Lecture 16 Handout

pandas Fundamentals Part 1: Series

The Foundation of Data Analysis

INF 605 - Introduction to Programming - Python

Prof. Rongyu Lin
Quinnipiac University
School of Computing and Engineering

Fall 2025

Required Reading

Textbook: Chapter 7, Section 7.14.1 (pandas Series)
Reference Notebooks: `ch07/07_14.01.ipynb` for Series fundamentals

Learning Objectives

By the end of this lecture, you will be able to:

1. **Explain the role of pandas** in Python's data science ecosystem and why Series are superior to NumPy arrays for labeled data
2. **Create pandas Series** from lists, dictionaries, scalars, and NumPy arrays with custom indices
3. **Access Series elements** using integer positions, custom index labels, and .loc[]/.iloc[] accessors
4. **Apply statistical methods** including count(), mean(), min(), max(), std(), and describe()
5. **Work with string Series** using the .str accessor for vectorized string operations
6. **Handle Series attributes** including dtype, values, index, and name
7. **Perform Series arithmetic** with automatic index alignment
8. **Build real-world applications** combining multiple Series operations

Prerequisites Review

Building on Your Complete Foundation:

From Lectures 1-11, you mastered all Python fundamentals: variables, data types, control structures, functions, lists, tuples, dictionaries, file I/O, and object-oriented programming with classes, inheritance, and polymorphism.

From Lectures 12-13, you mastered NumPy arrays with all their power: array creation and manipulation, indexing and slicing including boolean indexing, element-wise operations and vectorization, array aggregations and statistical functions, broadcasting rules, 2D and 3D array operations, matrix operations and linear algebra basics, and the tremendous performance benefits NumPy provides over Python lists.

You understand that NumPy arrays are incredibly fast for numerical operations but lack named indices (arrays use only integer positions), built-in handling of missing data, convenient data analysis methods, and alignment by index during operations.

Transformation Goal: Evolve from **using NumPy arrays with integer-based indexing** to **leveraging pandas Series with custom indices, automatic data alignment, missing data handling, and rich statistical methods for professional data analysis**.

1 Part 1: Introduction to pandas and Series

1.1 Why pandas? The Data Analysis Problem

Think about how you organize information in real life. When you write down a shopping list, you don't just write item names - you might write quantities next to each item. When a teacher records grades, they write student names alongside scores. In both cases, you have values (quantities, grades) paired with meaningful labels (items, names). This is how humans naturally think about data.

NumPy arrays, which we mastered in previous lectures, are like numbered lists - item 0, item 1, item 2. They're fast and efficient for computations, but they force you to remember that "position 3 is Alice" and "position 7 is the March sales data." This works for small datasets, but it's not intuitive, and it's error-prone when your data has hundreds or thousands of entries.

pandas Series solve this fundamental problem by allowing you to attach meaningful labels to your data. A Series is essentially a NumPy array with a built-in index that can use any labels you want - names, dates, product codes, anything that makes sense for your data. When you ask for `grades['Alice']`, you get Alice's grade directly, without having to remember that Alice is at position 3. This simple feature transforms data analysis from a technical exercise into an intuitive process.

pandas (Python Data Analysis Library) has become the industry standard for data manipulation in Python. Every data scientist, data analyst, and machine learning engineer uses pandas daily. It provides the same intuitive organization as spreadsheets (like Excel) but with the full power of Python programming. Throughout this lecture, you'll see how Series combine NumPy's computational speed with the clarity and convenience that professional data analysis requires.

1.2 Installing and Importing pandas

pandas is not part of Python's standard library and needs to be installed separately. It's included in Anaconda distributions, or you can install it using pip. The standard convention is to import pandas with the alias `pd`, which you'll see in all professional code.

```python
# Installation (if needed)
# Run in terminal: pip install pandas

# Standard import convention
import pandas as pd  # 'pd' is the universal alias

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

Always use `import pandas as pd` - this is the established convention in the entire data science community. When you see `pd.Series` or `pd.DataFrame` in code examples online, documentation, or books, you'll immediately understand what library is being used.

1.3 Creating Your First Series

Let's create our first pandas Series and explore its structure. A Series is displayed as two columns: the index on the left (labels for each element) and the values on the right (the actual data). This two-column display is your first hint that Series are more than just arrays - they're labeled data structures designed for analysis.

