(0, n, batch_size):
    batch = X[start:start+batch_size]
    # Process batch...
```

```
# Pattern 4: Softmax function — fully vectorized
def softmax(logits):
    """Numerically stable softmax."""
    shifted = logits - logits.max(axis=1, keepdims=True)  # Stability trick
    exp_vals = np.exp(shifted)
    return exp_vals / exp_vals.sum(axis=1, keepdims=True)  # Broadcasting!
probs = softmax(np.random.randn(32, 10))  # 32 samples, 10 classes
probs.sum(axis=1)  # All 1.0 — each row sums to 1
```

3

Pandas Core

DataFrames, Series, and Data Loading



What Is Pandas?

Pandas provides labeled, tabular data structures built on top of NumPy.

Series

1D labeled array — like a column

Has an index (labels) + values (NumPy array)

Every DataFrame column is a Series

DataFrame

2D labeled table — rows × columns

Has row index + column names + values

The primary Pandas data structure

```
import pandas as pd
s = pd.Series([10, 20, 30],
              index=['a', 'b', 'c'])
s['b']      # 20  (label-based access)
s.values    # np.array([10, 20, 30])
```

```
df = pd.DataFrame({
    'name': ['Alice', 'Bob', 'Carol'],
    'age':  [25, 30, 35],
    'score': [0.85, 0.92, 0.88]
})
df['age']   # Returns a Series
```

Loading Data into Pandas

```
# CSV (most common)
df = pd.read_csv('data.csv')                    # Basic
df = pd.read_csv('data.csv', usecols=['name', 'age'], nrows=1000)  # Selective
df = pd.read_csv('data.csv', parse_dates=['date'], dtype={'id': str})  # Type control
# Other formats
df = pd.read_excel('data.xlsx', sheet_name='Sheet1')
df = pd.read_parquet('data.parquet')   # Fast & compact — use for large datasets!
df = pd.read_sql('SELECT * FROM users', connection)
```

```
# First things to do after loading
df.shape          # (rows, columns)
df.info()         # dtypes, non-null counts
df.describe()     # statistics for numerics
df.head()         # first 5 rows
df.dtypes         # column data types
df.columns        # column names
df.isnull().sum() # missing values per col
```

```
# Saving data
df.to_csv('output.csv', index=False)
df.to_parquet('output.parquet')
df.to_excel('output.xlsx', index=False)
# Tip: always use index=False unless
# you intentionally want to save the index!
# Otherwise you get an extra 'Unnamed: 0' col
```

Indexing: loc vs iloc

The two most important indexing methods — know the difference!

df.loc[ ] — Label-based

```
# Select by label names
df.loc[0, 'name']        # Single value
df.loc[0:2, 'name']      # INCLUSIVE range!
df.loc[:, 'age':'score']  # Column range
# Boolean selection with loc
df.loc[df['age'] > 25, 'name']
# Set values
df.loc[df['age'] > 30, 'senior'] = True
```

df.iloc[ ] — Position-based

```
# Select by integer position
df.iloc[0, 1]            # row 0, col 1
df.iloc[0:2, :]          # EXCLUSIVE range!
df.iloc[:, -1]           # Last column
# Slice rows and columns
df.iloc[10:20, 0:3]
# Set values
df.iloc[0, 2] = 0.95
```

Key difference: loc[0:2] returns 3 rows (inclusive). iloc[0:2] returns 2 rows (exclusive, like Python).

```
# Most common: boolean indexing  →  df.loc[df['col'] > value]  or  df[df['col'] > value]
```

Boolean Filtering — The Pandas Way

```
# Single condition
adults = df[df['age'] >= 18]
# Multiple conditions (use & | ~ with parentheses!)
young_high = df[(df['age'] < 30) & (df['score'] > 0.9)]
excluded   = df[~df['name'].isin(['Alice', 'Bob'])]
# String methods
smith = df[df['name'].str.contains('Smith', case=False)]
starts_a = df[df['name'].str.startswith('A')]
```

```
# query() — cleaner for complex filters
result = df.query('age > 25 and score > 0.8')
result = df.query('name == @target_name')
# @ prefix references Python variables
# Especially nice with method chaining:
# df.query(...).groupby(...).agg(...)
```

