CS109A: Introduction to Data Science
Lecture 03: Introduction to Pandas

Harvard University

Fall 2024

- **Course:** CS109A: Introduction to Data Science
- **Lecture:** Lecture 03: Introduction to Pandas
- **Instructor:** Chris Gumb
- **Objective:** Learn the fundamentals of Pandas for data manipulation and analysis in Python

Key Summary

This lecture introduces **Pandas**, the essential Python library for data manipulation and analysis. You'll learn about the two core data structures: `Series` (1-dimensional) and `DataFrame` (2-dimensional tables). We cover how to create these structures, load data from CSV files, inspect and clean data, select subsets using boolean indexing and the `loc`/`iloc` accessors, and perform aggregations with `groupby`. By the end of this lecture, you'll have the foundational skills needed to work with tabular data in Python.

Contents

1 Introduction: Why Pandas?

1.1 What is Pandas?

Definition:

Pandas **Pandas** is a Python library that provides high-performance, easy-to-use data structures and data analysis tools. The name comes from "Panel Data" (a term from econometrics) and "Python Data Analysis."

Pandas gives you access to data types that don't exist in standard Python:

- **Series**: A 1-dimensional labeled array
- **DataFrame**: A 2-dimensional labeled table (like an Excel spreadsheet)

1.2 Why Use Pandas Instead of Plain Python?

You could store data in Python lists and dictionaries, but Pandas offers significant advantages:

- **Speed**: Pandas is built on NumPy, which stores data in contiguous memory blocks for fast operations
- **Convenience**: Methods like `.head()`, `.describe()`, `.groupby()` make common operations trivial
- **Data Alignment**: Pandas automatically aligns data by labels during operations
- **Missing Data Handling**: Built-in support for missing values (`NaN`)
- **File I/O**: Easy reading/writing of CSV, Excel, SQL databases, JSON, and more

1.3 Importing Pandas

The standard convention is to import pandas with the alias `pd`:

```
1 import pandas as pd
2 import numpy as np  # Often used alongside pandas
```

Warning

Don't break conventions! Always use `pd` for pandas and `np` for numpy. Using other aliases will confuse anyone reading your code.

2 The Series Data Structure

2.1 What is a Series?

Definition:

Series A **Series** is a one-dimensional array with labels (called an **index**). Think of it as:

- A single column from a spreadsheet
- A dictionary where keys are the index and values are the data
- A NumPy array with attached labels

2.2 Creating a Series

The simplest way is from a Python list:

```
# Create a simple Series
my_list = [10, 20, 30, 40, 50]
s = pd.Series(my_list)
print(s)
```

Listing 1: Creating a Series from a list

Output:

```
0    10
1    20
2    30
3    40
4    50
dtype: int64
```

Notice the output shows:

- **Left column**: The **index** (0, 1, 2, 3, 4)—created automatically
- **Right column**: The **values** from your list
- **Bottom**: The **dtype** (data type)—`int64` means 64-bit integers

2.3 Series Attributes

Every Series has four key attributes:

```
s = pd.Series([10, 20, 30], index=['a', 'b', 'c'], name='my_series')

print(s.values)    # The actual data (as a NumPy array)
# Output: array([10, 20, 30])

print(s.index)     # The labels
# Output: Index(['a', 'b', 'c'], dtype='object')
```

```
8
9  print(s.name)      # The series name (becomes column name in DataFrame)
10 # Output: 'my_series'
11
12 print(s.dtype)     # The data type
13 # Output: dtype('int64')
```

Listing 2: Series attributes

2.4 Custom Indexes

The index doesn't have to be integers—you can use any labels:

```
s = pd.Series([100, 200, 300])
s.index = ['apple', 'banana', 'cherry']
print(s)

# Output:
# apple      100
# banana     200
# cherry     300
# dtype: int64

# Access by label
print(s['banana'])  # Output: 200
```

Listing 3: Custom string index

Warning

Index Labels Can Be Non-Unique!

Unlike dictionary keys, Series indices can have duplicates:

```
s.index = ['a', 'a', 'b']
print(s['a'])
# Returns BOTH values with index 'a':
# a    100
# a    200
```

This can cause unexpected behavior if you're not careful!

