Python Programming

https://advancedinstitute.ai

Advanced Institute for Artificial Intelligence

Pandas - Data Wrangling 101

Cleaning, Transforming & Analyzing Real-World Data

References

References and Image Sources

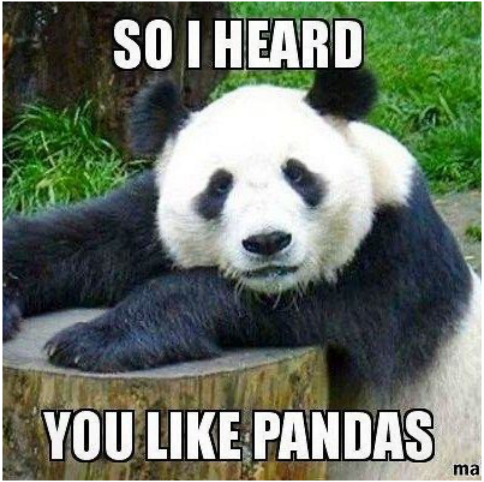
- Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython (Book)
- Learning the Pandas Library: Python Tools for Data Munging, Analysis, and Visualization (Book)
- Official Pandas Documentation

What is Data Wrangling?

Introduction

- **Data Wrangling** (or Data Munging): The process of transforming and mapping raw data into a format suitable for analysis
- Real-world data is often:
  - Messy and incomplete
  - In the wrong format
  - Inconsistent or duplicated
  - Mixed with errors and outliers
- Data scientists spend **60-80% of their time** on data wrangling!
- Pandas is the **primary tool** for data wrangling in Python

Pandas Framework



Why Pandas?

Introduction

- **High-level** data structures for tabular and structured data
- Functions designed to make working with data **fast**, **easy**, and **expressive**
- Combines ideas from:
  - **NumPy**: high-performance array operations
  - **Spreadsheets**: intuitive data manipulation
  - **Relational databases**: SQL-like operations
- Sophisticated indexing for *reshaping*, *filtering*, and *aggregating*
- Data structures inspired by `data.frame` from **R**

The Story of Pandas

History and Evolution

- Created by **Wes McKinney** in 2008 while at AQR Capital Management
  - Need for high-performance, flexible tool for quantitative analysis
- First released as **open source in 2009**
- Name comes from **"Panel Data"** - econometrics term for multi-dimensional data
- Became the **de facto standard** for data manipulation in Python
- Key milestones:
  - 2011: Integration with statsmodels
  - 2015: Major refactoring and performance improvements
  - 2020: Pandas 1.0 released with stability guarantees
  - 2024: Pandas 2.x with major performance enhancements

Pandas Design Philosophy

Core Principles

- **Intuitive** and **easy to use** for data analysis
- Designed for **in-memory analytics** on single machines
- Prioritizes **developer productivity** over raw performance
- Built on top of **NumPy** for numerical operations
- Inspired by **R's data.frame** but with Python ergonomics
- Excellent for **exploratory data analysis (EDA)**
- Trade-off: **Convenience vs. Memory Efficiency**

Pandas and NumPy Relationship

Understanding the Foundation

- Pandas is built **on top of NumPy**
- Each Pandas column is backed by a **NumPy array**
- **NumPy**: homogeneous arrays (single data type)
  - Memory efficient, fast numerical operations
  - No index labels, position-based only
- **Pandas**: heterogeneous DataFrames (multiple data types)
  - Labeled axes (indexes), more intuitive
  - Missing data handling, alignment by labels
  - Higher-level data manipulation operations
- When to use which:
  - **NumPy**: pure numerical computations, arrays
  - **Pandas**: tabular data, mixed types, labeled data

NumPy vs Pandas - Example

Comparing Approaches

```python
# NumPy - position-based access
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
arr[0, 1]  # 2 (row 0, column 1)

# Pandas - label-based access
import pandas as pd
df = pd.DataFrame(arr,
                  index=['row1', 'row2'],
                  columns=['A', 'B', 'C'])
df.loc['row1', 'B']  # 2 (labeled access)

# Pandas uses NumPy underneath
type(df['A'].values)  # <class 'numpy.ndarray'>
```

