Day 2, Session 1: Introduction to Pandas

Three-Day Data Analysis with Python Course

Yesterday's Recap

We covered Python fundamentals:

- Variables, data types, operators
- Control flow (if/else, loops)
- Functions
- Data structures (lists, dicts, tuples, sets)
- NumPy arrays
- File operations

Today: We move from Python basics to real data analysis!

Overview of Session 1

1. **From SQL to Pandas**
2. **Pandas Fundamentals**
3. **Data Loading and Inspection**
4. **Data Selection and Indexing**

Part 1: From SQL to Pandas

Why Pandas?

pandas is the industry-standard library for data analysis in Python.

- Built on top of NumPy
- Powerful data manipulation capabilities
- Intuitive syntax similar to SQL
- Handles messy real-world data
- Integrates seamlessly with visualization libraries

If you know SQL, you'll feel right at home!

SQL Tables vs DataFrames

Concept	SQL	Pandas
Data structure	Table	DataFrame
Row	Record/Row	Row
Column	Column/Field	Column/Series
Filter rows	`WHERE`	Boolean indexing
Select columns	`SELECT`	Column selection
Aggregate	`GROUP BY`	`.groupby()`
Join tables	`JOIN`	`.merge()`
Sort	`ORDER BY`	`.sort_values()`

Row-Based vs Columnar Storage

SQL (Row-based):

- Optimized for transactional operations
- Reads entire rows at a time

Pandas (Columnar):

- Optimized for analytical operations
- Reads columns at a time
- Much faster for aggregations and calculations

Why DataFrames Are Powerful

Key advantages:

1. **Labeled axes** - Rows and columns have names
2. **Mixed types** - Different columns can have different types
3. **Size mutable** - Can add/remove rows and columns
4. **Missing data** - Built-in handling of missing values
5. **Group operations** - Easy split-apply-combine
6. **Alignment** - Automatic data alignment by label

Part 2: Pandas Fundamentals

Importing Pandas

```python
# Import pandas (standard alias is 'pd')
import pandas as pd

# Also import numpy (often used together)
import numpy as np

print(f"pandas version: {pd.__version__}")
```

Convention: Always use `pd` as the alias for pandas.

Series: One-Dimensional Labeled Arrays

A **Series** is a one-dimensional array with labels (index).

- Similar to a column in a spreadsheet
- Built on top of NumPy arrays
- Has an index and values

Syntax:

```
series = pd.Series(data, index=index)
```

Series Examples

```python
# Create a Series from a list
sales = pd.Series([1200, 1450, 1100, 1800, 2100])
print(sales)

# Create a Series with custom index
daily_sales = pd.Series(
    [1200, 1450, 1100, 1800, 2100],
    index=['Mon', 'Tue', 'Wed', 'Thu', 'Fri']
)
print(daily_sales)

# Access by index
print(f"Monday sales: ${daily_sales['Mon']}")
```

Series (Continued)

```python
# Series from dictionary
prices = pd.Series({
    'Laptop': 999.99,
    'Mouse': 29.99,
    'Keyboard': 79.99,
    'Monitor': 299.99
})
print(prices)

# Series have attributes
print(f"Index: {prices.index}")
print(f"Values: {prices.values}")
print(f"Data type: {prices.dtype}")
```

DataFrames: Two-Dimensional Labeled Data

A **DataFrame** is a two-dimensional labeled data structure.

- Think of it as a table or spreadsheet
- Collection of Series (each column is a Series)
- Has both row index and column names

This is the primary data structure you'll work with!

DataFrame Anatomy

```
         Column1  Column2  Column3
Index1   val11    val12    val13
Index2   val21    val22    val23
Index3   val31    val32    val33
```

- **Columns**: Named Series
- **Index**: Row labels (like primary key)
- **Values**: The actual data

Creating DataFrames

Multiple ways to create DataFrames:

1. From dictionaries
2. From lists of dictionaries
3. From NumPy arrays
4. **From files** (most common!)

Let's see examples...