```
# isin() — filter by list of values
selected = df[df['city'].isin(
    ['Seoul', 'Busan', 'Daegu'])]
# between() — range filter
mid_age = df[df['age'].between(25, 35)]
# nlargest / nsmallest
top5 = df.nlargest(5, 'score')
```

Creating & Transforming Columns

```python
# Direct assignment
df['bmi'] = df['weight'] / (df['height']**2)
# Conditional column
df['grade'] = np.where(
    df['score'] > 0.9, 'A',
    np.where(df['score'] > 0.7, 'B', 'C')
)
# From apply (flexible but slower)
df['name_len'] = df['name'].apply(len)
```

```python
# assign() — returns new DataFrame (chainable)
df = df.assign(
    bmi = lambda x: x['weight']/x['height']**2,
    is_obese = lambda x: x['bmi'] > 30,
    age_group = lambda x: pd.cut(
        x['age'], bins=[0,30,50,100],
        labels=['young','mid','senior']
    )
)
```

```python
# Type conversions
df['id'] = df['id'].astype(str)                  # To string
df['price'] = pd.to_numeric(df['price'], errors='coerce')  # To number (NaN if fails)
df['date'] = pd.to_datetime(df['date'])          # To datetime
# Binning continuous values
df['age_bin'] = pd.cut(df['age'], bins=5)                  # Equal-width bins
df['score_q'] = pd.qcut(df['score'], q=4, labels=['Q1','Q2','Q3','Q4'])  # Quantile bins
```

Handling Missing Data

Missing data is inevitable in real-world datasets. Pandas represents it as NaN.

```
# Detection
df.isnull()            # Boolean mask
df.isnull().sum()      # Count per column
df.isnull().mean()     # Fraction per column
# Any row with missing values?
df[df.isnull().any(axis=1)]
# Which columns have missing?
df.columns[df.isnull().any()]
```

```
# Removal
df.dropna()                # Drop any row w/ NaN
df.dropna(subset=['age'])  # Only if age is NaN
df.dropna(thresh=3)        # Keep if >=3 non-NaN
# Filling
df.fillna(0)                # Fill with constant
df['age'].fillna(df['age'].median())  # Median
df.fillna(method='ffill')   # Forward fill
```

```
# Advanced filling strategies
# Group-specific filling (fill with group median)
df['age'] = df.groupby('city')['age'].transform(
    lambda x: x.fillna(x.median())
)
# Interpolation (great for time series)
df['temperature'] = df['temperature'].interpolate(method='linear')
```



Always understand WHY data is missing before choosing a strategy. MCAR, MAR, MNAR require different approaches.

4

Pandas Data Wrangling

GroupBy, Merge, Reshape, and String/DateTime Ops



GroupBy — Split-Apply-Combine

GroupBy is Pandas' most powerful aggregation tool. Think: SQL GROUP BY.

1. Split

Divide data into groups by key column

2. Apply

Apply function to each group independently

3. Combine

Merge results back into a single structure

```
# Basic groupby
df.groupby('city')['salary'].mean()
# Multiple aggregations
df.groupby('city')['salary'].agg(
    ['mean', 'median', 'std', 'count']
)
# Group by multiple columns
df.groupby(['city', 'dept'])['salary'].mean()
```

```
# Named aggregation (clean output!)
result = df.groupby('city').agg(
    avg_salary = ('salary', 'mean'),
    max_salary = ('salary', 'max'),
    headcount  = ('name', 'count'),
    avg_age    = ('age', 'mean'),
)
# Reset index to make groupby key a column
result = result.reset_index()
```



Named aggregation with .agg(name=('col', 'func')) is the cleanest way — use it!

GroupBy — Advanced Patterns

```
# transform: returns same-shaped result
# (broadcasts back to original index)
df['salary_z'] = df.groupby('city')['salary']\
    .transform(lambda x: (x - x.mean()) / x.std())
# Group-wise percentile rank
df['salary_rank'] = df.groupby('dept')['salary']\
    .transform('rank', pct=True)
# Filter: keep groups that satisfy condition
big_depts = df.groupby('dept').filter(
    lambda g: len(g) >= 10)
```

```
# apply: most flexible (but slower)
def top_3(group):
    return group.nlargest(3, 'salary')
top_per_city = df.groupby('city').apply(top_3)
# Iterate groups (debugging only!)
for name, group in df.groupby('city'):
    print(f'{name}: {len(group)} rows')
# Size vs Count
df.groupby('city').size()   # Includes NaN
df.groupby('city').count()  # Excludes NaN
```