3 The DataFrame Data Structure

3.1 What is a DataFrame?

Definition:

DataFrame A **DataFrame** is a 2-dimensional labeled data structure—like an Excel spreadsheet or SQL table.

- Each **column** is a Series
- All columns share the same **index** (row labels)
- Different columns can have different data types

3.2 Creating a DataFrame

3.2.1 Method 1: From a List of Dictionaries (Row by Row)

Each dictionary represents one row; keys become column names:

```
data = [
    {'fruit': 'apple', 'color': 'red', 'price': 1.50},
    {'fruit': 'banana', 'color': 'yellow', 'price': 0.75},
    {'fruit': 'cherry', 'color': 'red', 'price': 3.00}
]

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

Listing 4: DataFrame from list of dictionaries

Output:

```
    fruit   color  price
0   apple     red   1.50
1  banana  yellow   0.75
2  cherry     red   3.00
```

3.2.2 Method 2: From a Dictionary of Lists (Column by Column)

Each key is a column name; the list becomes that column's values:

```
data = {
    'fruit': ['apple', 'banana', 'cherry'],
    'color': ['red', 'yellow', 'red'],
    'price': [1.50, 0.75, 3.00]
}

df = pd.DataFrame(data)
# Same result as above
```

Listing 5: DataFrame from dictionary of lists

Key Information

Which Method to Use?

- **List of dictionaries**: Useful when scraping data (you build up rows one at a time)
- **Dictionary of lists**: Useful when you already have column-wise data

3.3 DataFrame Attributes

```python
print(df.shape)     # (rows, columns)
# Output: (3, 3)

print(df.columns)  # Column names
# Output: Index(['fruit', 'color', 'price'], dtype='object')

print(df.index)     # Row labels
# Output: RangeIndex(start=0, stop=3, step=1)

print(df.dtypes)    # Data type of each column
# Output:
# fruit      object
# color      object
# price     float64
# dtype: object
```

Listing 6: DataFrame attributes

4 Loading Data from Files

4.1 Reading CSV Files

The most common way to get data into pandas:

```
# Read a CSV file into a DataFrame
df = pd.read_csv('data/survey_results.csv')

# Peek at the first few rows
df.head()
```

Listing 7: Reading a CSV file

Key Information

Other File Formats

Pandas can read many formats:

- `pd.read_excel('file.xlsx')` — Excel files
- `pd.read_json('file.json')` — JSON files
- `pd.read_html('url')` — Tables from web pages
- `pd.read_sql(query, connection)` — SQL databases
- `pd.read_clipboard()` — From your clipboard!

4.2 Inspecting Your Data

After loading data, always inspect it first:

```
# First 5 rows (default) or specify n
df.head()
df.head(2)

# Last 5 rows
df.tail()

# Shape: (rows, columns)
df.shape

# Column names
df.columns

# Data types and memory usage
df.info()

# Summary statistics for numeric columns
df.describe()

# Data types only
df.dtypes
```

Listing 8: Inspecting a DataFrame

5 Selecting Data

5.1 Selecting Columns

```python
# Single column (returns a Series)
df['fruit']

# Single column using dot notation (shortcut)
df.fruit  # Only works if column name has no spaces

# Multiple columns (returns a DataFrame)
df[['fruit', 'price']]
```

Listing 9: Selecting columns

Warning

Be Careful with Dot Notation

`df.fruit` is convenient but dangerous if:

- Column name has spaces (`df.my column` doesn't work)
- Column name matches a DataFrame method (`df.head` returns the method, not a column!)

The bracket notation `df['fruit']` is always safe.

5.2 The loc and iloc Accessors

Pandas provides two ways to select rows:

Definition:

loc vs iloc

- **loc**: Selection by **label** (what you see in the index)
- **iloc**: Selection by **integer position** (0, 1, 2, ...)