DataFrame from Dictionary

```python
# Dictionary of lists (each key becomes a column)
data = {
    'product': ['Laptop', 'Mouse', 'Keyboard', 'Monitor'],
    'price': [999.99, 29.99, 79.99, 299.99],
    'stock': [15, 120, 45, 30],
    'category': ['Computer', 'Accessory', 'Accessory', 'Computer']
}

df = pd.DataFrame(data)
print(df)
```

DataFrame from List of Dictionaries

```python
# List of dictionaries (each dict becomes a row)
customers = [
    {'id': 101, 'name': 'Alice', 'city': 'NYC', 'purchases': 15},
    {'id': 102, 'name': 'Bob', 'city': 'LA', 'purchases': 8},
    {'id': 103, 'name': 'Carol', 'city': 'Chicago', 'purchases': 22}
]

customers_df = pd.DataFrame(customers)
print(customers_df)
```

Part 3: Data Loading and Inspection

Reading CSV Files

The most common way to create DataFrames:

```
# Read CSV file
df = pd.read_csv('filename.csv')

# Common parameters
df = pd.read_csv(
    'filename.csv',
    sep=',',            # Delimiter (default is comma, sometimes "\t")
    header=0,           # Row number for column names
    index_col=None,     # Column to use as index
    na_values=['NA']    # Additional strings to recognize as NaN
)
```

Reading Other File Formats

Pandas supports many file formats:

```
# JSON files
df = pd.read_json('data.json')

# Excel files
df = pd.read_excel('data.xlsx', sheet_name='Sheet1')

# SQL databases
df = pd.read_sql('SELECT * FROM table', connection)

# Many more: HDF5, Parquet...
```

Example: Reading Customer Data

```python
# Let's load some customer data
customers = pd.read_csv('customers.csv')

# What do we have?
print(type(customers))  # DataFrame
print(f"Shape: {customers.shape}")  # (rows, columns)
```

Basic DataFrame Inspection

Method	Description
`.head(n)`	First n rows (default 5)
`.tail(n)`	Last n rows (default 5)
`.info()`	Column types and missing values
`.describe()`	Statistical summary
`.shape`	Dimensions (rows, columns)
`.columns`	Column names
`.dtypes`	Data types of each column

`.head()` and `.tail()`

```python
# View first 5 rows
print(customers.head())

# View first 3 rows
print(customers.head(3))

# View last 5 rows
print(customers.tail())
```

Always start with `.head()` to see what you're working with!

`.info()` - Understanding Your Data

```
customers.info()
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 5 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   id         100 non-null    int64
 1   name       100 non-null    object
 2   email      98 non-null     object
 3   purchases  100 non-null    int64
 4   total      100 non-null    float64
dtypes: float64(1), int64(2), object(2)
memory usage: 4.0+ KB
```

`.describe()` - Statistical Summary

```
customers.describe()
```

Output:

```
               id   purchases        total
count   100.00000  100.000000   100.000000
mean    150.50000   12.450000  1845.234000
std      29.01149    7.823421   892.451234
min     101.00000    1.000000   150.500000
25%     125.75000    6.000000  1021.750000
50%     150.50000   11.500000  1789.250000
75%     175.25000   18.000000  2543.000000
max     200.00000   35.000000  4950.000000
```

Only includes numerical columns by default!

Understanding Data Types

Common pandas dtypes:

Dtype	Python Type	Usage
`int64`	int	Integer numbers
`float64`	float	Decimal numbers
`object`	str	Text/strings
`bool`	bool	True/False
`datetime64`	datetime	Dates and times
`category`	-	Categorical data

Shape and Size

```python
# Get dimensions
rows, cols = customers.shape
print(f"Rows: {rows}, Columns: {cols}")

# Number of elements
print(f"Total elements: {customers.size}")

# Number of rows
print(f"Row count: {len(customers)}")
```

Part 4: Data Selection and Indexing

Selecting Columns

Several ways to select columns:

```python
# Single column (returns Series)
names = customers['name']
print(type(names))  # Series

# Single column alternative (if no spaces in name)
names = customers.name

# Multiple columns (returns DataFrame)
subset = customers[['name', 'email']]
print(type(subset))  # DataFrame
```

Note: Double brackets `[[...]]` for multiple columns!