Memory Management in Pandas

How Pandas Handles Memory

- Each DataFrame column = separate **NumPy array in memory**
- **Data types matter:**
  - `int64` uses 8 bytes per value
  - `int32` uses 4 bytes per value
  - `category` dtype for repeated strings saves memory
  - `object` dtype (strings) most memory-intensive
- **Index overhead:** every DataFrame has an index stored separately
- **Copy-on-write (CoW):** Pandas 2.0+ reduces unnecessary copies
- Rule of thumb: **DataFrame needs 2-3x the CSV size in RAM**

Memory Optimization Example

Reducing Memory Footprint

```python
# Check memory usage
df.info(memory_usage='deep')

# Before optimization
df['City'].dtype  # object (8 bytes + string length)

# After optimization with category
df['City'] = df['City'].astype('category')
# Stores only unique values + integer codes

# Downcast numeric types
df['Sales'] = pd.to_numeric(df['Sales'], downcast='integer')

# Memory comparison
print(f"Memory saved: {original_mem - new_mem} bytes")
```

Part 1

Pandas Fundamentals

One-Dimensional Data

- One-dimensional array with axis labels (index)
- Like a column in a spreadsheet with the **same data type**
- Index can be:
  - Numeric (default: 0, 1, 2, ...)
  - String labels
  - Dates/timestamps

	apples
0	3
1	2
2	0
3	1

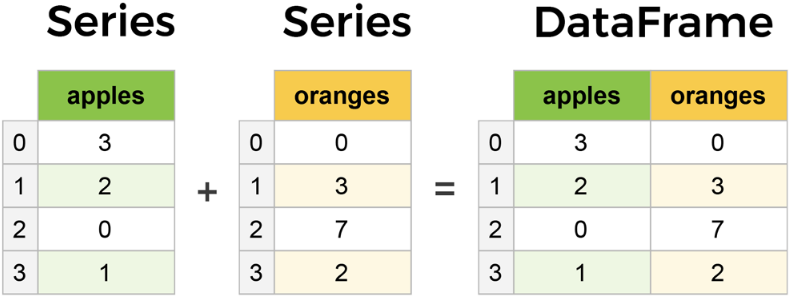
Creating and Accessing Series

```
In [1]: import pandas as pd
In [2]: import numpy as np

# Create a Series with automatic index
In [3]: temperatures = pd.Series([23, 25, 22, 24])
0     23
1     25
2     22
3     24
dtype: int64

# Create a Series with custom index
In [4]: temps = pd.Series([23, 25, 22, 24],
                          index=['Mon', 'Tue', 'Wed', 'Thu'])
Mon     23
Tue     25
Wed     22
```

Two-Dimensional Data



- Table-like structure with rows and columns
- Each column is a Series (can have different data types)
- Indexes for both rows and columns

Creating DataFrames

```
In [5]: data = {
     'City': ['São Paulo', 'Rio', 'Belo Horizonte', 'Salvador'],
     'Population': [12.3, 6.7, 2.5, 2.9],
     'Area_km2': [1521, 1200, 331, 693]
}
In [6]: df = pd.DataFrame(data)
In [7]: df
             City  Population  Area_km2
0       São Paulo        12.3      1521
1             Rio         6.7      1200
2  Belo Horizonte         2.5       331
3        Salvador         2.9       693
```

Reading Data from Files

Essential Skill for Data Wrangling

- Pandas supports multiple file formats:
  - **CSV**: `pd.read_csv('data.csv')`
  - **Excel**: `pd.read_excel('data.xlsx')`
  - **JSON**: `pd.read_json('data.json')`
  - **SQL**: `pd.read_sql(query, connection)`
- Common parameters:
  - `sep`: delimiter (default is comma for CSV)
  - `header`: row number for column names
  - `index_col`: column to use as row index
  - `dtype`: specify data types for columns
  - `parse_dates`: convert columns to datetime

Reading CSV Example

Real-World Data Loading

```
# Basic CSV read
In [8]: df = pd.read_csv('sales_data.csv')

# With custom options
In [9]: df = pd.read_csv('sales_data.csv',
                         sep=';',              # semicolon separator
                         index_col='ID',       # use ID as index
                         parse_dates=['Date'], # convert Date to datetime
                         na_values=['NA', '?']) # treat as missing

# Read from URL
In [10]: url = 'https://example.com/data.csv'
In [11]: df = pd.read_csv(url)
```

Basic Data Exploration

First Look at Your Data

- **Quick inspection:**
  - `df.head(n)`: first n rows (default 5)
  - `df.tail(n)`: last n rows
  - `df.sample(n)`: random n rows
- **Data structure info:**
  - `df.shape`: (rows, columns)
  - `df.columns`: column names
  - `df.index`: row index
  - `df.dtypes`: data types of each column
  - `df.info()`: comprehensive overview