```
# Real-world: customer RFM analysis
rfm = orders.groupby('customer_id').agg(
    recency    = ('order_date', lambda x: (pd.Timestamp.now() - x.max()).days),
    frequency  = ('order_id', 'nunique'),
    monetary   = ('amount', 'sum'),
)
rfm['r_score'] = pd.qcut(rfm['recency'], 4, labels=[4,3,2,1])   # Lower is better
rfm['f_score'] = pd.qcut(rfm['frequency'], 4, labels=[1,2,3,4])
```

Merging DataFrames — SQL-style Joins

Combining datasets is essential. Pandas merge works like SQL JOIN.

```
# Basic merge (inner join by default)
merged = pd.merge(orders, customers,
                  on='customer_id')
# Different column names
merged = pd.merge(df1, df2,
    left_on='cust_id', right_on='id')
```

```
# Specify join type
pd.merge(A, B, on='key', how='left')
pd.merge(A, B, on='key', how='right')
pd.merge(A, B, on='key', how='outer')
pd.merge(A, B, on='key', how='inner')
# Multi-key merge
pd.merge(A, B, on=['key1', 'key2'])
```

inner
Only matching rows in both

left
All left rows + matching right

right
All right rows + matching left

outer
All rows from both tables

```
# Practical: check for duplicates after merge!
merged = pd.merge(orders, products, on='product_id', how='left')
assert len(merged) == len(orders), 'Unexpected row multiplication!'
# If product_id is not unique in products, you'll get extra rows!
```

Concatenation — Stacking DataFrames

```python
# Stack vertically (add rows)
combined = pd.concat([df1, df2, df3])
# Reset index after concat
combined = pd.concat([df1, df2],
                     ignore_index=True)
# Add source label
combined = pd.concat(
    [df1, df2],
    keys=['train', 'test'],  # MultiIndex
)
```

```python
# Stack horizontally (add columns)
full = pd.concat([df_features, df_labels],
                 axis=1)
# Common pattern: load multiple files
import glob
files = glob.glob('data/sales_*.csv')
dfs = [pd.read_csv(f) for f in files]
all_sales = pd.concat(dfs,
                      ignore_index=True)
```

merge() vs concat() — When to Use Which?

Function	When to Use	Analogy
`pd.merge()`	Combine by matching key columns	Like SQL JOIN — horizontal by key
`pd.concat()`	Stack DataFrames together	Like UNION ALL — vertical or horizontal by position
`df.join()`	Merge on index (shortcut for merge)	Convenience method, same as merge on index

Pivot Tables — Reshape for Analysis

```
# Pivot table (like Excel PivotTable)
pivot = df.pivot_table(
    values='revenue',
    index='region',
    columns='quarter',
    aggfunc='sum',
    margins=True,      # Add row/col totals
    fill_value=0       # Fill NaN with 0
)
# Multiple aggregations
pivot = df.pivot_table(
    values='revenue',
    index='region',
    aggfunc=['sum', 'mean', 'count'])
```

```
# melt: reverse of pivot (wide → long)
wide = pd.DataFrame({
    'name': ['Alice', 'Bob'],
    'math': [90, 85],
    'english': [88, 92]
})
long = pd.melt(wide,
    id_vars='name',
    value_vars=['math', 'english'],
    var_name='subject',
    value_name='score'
)
# name  subject  score
# Alice math     90
```

```
# crosstab: frequency/contingency table
ct = pd.crosstab(df['gender'], df['purchased'],
                 normalize='index')       # Row percentages
# stack/unstack: reshape MultiIndex
stacked = pivot.stack()                    # Columns → row level
unstacked = stacked.unstack(level=0)       # Row level → columns
```

String Operations — Working with Text

```
# .str accessor — vectorized string ops
df['name'].str.lower()
df['name'].str.upper()
df['name'].str.strip()       # Remove whitespace
df['name'].str.len()         # Length of each string
# Splitting
df['first_name'] = df['name'].str.split().str[0]
df[['first', 'last']] = df['name'].str.split(
    ' ', expand=True)
# Replacing
df['phone'] = df['phone'].str.replace('-', '')
```