Selecting Columns (Example)

```python
# Select single column
print("Customer names:")
print(customers['name'].head())

# Select multiple columns
print("\nCustomer contact info:")
print(customers[['name', 'email']].head())

# Calculate on a column
total_purchases = customers['purchases'].sum()
print(f"\nTotal purchases: {total_purchases}")
```

Selecting Rows by Position

Use `.iloc[]` for integer-location based indexing:

```python
# First row
first_customer = customers.iloc[0]

# First 5 rows
first_five = customers.iloc[0:5]

# Last row
last_customer = customers.iloc[-1]

# Specific rows
some_rows = customers.iloc[[0, 5, 10]]
```

Selecting Rows by Label

Use `.loc[]` for label-based indexing:

```
# If index is set to a column (e.g., customer ID)
customers_indexed = customers.set_index('id')

# Select by index label
customer_125 = customers_indexed.loc[125]

# Select range of labels
customers_range = customers_indexed.loc[101:105]
```

`.loc` vs `.iloc`

Method	Type	Example
`.iloc[n]`	Integer position	`df.iloc[0]` (first row)
`.loc[label]`	Index label	`df.loc['2024-01-01']`

Key difference:

- `.iloc` - Think "i" for "integer"
- `.loc` - Think "l" for "label"

Boolean Indexing (Filtering)

The most powerful selection method - like SQL's `WHERE` clause!

```python
# Filter rows where purchases > 10
active_customers = customers[customers['purchases'] > 10]

# Filter with multiple conditions (use & and |)
premium = customers[
    (customers['purchases'] > 15) &
    (customers['total'] > 2000)
]

# Filter by string matching
nyc_customers = customers[customers['city'] == 'NYC']
```

This is how you'll filter data most of the time!

Boolean Indexing (Example)

```
# High-value customers
high_value = customers[customers['total'] > 2000]
print(f"High-value customers: {len(high_value)}")

# Active customers in NYC
active_nyc = customers[
    (customers['purchases'] > 10) &
    (customers['city'] == 'NYC')
]
print(f"\nActive NYC customers:\n{active_nyc[['name', 'purchases']]}")
```

Combining Selection Methods

You can combine row and column selection:

```
# Select specific rows and columns with .loc
subset = customers.loc[0:5, ['name', 'total']]

# Boolean filter + column selection
high_value_names = customers.loc[
    customers['total'] > 2000,
    ['name', 'total']
]

# .iloc with row and column positions
subset = customers.iloc[0:10, 0:3]
```

Practical Example: Customer Segmentation

```python
# Load customer data
customers = pd.read_csv('customers.csv')

# View structure
print(customers.info())
print(customers.head())

# Segment 1: VIP customers (>20 purchases, >$3000 total)
vip = customers[
    (customers['purchases'] > 20) &
    (customers['total'] > 3000)
]

# Segment 2: At-risk (few purchases, low total)
at_risk = customers[
    (customers['purchases'] < 5) &
    (customers['total'] < 500)
]
```

Practical Example (Continued)

```python
# Segment 3: Active (10-20 purchases)
active = customers[
    (customers['purchases'] >= 10) &
    (customers['purchases'] <= 20)
]

# Analyze segments
print(f"VIP customers: {len(vip)} ({len(vip)/len(customers)*100:.1f}%)")
print(f"At-risk customers: {len(at_risk)} ({len(at_risk)/len(customers)*100:.1f}%)")
print(f"Active customers: {len(active)} ({len(active)/len(customers)*100:.1f}%)")

# Average spending by segment
print(f"\nAverage VIP spending: ${vip['total'].mean():.2f}")
print(f"Average at-risk spending: ${at_risk['total'].mean():.2f}")
```

Session 1 Summary

✓ **SQL to Pandas**: DataFrames are like SQL tables with superpowers
✓ **Fundamentals**: Series and DataFrames are the core structures
✓ **Loading Data**: `read_csv()`, `read_excel()`, and other readers
✓ **Inspection**: `.head()`, `.info()`, `.describe()`, `.shape`
✓ **Selection**: Column selection, `.loc`, `.iloc`, boolean indexing

Questions?

Break time! ☕

See you in Session 2!