Exploration Example

Understanding Your Dataset

```
In [12]: df.head(3)
   ID       City  Sales  Date
0   1  São Paulo   1500  2024-01-15
1   2        Rio   2300  2024-01-16
2   3   Salvador   1200  2024-01-17

In [13]: df.shape
(1250, 4)

In [14]: df.info()
<class 'pandas.DataFrame'>
RangeIndex: 1250 entries, 0 to 1249
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   ID      1250 non-null   int64
 1   City    1245 non-null   object
```

Statistical Summary

Descriptive Statistics

- `df.describe()`: statistical summary for numeric columns
  - count, mean, std, min, 25%, 50%, 75%, max
- `df.describe(include='all')`: include non-numeric columns
- `df['column'].value_counts()`: frequency of unique values
- `df['column'].unique()`: array of unique values
- `df['column'].nunique()`: number of unique values

Indexing and Selection

Accessing Data

- **Column selection:**
  - `df['column']`: single column (returns Series)
  - `df[['col1', 'col2']]`: multiple columns (returns DataFrame)
- **Row selection by position:**
  - `df.iloc[0]`: first row
  - `df.iloc[0:5]`: first 5 rows
  - `df.iloc[[0, 2, 4]]`: specific rows
- **Row selection by label:**
  - `df.loc['index_label']`: row by index
  - `df.loc['start':'end']`: slice by labels

Selection Examples

Practical Indexing

```
# Select single column
In [15]: df['City']
0       São Paulo
1             Rio
...

# Select multiple columns
In [16]: df[['City', 'Sales']]
          City  Sales
0    São Paulo   1500
1          Rio   2300
...

# Select rows and columns together
In [17]: df.loc[0:2, ['City', 'Sales']]
          City  Sales
0    São Paulo   1500
```

Part 2

Data Cleaning

Why Clean Data?

The Importance of Data Cleaning

- Real-world data is rarely clean:
  - Missing values (NaN, None, empty strings)
  - Duplicate records
  - Incorrect data types
  - Inconsistent formatting
  - Outliers and errors
- Poor data quality leads to:
  - Incorrect analysis results
  - Failed machine learning models
  - Wrong business decisions
- **Clean data is the foundation of good analysis!**

Handling Missing Data

Identifying Missing Values

- Pandas represents missing data as `NaN` (Not a Number)
- **Detection methods:**
  - `df.isnull()`: returns Boolean DataFrame
  - `df.isnull().sum()`: count missing per column
  - `df.notnull()`: opposite of isnull()
- **Visualization:**
  - `df.isnull().sum().sort_values(ascending=False)`
  - Shows which columns have most missing data

Finding Missing Values

```
In [18]: df.isnull().sum()
ID           0
City         5
Sales       20
Date         0
dtype: int64

# Percentage of missing values
In [19]: (df.isnull().sum() / len(df)) * 100
ID        0.00
City      0.40
Sales     1.60
Date      0.00
dtype: float64

# Rows with any missing values
In [20]: df[df.isnull().any(axis=1)]
```

Dealing with Missing Data

Strategies

- **Removing missing data:**
  - `df.dropna()`: drop rows with any NaN
  - `df.dropna(axis=1)`: drop columns with any NaN
  - `df.dropna(thresh=n)`: keep rows with at least n non-NaN values
  - `df.dropna(subset=['col'])`: drop based on specific columns
- **Filling missing data:**
  - `df.fillna(value)`: fill with specific value
  - `df.fillna(method='ffill')`: forward fill
  - `df.fillna(method='bfill')`: backward fill
  - `df.fillna(df.mean())`: fill with column mean

Filling Missing Values

Practical Examples

```
# Fill numeric columns with mean
In [21]: df['Sales'].fillna(df['Sales'].mean(), inplace=True)

# Fill categorical columns with mode
In [22]: df['City'].fillna(df['City'].mode()[0], inplace=True)

# Fill with specific value
In [23]: df['Status'].fillna('Unknown', inplace=True)

# Forward fill (use previous value)
In [24]: df['Price'].fillna(method='ffill', inplace=True)

# Drop rows where critical columns are missing
In [25]: df.dropna(subset=['ID', 'Date'], inplace=True)
```