```
# Pattern matching (regex)
df['email_domain'] = df['email'].str.extract(
    r'@(.+)$')
# Check patterns
df['has_digit'] = df['name'].str.contains(
    r'\d', regex=True)
df['starts_a'] = df['name'].str.startswith('A')
# Padding and formatting
df['id_padded'] = df['id'].astype(str)\
    .str.zfill(5)  # '42' → '00042'
```

```
# Real-world: clean messy text data
df['address'] = (df['address']
    .str.strip()                              # Remove whitespace
    .str.replace(r'\s+', ' ', regex=True)     # Normalize spaces
    .str.title()                               # Title Case
    .str.replace(r'\bSt\b', 'Street', regex=True)  # Expand abbreviations
```



.str methods are vectorized and much faster than apply(lambda x: x.upper()). Always prefer .str!

DateTime Operations — Time is Data

```python
# Convert to datetime
df['date'] = pd.to_datetime(df['date'])
df['date'] = pd.to_datetime(
    df['date'], format='%Y-%m-%d')
# Extract components (.dt accessor)
df['year']    = df['date'].dt.year
df['month']   = df['date'].dt.month
df['weekday'] = df['date'].dt.day_name()
df['quarter'] = df['date'].dt.quarter
df['is_weekend'] = df['date'].dt.dayofweek >= 5
```

```python
# Time arithmetic
df['days_ago'] = (
    pd.Timestamp.now() - df['date']
).dt.days
# Date ranges
dates = pd.date_range(
    '2024-01-01', periods=365, freq='D')
# Time-based filtering
recent = df[df['date'] > '2024-01-01']
q1 = df[df['date'].between(
    '2024-01-01', '2024-03-31')]
```

```python
# Resampling time series (requires DatetimeIndex)
ts = df.set_index('date')
monthly = ts['revenue'].resample('M').sum()         # Monthly totals
weekly  = ts['revenue'].resample('W').mean()         # Weekly averages
quarterly = ts.resample('Q').agg({'revenue': 'sum', 'orders': 'count'})
```



Always convert date columns with pd.to_datetime() right after loading data!

Exploring Categorical Data

```
# value_counts — frequency table
df['city'].value_counts()       # Count
df['city'].value_counts(normalize=True)  # %
df['city'].value_counts().head(10) # Top 10
# nunique — number of unique values
df['city'].nunique()             # 42
df.nunique()                     # Per column
# describe for categoricals
df['city'].describe()
# count, unique, top, freq
```

```
# Category dtype — saves memory!
df['city'] = df['city'].astype('category')
# Ordered categories
df['size'] = pd.Categorical(
    df['size'],
    categories=['S', 'M', 'L', 'XL'],
    ordered=True
)
# Now you can compare: df['size'] > 'M'
# Memory comparison: object vs category
# 1M rows: object ~60MB → category ~1MB!
```

```
# Practical: crosstab with percentages
ct = pd.crosstab(df['gender'], df['product'],
                 normalize='all',            # Overall %
                 margins=True)               # With totals
# map/replace — recode categories
df['region'] = df['state'].map({'CA': 'West', 'NY': 'East', 'TX': 'South'})
df['status'] = df['status'].replace({'active': 1, 'inactive': 0})
```

Handling Duplicates

```
# Detect duplicates
df.duplicated()              # Boolean mask
df.duplicated().sum()        # Count duplicates
# Check specific columns
df.duplicated(subset=['name', 'date'])
# Show all duplicate rows
mask = df.duplicated(keep=False)
df[mask].sort_values('name')  # See all copies
```

```
# Remove duplicates
df = df.drop_duplicates()
# Keep first/last occurrence
df = df.drop_duplicates(
    subset=['email'],
    keep='last'      # 'first', 'last', False
)
# Practical: deduplicate logs
# Keep the latest entry per user
df = df.sort_values('timestamp')
df = df.drop_duplicates('user_id', keep='last')
```



Always check for duplicates after loading data AND after merge operations!