```python
import pandas as pd

# Create Series from list with default integer indices
grades = pd.Series([87, 92, 88, 95])
print(grades)
print(f"\nType: {type(grades)}")
print(f"Data type: {grades.dtype}")
```

When you run this code, you'll see output like:

```
0    87
1    92
2    88
3    95
dtype: int64
```

Notice the two-column format: the left column shows indices (0, 1, 2, 3), and the right column shows values (87, 92, 88, 95). The `dtype: int64` at the bottom tells you the data type - pandas automatically inferred that these are integers. This is very similar to NumPy's `dtype` attribute, because underneath every Series is a NumPy array providing the computational power.

The difference between a Series and a NumPy array becomes clear when we add custom indices in the next section.

2 Part 2: Series Creation Methods

2.1 Series with Custom Indices

The true power of Series emerges when we use custom indices. Instead of relying on positions (0, 1, 2, 3), we can use meaningful labels that describe what each data point represents. This makes your code readable and self-documenting.

```python
# Create Series with student names as indices
grades = pd.Series([87, 92, 88, 95],
                   index=['Alice', 'Bob', 'Charlie', 'Diana'])
print(grades)
print(f"\nAccess Alice's grade: {grades['Alice']}")
```

Now the output shows:

```
Alice      87
Bob        92
Charlie    88
Diana      95
dtype: int64
```

See how much more readable this is? You can immediately see that Alice scored 87, Bob scored 92, and so on. No need to remember positions or create separate name lists. The index and values travel together as a unit, making your data self-describing. This is exactly how you'd organize data in a spreadsheet, but now you have all the power of Python programming at your fingertips.

2.2 Series from Dictionaries

Python dictionaries naturally map keys to values, making them perfect for creating Series. When you create a Series from a dictionary, the dictionary keys automatically become the index, and the dictionary values become the Series values. This is incredibly convenient for data that already has a key-value structure.

```python
# Dictionary naturally maps to Series
temperatures = {
    'Monday': 72,
    'Tuesday': 68,
    'Wednesday': 75,
    'Thursday': 71,
    'Friday': 69
}

temp_series = pd.Series(temperatures)
print(temp_series)
print(f"\nWednesday temperature: {temp_series['Wednesday']}F")
```

This approach is perfect when you're working with data that comes in dictionary form - perhaps from JSON files, API responses, or configuration files. The conversion to Series is seamless and preserves the meaningful labels from your dictionary keys.

2.3 Series from Scalars

Sometimes you need to create a Series with the same value repeated multiple times - perhaps for initialization, creating default values, or mathematical operations. pandas lets you create a Series from a scalar (single value) by specifying what indices you want.

```python
# Create Series with repeated values
default_score = pd.Series(70, index=['Alice', 'Bob', 'Charlie', 'Diana'])
print(default_score)

# Useful for initialization or comparisons
baseline = pd.Series(0, index=range(5))
print(f"\nBaseline: {baseline}")
```

The scalar value (70 or 0) is broadcast to every position in the Series. The length of the Series is determined by the length of the index you provide. This is particularly useful when you need to create a template Series or when performing threshold comparisons.

3 Part 3: Accessing Series Elements

3.1 Integer-Based and Label-Based Indexing

pandas Series support multiple ways to access elements. You can use integer positions (just like lists and NumPy arrays) or custom labels (the unique pandas feature). Understanding when to use each method is crucial for writing clear, correct code.