Removing Duplicates

Finding and Removing Duplicate Records

- **Detection:**
  - `df.duplicated()`: returns Boolean Series
  - `df.duplicated().sum()`: count duplicates
  - `df[df.duplicated()]`: view duplicate rows
- **Removal:**
  - `df.drop_duplicates()`: remove duplicate rows
  - `df.drop_duplicates(subset=['col'])`: based on specific columns
  - `df.drop_duplicates(keep='first')`: keep first occurrence
  - `df.drop_duplicates(keep='last')`: keep last occurrence
  - `df.drop_duplicates(keep=False)`: remove all duplicates

Cleaning Duplicate Data

```
# Check for duplicates
In [26]: df.duplicated().sum()
15

# View duplicates
In [27]: df[df.duplicated(keep=False)]
      ID       City   Sales        Date
10    3   Salvador    1200   2024-01-17
25    3   Salvador    1200   2024-01-17

# Remove duplicates (keep first)
In [28]: df = df.drop_duplicates()

# Remove based on specific columns only
In [29]: df = df.drop_duplicates(subset=['ID'], keep='first')
```

Data Type Conversion

Fixing Data Types

- **Check current types:** `df.dtypes`
- **Convert types:**
  - `df['col'].astype(int)`: convert to integer
  - `df['col'].astype(float)`: convert to float
  - `df['col'].astype(str)`: convert to string
  - `pd.to_numeric(df['col'], errors='coerce')`: convert to numeric, NaN for errors
  - `pd.to_datetime(df['col'])`: convert to datetime
- Common issues:
  - Numbers stored as strings
  - Dates stored as strings
  - Categories stored as objects

Type Conversion Examples

Practical Data Type Fixes

```
# Convert string to numeric (with error handling)
In [30]: df['Sales'] = pd.to_numeric(df['Sales'], errors='coerce')

# Convert to datetime
In [31]: df['Date'] = pd.to_datetime(df['Date'])

# Convert to category (saves memory for repeated values)
In [32]: df['City'] = df['City'].astype('category')

# Check types after conversion
In [33]: df.dtypes
ID                  int64
City             category
Sales             float64
Date       datetime64[ns]
dtype: object
```

Cleaning Text Data

- Access string methods with `.str` accessor:
  - `df['col'].str.lower()`: convert to lowercase
  - `df['col'].str.upper()`: convert to uppercase
  - `df['col'].str.strip()`: remove leading/trailing whitespace
  - `df['col'].str.replace('old', 'new')`: replace text
  - `df['col'].str.contains('pattern')`: check if contains text
  - `df['col'].str.split('delimiter')`: split strings
- Common use cases:
  - Standardizing case
  - Removing extra whitespace
  - Extracting substrings

Part 3

Data Transformation

Filtering Data

Boolean Indexing

- Use conditions to filter rows:
  - `df[df['Sales'] > 1000]`: rows where Sales $>$ 1000
  - `df[df['City'] == 'São Paulo']`: rows for specific city
- Combine multiple conditions:
  - `&`: AND operator
  - `|`: OR operator
  - `~`: NOT operator
- **Important:** Use parentheses around each condition!
- Example: `df[(df['Sales'] > 1000) & (df['City'] == 'Rio')]`

Filtering Examples

Practical Filtering

```
# Simple filter
In [34]: high_sales = df[df['Sales'] > 2000]

# Multiple conditions (AND)
In [35]: result = df[(df['Sales'] > 1500) & (df['City'] == 'Rio')]

# Multiple conditions (OR)
In [36]: result = df[(df['City'] == 'Rio') | (df['City'] == 'São Paulo')]

# Using isin() for multiple values
In [37]: cities = ['Rio', 'São Paulo', 'Salvador']
In [38]: result = df[df['City'].isin(cities)]

# NOT condition
In [39]: result = df[~df['City'].isin(['Rio'])]
```

Creating New Columns

Adding Calculated Fields

- **Simple assignment:**
  - `df['new_col'] = value`: create with constant
  - `df['total'] = df['price'] * df['quantity']`: calculation
- **Based on conditions:**
  - `np.where(condition, value_if_true, value_if_false)`
  - Multiple conditions: `np.select()`
- **From existing columns:**
  - Mathematical operations
  - String operations
  - Date operations