```
# Practical: comprehensive data quality check
def data_quality_report(df):
    return pd.DataFrame({
        'dtype': df.dtypes, 'non_null': df.notnull().sum(),
        'null_pct': (df.isnull().mean() * 100).round(1),
        'nunique': df.nunique(), 'sample': df.iloc[0]
    })
```

Sorting & Ranking

```python
# Sort by values
df.sort_values('score', ascending=False)
# Multi-column sort
df.sort_values(
    ['dept', 'salary'],
    ascending=[True, False]  # dept asc, salary desc
)
# Sort by index
df.sort_index()  # Useful after groupby
```

```python
# Ranking
df['salary_rank'] = df['salary'].rank(
    ascending=False, method='min')
# Group-wise ranking
df['dept_rank'] = df.groupby('dept')['salary']\
    .rank(ascending=False, method='dense')
# Percentile rank (0-1)
df['pct_rank'] = df['salary'].rank(pct=True)
```

Ranking Methods

Method	Description	Example (descending)
`average`	(Default) Tie: average of ranks	`[80,90,90,100]` → `[4, 2.5, 2.5, 1]`
`min`	Tie: lowest rank	`[80,90,90,100]` → `[4, 2, 2, 1]`
`max`	Tie: highest rank	`[80,90,90,100]` → `[4, 3, 3, 1]`
`dense`	Like min, but no gaps	`[80,90,90,100]` → `[3, 2, 2, 1]`
`first`	Tie broken by position	`[80,90,90,100]` → `[4, 2, 3, 1]`

Window Functions — Rolling & Expanding

```
# Rolling window (moving statistics)
df['ma_7'] = df['price'].rolling(7).mean()
df['ma_30'] = df['price'].rolling(30).mean()
df['vol_7'] = df['price'].rolling(7).std()
# Min periods (handle start of series)
df['ma'] = df['price'].rolling(
    7, min_periods=1).mean()
# Expanding window (cumulative)
df['cum_mean'] = df['price'].expanding().mean()
df['cum_max']  = df['price'].expanding().max()
df['cum_sum']  = df['price'].cumsum()
```

```
# Shift — compare with previous values
df['prev_price'] = df['price'].shift(1)
df['daily_ret'] = df['price'].pct_change()
df['price_diff'] = df['price'].diff()
# Lead (future values)
df['next_price'] = df['price'].shift(-1)
# Group-wise window operations
df['stock_ma'] = df.groupby('ticker')\
    ['price'].transform(
        lambda x: x.rolling(20).mean())
```

```
# Real-world: technical indicators for stock analysis
df['sma_20'] = df['close'].rolling(20).mean()
df['sma_50'] = df['close'].rolling(50).mean()
df['bollinger_upper'] = df['sma_20'] + 2 * df['close'].rolling(20).std()
df['bollinger_lower'] = df['sma_20'] - 2 * df['close'].rolling(20).std()
df['golden_cross'] = (df['sma_20'] > df['sma_50']) & (df['sma_20'].shift(1) <= df['sma_50'].shift(1))
```

MultiIndex — Hierarchical Indexing

```python
# Create MultiIndex DataFrame
idx = pd.MultiIndex.from_tuples([
    ('Seoul', 2023), ('Seoul', 2024),
    ('Busan', 2023), ('Busan', 2024),
], names=['city', 'year'])
df = pd.DataFrame(
    {'revenue': [100, 120, 80, 95]},
    index=idx)
# GroupBy results are often MultiIndex
result = df.groupby(['city','dept']).mean()
```

```python
# Accessing MultiIndex data
df.loc['Seoul']           # All Seoul rows
df.loc[('Seoul', 2024)]   # Specific tuple
# xs: cross-section (select a level)
df.xs(2024, level='year') # All cities, 2024
# Reset to regular columns
df.reset_index()          # Both levels
df.reset_index(level='year') # Only year
# Set columns as index
df.set_index(['city', 'year'])
```



When in doubt, use .reset_index() to flatten MultiIndex — it's easier to work with regular columns.