```python
# Create a DataFrame with a non-default index
df = pd.DataFrame({
    'name': ['Alice', 'Bob', 'Charlie'],
    'age': [25, 30, 35]
}, index=['a', 'b', 'c'])

# loc: by label
df.loc['b']         # Row with label 'b'
df.loc['a':'b']     # Rows 'a' through 'b' (inclusive!)
df.loc['a', 'age'] # Specific cell: row 'a', column 'age'

# iloc: by position
df.iloc[1]          # Second row (position 1)
df.iloc[0:2]        # First two rows (exclusive end!)
df.iloc[0, 1]       # Specific cell: row 0, column 1
```

Listing 10: Using loc and iloc

Important:

The Index Trap After filtering or sorting, row indices get scrambled:

```
df_sorted = df.sort_values('age', ascending=False)
print(df_sorted)
#       name  age
# c   Charlie   35
# b       Bob   30
# a     Alice   25

# Trying to get "first row" by position
df_sorted.iloc[0]  # Gets Charlie (correct)

# Trying by label 0
df_sorted.loc[0]   # ERROR! No label '0' exists
```

Solution: Use `.reset_index(drop=True)` after sorting/filtering.

6 Boolean Indexing (Filtering)

6.1 Creating Boolean Masks

A **boolean mask** is a Series of True/False values that you use to filter rows:

```
df = pd.DataFrame({
    'name': ['Alice', 'Bob', 'Charlie', 'Diana'],
    'age': [25, 30, 35, 28],
    'city': ['NYC', 'LA', 'NYC', 'LA']
})

# Create a mask: which rows have age > 27?
mask = df['age'] > 27
print(mask)
# 0    False
# 1     True
# 2     True
# 3     True
# dtype: bool
```

Listing 11: Creating boolean masks

6.2 Applying the Mask

Use the mask inside brackets to filter:

```
# Get rows where age > 27
df[mask]
# or directly:
df[df['age'] > 27]

#       name  age city
# 1      Bob   30   LA
# 2  Charlie   35  NYC
# 3    Diana   28   LA
```

Listing 12: Filtering with boolean masks

6.3 Combining Conditions

```
# Use & for AND, | for OR, ~ for NOT
# IMPORTANT: Each condition must be in parentheses!

# Age > 27 AND city is NYC
df[(df['age'] > 27) & (df['city'] == 'NYC')]

# Age > 30 OR city is LA
df[(df['age'] > 30) | (df['city'] == 'LA')]

```

```
# NOT in NYC
df[~(df['city'] == 'NYC')]
```

Listing 13: Combining conditions

Warning

Parentheses are Required!

Due to Python operator precedence, you must wrap each condition in parentheses:

```
# WRONG:
df[df['age'] > 27 & df['city'] == 'NYC']  # Error!

# CORRECT:
df[(df['age'] > 27) & (df['city'] == 'NYC')]
```

7 Common DataFrame Operations

7.1 Sorting

```python
# Sort by one column
df.sort_values('age')

# Sort descending
df.sort_values('age', ascending=False)

# Sort by multiple columns
df.sort_values(['city', 'age'])

# Reset index after sorting (usually what you want)
df.sort_values('age').reset_index(drop=True)
```

Listing 14: Sorting DataFrames

7.2 Value Counts

Count occurrences of unique values:

```python
# How many people in each city?
df['city'].value_counts()
# NYC    2
# LA     2

# Get unique values
df['city'].unique()
# array(['NYC', 'LA'], dtype=object)

# Number of unique values
df['city'].nunique()
# 2
```

Listing 15: Counting values

7.3 Group By and Aggregation

Definition:

GroupBy **GroupBy** splits data into groups based on a column's values, then applies an aggregation function to each group.

```python
# Average age by city
df.groupby('city')['age'].mean()
# city
# LA     29.0
# NYC    30.0
```

```
6
7  # Multiple aggregations
8  df.groupby('city').agg({
9      'age': ['mean', 'min', 'max'],
10     'name': 'count'
11 })
12
13 # Count rows per group
14 df.groupby('city').size()
```

Listing 16: GroupBy operations

7.4 Handling Missing Data

```python
# Check for missing values
df.isna()            # Boolean mask of NaN locations
df.isna().sum()      # Count NaNs per column

# Drop rows with any NaN
df.dropna()

# Fill NaN with a value
df.fillna(0)
df['column'].fillna('Unknown')

# Fill with column mean
df['age'].fillna(df['age'].mean())
```

Listing 17: Working with missing values

7.5 Renaming Columns