Practical Column Creation

```
# Simple calculation
In [40]: df['Sales_k'] = df['Sales'] / 1000

# Conditional column
In [41]: df['Performance'] = np.where(df['Sales'] > 2000,
                                       'High',
                                       'Low')


# Multiple conditions
In [42]: conditions = [
    df['Sales'] > 3000,
    df['Sales'] > 2000,
    df['Sales'] > 1000
]
In [43]: choices = ['Excellent', 'Good', 'Average']
In [44]: df['Category'] = np.select(conditions, choices, default='Poor')
```

Applying Functions

Transform Data with Functions

- **apply() method:** apply function to rows or columns
  - `df['col'].apply(func)`: apply to each element in column
  - `df.apply(func, axis=0)`: apply to each column
  - `df.apply(func, axis=1)`: apply to each row
- **Lambda functions:** quick anonymous functions
  - `df['col'].apply(lambda x: x * 2)`
- **map() method:** map values using dictionary
  - `df['col'].map({'old': 'new'})`
- Use for complex transformations not possible with simple operations

Apply and Lambda Examples

Function Application

```
# Apply lambda to single column
In [45]: df['Sales_taxed'] = df['Sales'].apply(lambda x: x * 1.15)

# Apply custom function
In [46]: def categorize_sales(value):
             if value > 2500:
                 return 'High'
             elif value > 1500:
                 return 'Medium'
             else:
                 return 'Low'

In [47]: df['Level'] = df['Sales'].apply(categorize_sales)

# Map values using dictionary
In [48]: city_codes = {'São Paulo': 'SP', 'Rio': 'RJ', 'Salvador': 'BA'}
In [49]: df['Code'] = df['City'].map(city_codes)
```

Sorting Data

Ordering Your Data

- **Sort by values:**
  - `df.sort_values('col')`: sort by single column
  - `df.sort_values(['col1', 'col2'])`: sort by multiple columns
  - `ascending=False`: descending order
- **Sort by index:**
  - `df.sort_index()`: sort by row index
- **Parameters:**
  - `inplace=True`: modify original DataFrame
  - `na_position`: where to put NaN values ('first' or 'last')

Sorting Examples

Practical Sorting

```
# Sort by single column (ascending)
In [50]: df_sorted = df.sort_values('Sales')

# Sort descending
In [51]: df_sorted = df.sort_values('Sales', ascending=False)

# Sort by multiple columns
In [52]: df_sorted = df.sort_values(['City', 'Sales'],
                                    ascending=[True, False])

# Sort and modify original
In [53]: df.sort_values('Date', inplace=True)

# Reset index after sorting
In [54]: df = df.sort_values('Sales').reset_index(drop=True)
```

Renaming Columns

Better Column Names

- **Rename specific columns:**
  - `df.rename(columns={'old': 'new'})`
- **Rename all columns:**
  - `df.columns = ['name1', 'name2', ...]`
- **Clean column names:**
  - Remove spaces: `df.columns.str.replace(' ', '_')`
  - Lowercase: `df.columns.str.lower()`
  - Strip whitespace: `df.columns.str.strip()`
- Good practice: use lowercase with underscores

Part 4

Aggregation & Grouping

GroupBy Operations

Split-Apply-Combine Pattern

- **GroupBy** is one of the most powerful Pandas features
- Three-step process:
  - **Split**: divide data into groups based on criteria
  - **Apply**: apply function to each group independently
  - **Combine**: combine results into a data structure
- Basic syntax: `df.groupby('column')`
- Similar to SQL's GROUP BY

Simple Aggregations

```
# Group by single column and calculate mean
In [55]: df.groupby('City')['Sales'].mean()
City
Belo Horizonte    1850.5
Rio               2150.3
Salvador          1320.8
São Paulo         2850.2
Name: Sales, dtype: float64

# Group by and get multiple statistics
In [56]: df.groupby('City')['Sales'].agg(['mean', 'sum', 'count'])
                mean     sum  count
City
Belo Horizonte  1850.5  22206     12
Rio             2150.3  64509     30
...
```

Aggregation Functions

Common Aggregations

- After groupby, apply aggregation functions:
  - `.sum()`: sum of values
  - `.mean()`: average
  - `.median()`: median value
  - `.min()`, `.max()`: minimum/maximum
  - `.count()`: number of non-null values
  - `.std()`, `.var()`: standard deviation/variance
- **agg() method:** apply multiple functions at once
  - `df.groupby('City').agg(['mean', 'sum', 'count'])`