```python
# Practical: pivot table creates MultiIndex → flatten for ML
pivot = df.pivot_table(values='revenue', index='city', columns='quarter', aggfunc='sum')
pivot.columns = [f'Q{q}_revenue' for q in pivot.columns]  # Flatten column names
pivot = pivot.reset_index()  # City back as column → ready for modeling
```

Renaming & Dropping Columns

```python
# Rename columns
df = df.rename(columns={
    'old_name': 'new_name',
    'customer_id': 'cust_id'
})
# Rename all columns (function)
df.columns = df.columns.str.lower()
df.columns = df.columns.str.replace(' ', '_')
# Rename with a function
df = df.rename(columns=str.lower)
df = df.rename(columns=lambda c: c.strip())
```

```python
# Drop columns
df = df.drop(columns=['col1', 'col2'])
# Drop rows by index
df = df.drop(index=[0, 1, 2])
# Select/exclude by dtype
nums = df.select_dtypes(include='number')
cats = df.select_dtypes(include='object')
no_dates = df.select_dtypes(
    exclude='datetime')
```

```python
# Practical: clean column names pipeline
def clean_columns(df):
    df = df.copy()
    df.columns = (df.columns
        .str.strip().str.lower().str.replace(r'[^a-z0-9_]', '_', regex=True)
        .str.replace(r'_+', '_', regex=True).str.strip('_'))
    return df
```

apply, map, and transform

```
# map: element-wise on Series
df['grade'] = df['score'].map({
    'A': 4.0, 'B': 3.0, 'C': 2.0})
# apply: flexible function application
# On Series (element-wise)
df['name_len'] = df['name'].apply(len)
df['log_price'] = df['price'].apply(np.log1p)
# On DataFrame (row-wise)
df['full_addr'] = df.apply(
    lambda r: f"{r['city']}, {r['state']}",
    axis=1)
```

```
# transform: returns same shape
df['salary_z'] = df.groupby('dept')\
    ['salary'].transform('mean')
# applymap: element-wise on DataFrame
# (renamed to map() in pandas 2.1+)
df[['a','b']].map(lambda x: f'{x:.2f}')
# Vectorized alternatives (PREFER these!)
# Instead of apply(lambda x: x**2):
df['x_sq'] = df['x'] ** 2
# Instead of apply(len):
df['name_len'] = df['name'].str.len()
```

Performance Hierarchy (fastest → slowest)

Vectorized (NumPy/Pandas)	~1ms	`df['x'] ** 2`
.str / .dt accessors	~10ms	`df['name'].str.upper()`
apply() on Series	~100ms	`df['x'].apply(func)`
apply(axis=1) on DataFrame	~1s	`df.apply(func, axis=1)`
iterrows() / itertuples()	~10s	`for _, row in df.iterrows()`

5

Pandas Advanced Patterns

Method Chaining, Performance, and Real-World Workflows



Method Chaining — The Pandas Way

Chain methods for readable, functional-style data transformations.

```python
# Without chaining (messy)
df2 = df.dropna(subset=['age'])
df2 = df2[df2['age'] > 18]
df2['age_bin'] = pd.cut(df2['age'], 5)
result = df2.groupby('age_bin')['salary'].mean()
result = result.reset_index()
result = result.sort_values('salary')
```

```python
# With chaining (clean!)
result = (df
    .dropna(subset=['age'])
    .query('age > 18')
    .assign(age_bin=lambda x: pd.cut(x['age'],5))
    .groupby('age_bin')['salary']
    .mean()
    .reset_index()
    .sort_values('salary')
)
```

```python
# pipe() — insert custom functions into the chain
def remove_outliers(df, col, n_std=3):
    mean, std = df[col].mean(), df[col].std()
    return df[df[col].between(mean - n_std*std, mean + n_std*std)]
result = (df
    .pipe(remove_outliers, 'salary')   # Custom function in chain!
    .groupby('dept')['salary'].mean())
```



Wrap chains in parentheses ( ) for multi-line readability. Use pipe() for custom functions.

Performance Tips — Speed Up Pandas

From Slow to Fast: Practical Optimization Guide

`for _, row in df.iterrows():` → **Use vectorized operations**

iterrows is 100-1000x slower than vectorized ops

`df.apply(func, axis=1)` → **Use .str/.dt or np.where()**

apply(axis=1) is slow; vectorized alternatives are usually available

`pd.concat in a loop` → **Collect list, concat once**

Growing DataFrame in a loop copies all data each time

`df['col'].astype('object')` → **Use category dtype**

String columns with few uniques → category saves 50-90% memory

`df = pd.read_csv('huge.csv')` → **Use read_parquet or chunking**

Parquet: 2-10x faster, 50-90% smaller; or use chunksize=

Anti-Pattern: Growing DataFrames in Loops