```python
# Rename specific columns
df.rename(columns={'old_name': 'new_name'})

# Rename all columns at once
df.columns = ['col1', 'col2', 'col3', 'col4']

# Make all column names lowercase
df.columns = df.columns.str.lower()
```

Listing 18: Renaming columns

8 Method Chaining

8.1 What is Method Chaining?

Instead of saving intermediate results to variables, you can chain multiple operations together:

```
# Without chaining (verbose)
df_filtered = df[df['age'] > 25]
df_sorted = df_filtered.sort_values('age')
df_top = df_sorted.head(3)

# With chaining (concise)
result = (df[df['age'] > 25]
          .sort_values('age')
          .head(3))
```

Listing 19: Method chaining example

Key Information

Benefits of Method Chaining

- More concise code
- No intermediate variables cluttering your namespace
- Reads like a pipeline: "Take df, filter it, sort it, take top 3"

Tip: Wrapping in parentheses lets you break across multiple lines for readability.

9 Practical Example: Survey Data

Let's walk through a realistic example using class survey data:

```python
# 1. Load the data
df = pd.read_csv('data/survey_raw.csv')

# 2. Inspect
print(df.shape)
print(df.columns)
df.head()

# 3. Clean column names (remove spaces, lowercase)
df.columns = ['timestamp', 'program', 'jupyter_exp',
              'python_exp', 'pandas_skill', 'os',
              'dark_mode', 'languages', 'dob']

# 4. Check data types
df.dtypes
df.info()

# 5. Convert dark_mode to boolean
df['dark_mode'] = df['dark_mode'] == 'Yes'

# 6. Filter to students with pandas skill > 3
skilled = df[df['pandas_skill'] > 3]

# 7. Count programs
df['program'].value_counts()

# 8. Average pandas skill by program
(df.groupby('program')['pandas_skill']
   .mean()
   .sort_values(ascending=False))

# 9. Save cleaned data
df.to_csv('data/survey_clean.csv', index=False)
```

Listing 20: Complete example workflow

10 Advanced: Exploding Lists

Sometimes a column contains comma-separated values that you want to split:

```python
# Sample data
df = pd.DataFrame({
    'name': ['Alice', 'Bob'],
    'languages': ['English, Spanish', 'English, Mandarin, French']
})

# Step 1: Split the string into a list
df['lang_list'] = df['languages'].str.split(', ')

# Step 2: Explode - each list item becomes its own row
df_exploded = df.explode('lang_list')

#     name                    languages    lang_list
# 0  Alice           English, Spanish      English
# 0  Alice           English, Spanish      Spanish
# 1    Bob  English, Mandarin, French      English
# 1    Bob  English, Mandarin, French     Mandarin
# 1    Bob  English, Mandarin, French       French

# Now you can count languages, filter, etc.
df_exploded['lang_list'].value_counts()
```

Listing 21: Exploding comma-separated values

11 Key Takeaways

Key Summary

Summary of Lecture 03: Pandas Fundamentals

Core Data Structures

- **Series**: 1D labeled array (values + index + name + dtype)
- **DataFrame**: 2D table of Series (columns share an index)

Creating DataFrames

- `pd.DataFrame(list_of_dicts)` — Row by row
- `pd.DataFrame(dict_of_lists)` — Column by column
- `pd.read_csv('file.csv')` — From file

Inspecting Data

- `df.head()`, `df.tail()`, `df.shape`
- `df.info()`, `df.describe()`, `df.dtypes`

Selecting Data

- Columns: `df['col']` or `df[['col1', 'col2']]`
- Rows by label: `df.loc['label']`
- Rows by position: `df.iloc[0]`
- Boolean filtering: `df[df['col'] > value]`

Common Operations

- Sort: `df.sort_values('col')`
- Count: `df['col'].value_counts()`
- Group: `df.groupby('col').agg('other': 'mean')`
- Reset: `df.reset_index(drop=True)`

Key Warnings

- `loc` uses labels; `iloc` uses positions
- After sorting/filtering, indices get scrambled—use `reset_index()`
- Use parentheses around each condition when combining with `&`/`|`
- Mixed types become `object` dtype (slow!)