Multiple Aggregations

Advanced Grouping

```
# Different aggregations for different columns
In [57]: df.groupby('City').agg({
    'Sales': ['mean', 'sum'],
    'Quantity': 'count',
    'Profit': ['min', 'max']
})

# Group by multiple columns
In [58]: df.groupby(['City', 'Product'])['Sales'].sum()

# Custom aggregation function
In [59]: df.groupby('City')['Sales'].agg(
    lambda x: x.max() - x.min()
)
```

Excel-Style Pivot Tables

- **pivot_table():** create spreadsheet-style pivot tables
- Key parameters:
  - `values`: column to aggregate
  - `index`: column(s) for rows
  - `columns`: column(s) for columns
  - `aggfunc`: aggregation function (default: mean)
  - `fill_value`: value for missing combinations
- Useful for cross-tabulation and multi-dimensional aggregation

Pivot Table Example

Creating Pivot Tables

```
# Simple pivot table
In [60]: pd.pivot_table(df,
                        values='Sales',
                        index='City',
                        columns='Month',
                        aggfunc='sum')
Month            Jan      Feb      Mar
City
Rio          12300.0  15200.0  14100.0
Salvador      8900.0   9200.0  10500.0
São Paulo    18500.0  19200.0  21300.0

# With multiple aggregations
In [61]: pd.pivot_table(df, values='Sales',
                        index='City',
                        aggfunc=['sum', 'mean', 'count'])
```

Part 5

Combining Datasets

Concatenating DataFrames

Stacking Data

- **concat():** combine DataFrames along an axis
- Vertical concatenation (stack rows):
  - `pd.concat([df1, df2])`
  - Use when combining data with same columns
- Horizontal concatenation (stack columns):
  - `pd.concat([df1, df2], axis=1)`
- Parameters:
  - `ignore_index=True`: create new sequential index
  - `keys`: create hierarchical index

Combining DataFrames

```
# Vertical concatenation (adding rows)
In [62]: jan_data = pd.read_csv('january.csv')
In [63]: feb_data = pd.read_csv('february.csv')
In [64]: combined = pd.concat([jan_data, feb_data], ignore_index=True)

# Horizontal concatenation (adding columns)
In [65]: sales = pd.DataFrame({'Sales': [100, 200]})
In [66]: costs = pd.DataFrame({'Costs': [60, 120]})
In [67]: combined = pd.concat([sales, costs], axis=1)
   Sales  Costs
0    100     60
1    200    120
```

Merging DataFrames

SQL-Style Joins

- **merge():** combine DataFrames based on common columns
- Types of joins:
  - `how='inner'`: keep only matching rows (default)
  - `how='left'`: keep all rows from left DataFrame
  - `how='right'`: keep all rows from right DataFrame
  - `how='outer'`: keep all rows from both
- Parameters:
  - `on`: column name(s) to join on
  - `left_on`, `right_on`: different column names

Joining DataFrames

```
# Inner join (default)
In [68]: customers = pd.DataFrame({'ID': [1,2,3], 'Name': ['Ana','Bob','Carol']})
In [69]: orders = pd.DataFrame({'ID': [1,2,4], 'Amount': [100,200,150]})
In [70]: pd.merge(customers, orders, on='ID')
   ID  Name  Amount
0   1   Ana     100
1   2   Bob     200

# Left join (keep all customers)
In [71]: pd.merge(customers, orders, on='ID', how='left')
   ID  Name  Amount
0   1   Ana   100.0
1   2   Bob   200.0
2   3 Carol     NaN
```

When to Use Each Method

Concat vs Merge

- **Use concat() when:**
  - Combining data with same structure (same columns)
  - Stacking data vertically (e.g., monthly files)
  - Adding new columns to existing data
- **Use merge() when:**
  - Joining related data from different sources
  - Combining based on a key/relationship
  - Need SQL-like join operations
- Think: concat for simple stacking, merge for relational joins

Wrap-Up

Performance and Parallel Processing

Pandas Memory Challenges

Limitations for Large Datasets

- **In-memory only:** entire dataset must fit in RAM
  - Typical limit: **5-10 GB** on consumer hardware
  - With 32 GB RAM: safely handle **10-15 GB datasets**
- **Single-threaded by default:** doesn't utilize all CPU cores
- **Memory copies:** operations often create intermediate copies
- **String overhead:** object dtype stores Python objects (expensive)
- **Index overhead:** can be 20-30% of total memory
- When Pandas struggles:
  - Datasets larger than available RAM
  - Operations requiring multiple passes over data
  - Need for distributed processing