```python
grades = pd.Series([87, 92, 88, 95],
                   index=['Alice', 'Bob', 'Charlie', 'Diana'])
```

```python

# Integer position access (like lists)
print(f"First grade: {grades[0]}")      # 87
print(f"Last grade: {grades[-1]}")      # 95

# Label-based access (pandas feature)
print(f"Alice's grade: {grades['Alice']}")     # 87
print(f"Charlie's grade: {grades['Charlie']}")  # 88
```

When you use square brackets with an integer (`grades[0]`), you're accessing by position. When you use square brackets with a string (`grades['Alice']`), you're accessing by index label. This flexibility is convenient, but it can sometimes be ambiguous - what if your index labels are also integers?

That's where .loc[] and .iloc[] come in.

3.2 Using .loc[] and .iloc[] for Explicit Access

To eliminate any ambiguity, pandas provides two explicit accessors: `.loc[]` for label-based access and `.iloc[]` for integer position-based access. Professional pandas code uses these accessors because they make your intent crystal clear.

```python
grades = pd.Series([87, 92, 88, 95],
                   index=['Alice', 'Bob', 'Charlie', 'Diana'])

# .iloc[] - always uses integer positions
print(f"First element: {grades.iloc[0]}")    # 87
print(f"Last element: {grades.iloc[-1]}")    # 95

# .loc[] - always uses index labels
print(f"Alice's grade: {grades.loc['Alice']}")     # 87
print(f"Diana's grade: {grades.loc['Diana']}")     # 95

# Slicing with .iloc[] (excludes endpoint)
print(f"\nFirst two: {grades.iloc[0:2]}")
# Output: Alice 87, Bob 92

# Slicing with .loc[] (includes endpoint!)
print(f"\nAlice to Charlie: {grades.loc['Alice':'Charlie']}")
# Output: Alice 87, Bob 92, Charlie 88
```

Notice the crucial difference: `.iloc[0:2]` excludes position 2 (standard Python slicing behavior), but `.loc['Alice':'Charlie']` includes 'Charlie' because it's label-based slicing. This distinction matters for correctness. Best practice: use `.loc[]` and `.iloc[]` in your code to make your intent explicit.

4 Part 4: Statistical Analysis with Series

4.1 Building on NumPy Statistical Methods

Connection to Lecture 13: Remember all the statistical methods you learned for NumPy arrays in Lecture 13 - `mean()`, `std()`, `min()`, `max()`, `sum()`, and `var()`? Great news: pandas Series inherited all these methods! They work exactly the same way, but with a crucial pandas advantage: they preserve and work with your meaningful index labels.

```python
# Student test scores with names as indices
scores = pd.Series([85, 92, 78, 90, 88, 95, 82, 87, 91, 86],
                   index=['Alice', 'Bob', 'Charlie', 'Diana', 'Eve',
                          'Frank', 'Grace', 'Henry', 'Iris', 'Jack'])

```

```python
# All your familiar statistical methods work!
print(f"Average score: {scores.mean():.2f}")  # From Lecture 13
print(f"Standard deviation: {scores.std():.2f}")  # From Lecture 13
print(f"Score range: {scores.max() - scores.min()}")  # From Lecture 13
```

Key Insight: These are the same statistical concepts from Lecture 13, now applied to labeled data. The calculations work identically, but pandas makes the results more interpretable by maintaining the association between values and their labels.

4.2 pandas Special Feature: Index-Returning Methods

Here is where pandas truly shines beyond NumPy. While NumPy's `argmax()` and `argmin()` return integer positions, pandas provides `idxmax()` and `idxmin()` that return **index labels** - making your analysis immediately interpretable without looking up positions!