```python
# BAD: concat in a loop O(n²)
result = pd.DataFrame()
for file in csv_files:
    chunk = pd.read_csv(file)
    chunk = process(chunk)
    result = pd.concat([result, chunk])
    # Each iteration copies ALL previous data!
# 10 files × 1M rows: ~45 seconds
# 100 files × 1M rows: ~45 MINUTES!
```

```python
# GOOD: collect list, concat once O(n)
chunks = []
for file in csv_files:
    chunk = pd.read_csv(file)
    chunk = process(chunk)
    chunks.append(chunk)
result = pd.concat(chunks, ignore_index=True)
# 10 files × 1M rows: ~2 seconds
# 100 files × 1M rows: ~20 seconds!
```

```python
# BEST: list comprehension + concat (most Pythonic)
result = pd.concat(
    [process(pd.read_csv(f)) for f in csv_files],
    ignore_index=True
)
# For very large files: chunked reading
chunks = pd.read_csv('huge.csv', chunksize=100_000)
result = pd.concat([process(chunk) for chunk in chunks], ignore_index=True)
```

Memory Optimization

```python
# Check memory usage
df.info(memory_usage='deep')
df.memory_usage(deep=True) / 1e6  # MB per col
# Downcast numeric types
df['price'] = pd.to_numeric(
    df['price'], downcast='float')  # 64→32
df['count'] = pd.to_numeric(
    df['count'], downcast='integer') # 64→8/16
# Object → category (huge savings!)
for col in df.select_dtypes('object'):
    if df[col].nunique() / len(df) < 0.5:
        df[col] = df[col].astype('category')
```

```python
# Practical memory reduction function
def reduce_mem(df):
    start = df.memory_usage(deep=True).sum()/1e6
    for col in df.columns:
        dtype = df[col].dtype
        if dtype == 'float64':
            df[col] = df[col].astype('float32')
        elif dtype == 'int64':
            df[col] = pd.to_numeric(
                df[col], downcast='integer')
        elif dtype == 'object':
            if df[col].nunique()/len(df) < 0.5:
                df[col] =
df[col].astype('category')
    end = df.memory_usage(deep=True).sum()/1e6
```



A typical dataset can be reduced 60-80% with dtype optimization. Always do this for large datasets!

Complete EDA Workflow Example

```
# Step-by-step Exploratory Data Analysis pipeline
import pandas as pd
import numpy as np
# 1. Load & first look
df = pd.read_csv('sales.csv', parse_dates=['date'])
print(df.shape, df.dtypes)
df.head()
# 2. Data quality
print(df.isnull().mean().sort_values(ascending=False))  # Missing %
print(df.duplicated().sum())                              # Duplicates
# 3. Clean & transform
df = (df
    .drop_duplicates()
    .dropna(subset=['customer_id'])
    .assign(
        revenue = lambda x: x['quantity'] * x['unit_price'],
        month   = lambda x: x['date'].dt.to_period('M'),
    )
)
# 4. Analyze
monthly = df.groupby('month').agg(total=('revenue','sum'), orders=('order_id','nunique'))
top_products = df.groupby('product')['revenue'].sum().nlargest(10)
```

6

Best Practices & Summary

Tips, Cheat Sheet, and Where to Go Next



NumPy vs Pandas — When to Use Which?

Use NumPy When...

- Pure numerical computation
- Matrix operations / linear algebra
- All data is the same type (homogeneous)
- Performance is critical (inner loops)
- Building ML algorithms from scratch
- Image/signal processing (pixel arrays)
- You need maximum speed and control

Use Pandas When...

- Working with tabular / mixed-type data
- Data has column labels & row indices
- Need groupby, merge, pivot operations
- Exploratory data analysis (EDA)
- Data cleaning & preprocessing
- Time series analysis with dates
- Reading from CSV, Excel, SQL databases

Common Mistakes to Avoid

✗ df['new'] = df['a'] + df['b'] after filter

✓ Use .loc to avoid SettingWithCopyWarning

```
df.loc[mask, 'new'] = df.loc[mask, 'a'] + df.loc[mask, 'b']
```

✗ Comparing with == None or np.nan

✓ Use .isna() / .notna() — NaN ≠ NaN!