Scaling Beyond Pandas

When You Need More Power

- **For larger-than-memory datasets:**
  - **Dask**: parallel computing with Pandas-like API
  - **Vaex**: lazy evaluation, out-of-core processing
  - **Polars**: Rust-based, faster and more memory efficient
- **For distributed computing:**
  - **PySpark**: distributed DataFrames across cluster
  - **Ray**: distributed Python for large-scale data
- **For columnar analytics:**
  - **Apache Arrow**: in-memory columnar format
  - **DuckDB**: SQL on Pandas DataFrames

Alternative: Dask

Parallel Pandas

- Extends Pandas to **parallel and distributed computing**
- **Familiar API:** looks like Pandas, scales like Spark
- Use cases:
  - Datasets **larger than RAM** (10s to 100s of GB)
  - Parallel processing on **multi-core machines**
  - Distributed processing on **clusters**
- **Lazy evaluation:** computes only when needed
- Limitations:
  - Not all Pandas operations supported
  - Overhead for small datasets
  - Requires understanding of parallel computing

Using Dask for Large Datasets

```python
import dask.dataframe as dd

# Read large CSV that doesn't fit in memory
df = dd.read_csv('huge_file.csv')

# Dask API similar to Pandas
result = df.groupby('City')['Sales'].mean()

# Lazy evaluation - nothing computed yet!
# Compute triggers execution
result.compute()

# Parallel processing across multiple cores
df = dd.read_csv('data/*.csv')  # Read many files
df.groupby('Category').sum().compute()
```

Alternative: Vaex

Out-of-Core DataFrames

- Designed for **lazy, out-of-core DataFrames**
- **Memory-mapped files:** doesn't load entire dataset
- Key features:
  - **Instant visualization** of billion-row datasets
  - **Zero memory copy** operations
  - **Expression system** - only computes what's needed
  - Excellent for **exploratory analysis** of huge datasets
- Best for:
  - Numerical data (astronomy, finance, sensor data)
  - Aggregations and filtering on large datasets
  - When you need speed without loading everything
- Trade-offs:
  - Smaller ecosystem than Pandas
  - Works best with specific file formats (HDF5, Apache Arrow)

Vaex Example

Billion Row Processing

```python
import vaex

# Open large file - loads metadata only, not data
df = vaex.open('huge_dataset.hdf5')

# Instant statistics on billion rows
print(df.mean('Sales'))  # No data loaded!

# Lazy filtering and aggregation
filtered = df[df.Sales > 1000]
result = filtered.groupby('City').agg({'Sales': 'mean'})

# Only computes when exporting
result.export('result.csv')

# Memory efficient - processes in chunks
print(df.memory_usage())  # Minimal!
```

Alternative: Polars

Next-Generation DataFrame Library

- Written in **Rust**, Python bindings
- **Multi-threaded by default** - uses all CPU cores
- Key advantages:
  - **2-10x faster** than Pandas for most operations
  - **Lower memory usage** through better algorithms
  - **Lazy evaluation** with query optimization
  - **Type system** prevents common errors
- API differences from Pandas:
  - Method chaining preferred over in-place operations
  - Expressions instead of direct column access
- Best for:
  - New projects where performance matters
  - Data that fits in memory but needs speed
  - ETL pipelines and data transformations

High-Performance Analytics

```python
import polars as pl

# Read CSV - uses all CPU cores
df = pl.read_csv('sales_data.csv')

# Lazy evaluation with query optimization
result = (
    df.lazy()
    .filter(pl.col('Sales') > 1000)
    .groupby('City')
    .agg([
        pl.col('Sales').mean().alias('avg_sales'),
        pl.col('Sales').sum().alias('total_sales')
    ])
    .collect()  # Execute optimized query
)

```

Comparison: When to Use What?

Decision Guide

- **Pandas:**
  - Data **fits in memory** ($<$ 10 GB)
  - Exploratory data analysis, prototyping
  - Maximum ecosystem support and documentation
- **Polars:**
  - Need **speed** and **efficiency** on single machine
  - Data fits in memory but performance matters
  - New projects, willing to learn new API
- **Dask:**
  - Data **larger than RAM** (10-1000 GB)
  - Need to scale to multiple machines
  - Want Pandas-like API

Comparison: When to Use What? (cont.)