```python
# Find extremes - pandas returns meaningful labels!
print(f"\nHighest score: {scores.max()}")
print(f"Student with highest score: {scores.idxmax()}")  # Returns 'Frank'!

print(f"\nLowest score: {scores.min()}")
print(f"Student with lowest score: {scores.idxmin()}")  # Returns 'Charlie'!
```

Compare with NumPy approach:

```python
# NumPy way (Lecture 13): returns position
# arr.argmax() returns 5, then you look up names[5] = 'Frank'

# pandas way: returns label directly
# scores.idxmax() returns 'Frank' - no lookup needed!
```

This is the pandas advantage in action: immediate, human-readable results. You get "Frank scored highest" instead of "position 5 scored highest, let me check who that is..."

4.3 The describe() Method - All Lecture 13 Stats in One Call

The `describe()` method combines all the statistical methods from Lecture 13 into one comprehensive summary. Instead of calling `mean()`, `std()`, `min()`, `max()`, and calculating quartiles separately, `describe()` computes everything at once.

```python
# One method call for complete statistical analysis
scores = pd.Series([85, 92, 78, 90, 88, 95, 82, 87, 91, 86],
                   index=['Alice', 'Bob', 'Charlie', 'Diana', 'Eve',
                          'Frank', 'Grace', 'Henry', 'Iris', 'Jack'])

print("Complete Statistical Summary:")
print(scores.describe())
```

Output shows eight key statistics:

```
count    10.000000    # Number of data points
mean     87.400000    # Average (from Lecture 13)
std       5.039841    # Standard deviation (from Lecture 13)
min      78.000000    # Minimum (from Lecture 13)
25%      85.750000    # First quartile (like percentile from Lecture 13)
50%      87.500000    # Median - middle value
75%      90.750000    # Third quartile
max      95.000000    # Maximum (from Lecture 13)
```

Why describe() matters: This is typically the *first* thing data scientists run on any new dataset - it reveals data quality issues (impossible values?), distribution shape (quartiles), and basic characteristics (mean, spread) in one glance. Think of it as a comprehensive health checkup for your data, combining all your Lecture 13 statistical knowledge.

5 Part 5: Working with String Series

5.1 The .str Accessor for Vectorized String Operations

When your Series contains strings, you need special methods to manipulate them. You can't just call `.upper()` on a Series - that would try to uppercase the entire Series object, not each string element. pandas provides the `.str` accessor that applies string methods to each element automatically, using vectorized operations under the hood for performance.

Think of the `.str` accessor like this: if you have a class of students and want everyone to raise their hand, you don't go to each student individually - you just say "class, raise your hands!" The `.str` accessor is that class-wide command for string operations.

```python
# Student names Series
names = pd.Series(['Alice Johnson', 'Bob Smith', 'Charlie Brown',
                   'Diana Prince'])

# String methods require .str accessor
upper_names = names.str.upper()
print("Uppercase names:")
print(upper_names)

# Extract first names
first_names = names.str.split().str[0]
print(f"\nFirst names: {first_names}")

# Check for pattern
has_brown = names.str.contains('Brown')
print(f"\nNames containing 'Brown': {names[has_brown]}")
```

The `.str` accessor gives you access to almost all Python string methods: `.upper()`, `.lower()`, `.split()`, `.replace()`, `.contains()`, `.startswith()`, and many more. These operations happen vectorized (all at once) rather than looping through each string, making them both faster and cleaner to write.

5.2 Common String Operations

String manipulation is crucial for data cleaning and analysis. Email addresses need standardization, product codes need parsing, text data needs filtering. The `.str` accessor makes these operations straightforward and efficient.

```python
# Product codes that need cleaning
products = pd.Series(['WIDGET-A', 'gadget-b', 'WIDGET-C', 'tool-d'])

# Standardize to lowercase
standardized = products.str.lower()
print("Standardized codes:")
print(standardized)

# Filter for widgets only
widgets = products[products.str.contains('WIDGET')]
print(f"\nWidget products: {widgets}")

# Replace text
updated = products.str.replace('WIDGET', 'ITEM')
print(f"\nUpdated codes: {updated}")

```

```python
# Extract parts (after hyphen)
suffix = products.str.split('-').str[1]
print(f"\nProduct types: {suffix}")
```

These operations are fundamental to data preprocessing. Real-world data is messy - product codes have inconsistent capitalization, emails have typos, text has extra spaces. The `.str` methods let you clean and standardize this data efficiently. Notice how `.str.contains()` creates a boolean mask you can use for filtering - this combines string operations with boolean indexing, showing how pandas features work together.

6 Part 6: Series Arithmetic and Index Alignment

6.1 Element-wise Operations and Broadcasting

Series support all the arithmetic operations you learned with NumPy arrays - addition, subtraction, multiplication, division. These operations are element-wise and vectorized, meaning they apply to entire Series at once, just like NumPy arrays. Broadcasting with scalars works exactly as you'd expect.