```
df[df['col'].isna()]  # NOT df[df['col'] == None]
```

✗ Forgetting axis= in aggregations

✓ axis=0: across rows, axis=1: across columns

```
df.mean(axis=0)  # Column means (usually what you want)
```

✗ Using inplace=True everywhere

✓ Chain methods instead — cleaner and safer

```
df = df.dropna().reset_index(drop=True)  # NOT inplace
```

✗ Ignoring dtypes after read_csv

✓ Always check/convert dtypes after loading

```
df['id'] = df['id'].astype(str); df['date'] = pd.to_datetime(df['date'])
```

NumPy Quick Reference

Array Creation & Info

```
np.array([1,2,3])        # From list
np.zeros((3,4))           # 3×4 zeros
np.ones((2,3))            # 2×3 ones
np.arange(0,10,0.5)       # Range
np.linspace(0,1,100)      # 100 points
np.random.randn(m,n)      # Random normal
a.shape, a.dtype, a.ndim  # Info
```

Operations

```
a + b, a * b, a @ b       # +, ×, matmul
a.mean(axis=0)             # Col means
a.reshape(3,-1)            # Reshape
np.where(cond, x, y)      # Conditional
np.linalg.solve(A, b)     # Solve Ax=b
a[a > 0], a[mask]         # Boolean idx
```

Indexing & Slicing

```
a[2:5]                    # Slice (view!)
a[::2]                    # Every 2nd
a[[1,3,5]]                # Fancy indexing
a[a > 0]                  # Boolean indexing
a[2:5].copy()             # Independent copy
```

Broadcasting Rules

Compare from trailing dimensions:

- Same size → OK
- One is 1 → stretch to match
- Different → ERROR

```
(100,5) ∘ (5,)   → (100,5)  ✓
(3,4)   ∘ (3,1)  → (3,4)    ✓
(3,4)   ∘ (3,)   → ERROR    ✗
```

Pandas Quick Reference

Data Loading & Inspection

```
pd.read_csv('f.csv')       # Load CSV
pd.read_parquet('f.pq')    # Load Parquet
df.shape, df.dtypes         # Size & types
df.info(), df.describe()    # Summary
df.isnull().sum()           # Missing counts
df.head(), df.sample(5)     # Preview
```

Selection & Filtering

```
df['col']                   # Column (Series)
df.loc[row, col]            # Label-based
df.iloc[row, col]           # Position-based
df[df['col'] > val]         # Boolean filter
df.query('col > val')       # Query string
df['col'].isin([a, b])      # In list
```

Aggregation & Grouping

```
df.groupby('k')['v'].mean()     # Agg
df.groupby('k').agg(n=('v','f')) # Named
.transform()      # Same shape result
.filter()          # Filter groups
pd.merge(a,b,on='k',how='left')  # Join
pd.concat([a,b])   # Stack vertically
```

Cleaning & Reshaping

```
df.dropna(), df.fillna(val)      # Missing
df.drop_duplicates()              # Dupes
df.rename(columns={...})          # Rename
df.astype('category')             # Dtype
df.pivot_table(v,idx,col,agg)     # Pivot
pd.melt(df, id_vars, val_vars)    # Unpivot
df.sort_values('col')             # Sort
df.assign(new=lambda x: ...)      # New col
```

Best Practices Summary

Loading	Use parquet for large data • Specify dtypes • Check shape & nulls immediately
Cleaning	dropna/fillna strategically • Check duplicates after merge • Convert dtypes early
Analysis	GroupBy for aggregation • merge for combining • pivot_table for cross-tabulation
Performance	Vectorize everything • Avoid iterrows/apply(axis=1) • Use category dtype • Parquet > CSV
Code Style	Method chaining with pipe() • query() over brackets • assign() over direct assignment
Memory	Downcast numerics • Category for strings • Chunked reading for huge files
Debugging	Check .shape after every step • Use .head() liberally • assert after merge • .info() often

Key Takeaways

1

NumPy = Speed

Arrays beat lists by 100x+. Think in vectors, not loops. Broadcasting is your best friend.

2

Pandas = Productivity

DataFrames make data manipulation intuitive. Master loc, groupby, and merge — they cover 80% of tasks.

3

Vectorize Everything

If you see a for loop over DataFrame rows, there's almost always a faster vectorized alternative.

4

Data Quality First

Always check dtypes, nulls, and duplicates before analysis. Garbage in = garbage out.

5

Practice with Real Data

Theory isn't enough. Use Kaggle datasets, your own data, or public APIs to build muscle memory.