Decision Guide

- **Vaex:**
  - **Billions of rows**, numerical data
  - Exploratory analysis without loading full dataset
  - Streaming data processing
- **PySpark:**
  - **Truly big data** (TBs to PBs)
  - Existing Spark infrastructure
  - Complex distributed workflows
- **Rule of thumb:**
  - Start with Pandas
  - Optimize with Polars when needed
  - Scale with Dask/Spark when data grows

Performance Tips for Pandas

Making Pandas Faster

- **Use appropriate data types:**
  - category for repeated strings
  - Downcast integers (int64 → int32 → int16)
- **Vectorized operations:** avoid loops
  - Use built-in methods instead of `apply()`
  - `df['col'] * 2` instead of `df['col'].apply(lambda x: x * 2)`
- **Read data efficiently:**
  - Specify dtypes when reading CSV
  - Use `usecols` to read only needed columns
  - Consider Parquet format instead of CSV
- **Use Copy-on-Write (CoW):** Pandas 2.0+
- **Chunk large operations:** process in batches

Performance Optimization Example

Before and After

```python
# SLOW - Python loop
for idx, row in df.iterrows():
    df.loc[idx, 'result'] = row['a'] * row['b']

# FAST - Vectorized operation
df['result'] = df['a'] * df['b']

# SLOW - apply with lambda
df['price_taxed'] = df['price'].apply(lambda x: x * 1.15)

# FAST - Direct vectorization
df['price_taxed'] = df['price'] * 1.15

# Memory efficient reading
df = pd.read_csv('data.csv',
                 dtype={'City': 'category'},
                 usecols=['City', 'Sales'],
```

The Future of Pandas

What's Next

- **Pandas 2.x and beyond:**
  - Copy-on-Write (CoW) as default
  - Better memory management
  - Improved performance through refactoring
  - PyArrow backend for better efficiency
- **Integration with Arrow ecosystem**
- **Continued focus on:**
  - User experience and API consistency
  - Better error messages
  - Type hints and modern Python features
- Pandas remains **essential** even as alternatives emerge
  - Lingua franca of data analysis in Python
  - Gateway to more specialized tools

Key Takeaways - Architecture & Scaling

Summary

- Pandas is built on NumPy, trades memory for convenience
- Designed for in-memory, single-machine analytics
- Memory footprint: **2-3x the data size**
- Practical limit: **5-15 GB datasets** on typical hardware
- For larger data:
  - Polars: speed on single machine
  - Dask: larger-than-memory, multi-machine
  - Vaex: billion-row exploration
  - PySpark: truly big data (TB+)
- **Pandas is not going away** - it's the foundation!
- Learn Pandas first, then explore alternatives as needed

Wrap-Up

Key Takeaways & Best Practices

Data Wrangling Workflow

Typical Process

- **1. Load Data**: read from files or databases
- **2. Explore**: understand structure, types, and issues
  - head(), info(), describe(), shape, dtypes
- **3. Clean**: handle missing data, duplicates, types
  - dropna(), fillna(), drop_duplicates(), astype()
- **4. Transform**: filter, create columns, apply functions
  - Boolean indexing, apply(), new columns
- **5. Aggregate**: group and summarize
  - groupby(), pivot_table()
- **6. Combine**: merge multiple datasets
  - concat(), merge()

Best Practices

Tips for Effective Data Wrangling

- **Always explore before cleaning**
  - Understand your data first
- **Document your assumptions**
  - How you handle missing data, why you remove outliers
- **Keep original data intact**
  - Work on copies: `df_clean = df.copy()`
- **Use meaningful variable names**
- **Chain operations carefully**
  - Method chaining is powerful but can be hard to debug
- **Save intermediate results**
  - Especially after time-consuming operations

Common Mistakes to Avoid

Pitfalls

- **Not using inplace parameter correctly**
  - Remember: most operations return a new DataFrame
- **Forgetting to reset index after sorting/filtering**
  - Use `reset_index(drop=True)`
- **Not handling missing data properly**
  - Understand why data is missing before deciding what to do
- **Ignoring data types**
  - Wrong types lead to wrong results
- **Not checking for duplicates**
- **Overwriting original data without backup**

Resources for Learning More

Continue Your Journey

- **Official Documentation:**
  - pandas.pydata.org/docs
- **Books:**
  - Python for Data Analysis by Wes McKinney (Pandas creator)
- **Practice Datasets:**
  - Kaggle datasets: kaggle.com/datasets
  - UCI Machine Learning Repository
- **Interactive Learning:**
  - DataCamp, Coursera, edX courses on Pandas
- **Practice is key!** Work with real datasets

Questions?