```python
# Quiz scores for students
quiz1 = pd.Series([85, 92, 88, 90],
                  index=['Alice', 'Bob', 'Charlie', 'Diana'])
quiz2 = pd.Series([87, 89, 91, 93],
                  index=['Alice', 'Bob', 'Charlie', 'Diana'])

# Element-wise addition
total_score = quiz1 + quiz2
print("Total scores:")
print(total_score)

# Broadcasting with scalar
weighted_quiz1 = quiz1 * 0.6  # Quiz 1 worth 60%
weighted_quiz2 = quiz2 * 0.4  # Quiz 2 worth 40%
final_score = weighted_quiz1 + weighted_quiz2
print(f"\nWeighted final scores:")
print(final_score)
```

These operations maintain the index, so the result is still a labeled Series. You can see which student has which total score without remembering positions. The arithmetic is straightforward when indices match perfectly between Series.

6.2 Index Alignment - pandas Magic

Here's where pandas reveals its true power: automatic index alignment. When you add two Series, pandas doesn't just add element 0 to element 0 based on position (like NumPy). Instead, it aligns by index labels, matching 'Alice' with 'Alice', 'Bob' with 'Bob', regardless of their positions.

```python
# Series with different orders
quiz1 = pd.Series([85, 92, 88],
                  index=['Alice', 'Bob', 'Charlie'])
quiz2 = pd.Series([90, 87, 89],
                  index=['Charlie', 'Alice', 'Bob'])

# pandas aligns by index automatically!
total = quiz1 + quiz2
print("Total scores (aligned by name):")
print(total)
# Alice: 85 + 87 = 172
# Bob: 92 + 89 = 181
# Charlie: 88 + 90 = 178
```

Notice that `quiz1` and `quiz2` have students in different orders. NumPy would add based on positions: 85+90, 92+87, 88+89 (wrong!). pandas aligns by index labels: Alice gets 85+87, Bob gets 92+89, Charlie gets 88+90 (correct!). This automatic alignment prevents a huge category of errors in data analysis.

6.3 Handling Missing Data

What happens when indices don't match completely? pandas introduces the concept of missing data, represented by NaN (Not a Number). This is pandas' way of saying "there's no data here for this index." It's not an error - it's information that tells you about your data's structure.

```
# Students who took different quizzes
quiz1 = pd.Series([85, 92, 88],
                  index=['Alice', 'Bob', 'Charlie'])
quiz2 = pd.Series([87, 89, 91],
                  index=['Bob', 'Charlie', 'Diana'])

# Addition with mismatched indices
result = quiz1 + quiz2
print("Combined scores:")
print(result)
# Alice has NaN (only in quiz1)
# Diana has NaN (only in quiz2)

# Detect missing data
print(f"\nHas missing data: {result.isna()}")
print(f"\nHas valid data: {result.notna()}")

# Most operations ignore NaN
print(f"Mean (ignoring NaN): {result.mean():.2f}")
```

NaN appears when Alice (only in quiz1) or Diana (only in quiz2) can't be matched. This is the correct behavior - pandas tells you there's incomplete data rather than making up values or causing an error. Most statistical methods (`mean()`, `sum()`, etc.) automatically ignore NaN values, which is usually what you want. You can detect NaN with `.isna()` and `.notna()`, giving you full control over how to handle missing data.

7 Part 7: Real-World Applications

7.1 Application 1: Grade Analysis System

Let's build a complete grade analysis system that demonstrates how Series operations combine to solve real problems. This example shows how you'd actually use Series in practice - creating data, performing calculations, filtering based on conditions, and generating insights.

```
import pandas as pd

# Student grades
grades = pd.Series([87, 92, 78, 95, 85, 88, 91, 76, 89, 94],
                   index=['Alice', 'Bob', 'Charlie', 'Diana', 'Eve',
                          'Frank', 'Grace', 'Henry', 'Iris', 'Jack'])

# Class statistics
print("Class Statistics:")
print(f"Average grade: {grades.mean():.1f}")
print(f"Median grade: {grades.median():.1f}")
print(f"Standard deviation: {grades.std():.1f}")
```

```

# Find top performers
top_3 = grades.nlargest(3)
print(f"\nTop 3 students:")
print(top_3)

# Students needing help (below 80)
struggling = grades[grades < 80]
print(f"\nStudents below 80:")
print(struggling)

# Grade distribution
print(f"\nGrade Distribution:")
print(f"A (90+): {(grades >= 90).sum()} students")
print(f"B (80-89): {((grades >= 80) & (grades < 90)).sum()}")
print(f"C (70-79): {((grades >= 70) & (grades < 80)).sum()}")
```

This example demonstrates real data analysis workflow: load data into Series with meaningful indices, calculate summary statistics to understand overall performance, identify outliers (top performers and struggling students), filter data based on conditions, and generate summary reports. Each step builds on Series features we've learned.

7.2 Application 2: Temperature Analysis

Environmental data often comes with time-based indices. This example shows how Series work with dates as indices, a common pattern in time-series analysis.

```
# Weekly temperature data
temperatures = pd.Series([72, 68, 75, 71, 69, 76, 73],
                        index=['Mon', 'Tue', 'Wed', 'Thu',
                               'Fri', 'Sat', 'Sun'])

# Basic analysis
print("Temperature Statistics:")
print(f"Average: {temperatures.mean():.1f}F")
print(f"Range: {temperatures.max() - temperatures.min():.1f}F")

# Find extreme days
hottest = temperatures.idxmax()  # Returns index of max
coldest = temperatures.idxmin()  # Returns index of min
print(f"\nHottest: {hottest} ({temperatures.max()}F)")
print(f"Coldest: {coldest} ({temperatures.min()}F)")

# Days above/below average
avg = temperatures.mean()
warm_days = temperatures[temperatures > avg]
cool_days = temperatures[temperatures <= avg]
print(f"\nDays above average: {warm_days}")
print(f"Days below average: {cool_days}")
```

Notice how `idxmax()` and `idxmin()` return the index labels (day names) rather than positions. This is incredibly useful - you get ”Wednesday” instead of ”2”, making your analysis immediately interpretable. This is the power of labeled data.

Summary and Best Practices

Key Takeaways:

1. pandas Series combine NumPy's computational power with meaningful labels and data analysis convenience

2. Series can be created from lists, dictionaries, scalars, and NumPy arrays with custom indices
3. Use .loc[] for label-based access and .iloc[] for position-based access to eliminate ambiguity
4. Statistical methods (mean, std, describe) provide instant insight into data characteristics
5. The .str accessor enables vectorized string operations on text data
6. Index alignment automatically matches data by labels during operations
7. NaN represents missing data and is handled gracefully by most operations
8. Series are the foundation for DataFrames (coming in Lecture 17)

Common Pitfalls to Avoid

- Using regular string methods instead of .str accessor (causes errors)
- Confusing .loc[] (label-based) with .iloc[] (position-based) slicing behavior
- Not understanding that NaN is information about missing data, not an error
- Assuming Series operations align by position (they align by index!)
- Writing loops over Series instead of using vectorized methods
- Forgetting that .loc[] slicing includes the endpoint while .iloc[] excludes it

Practice Exercises

Exercise 1: Product Sales Analysis
Create a Series of monthly sales for 5 products. Calculate total sales, average sales, best and worst performing products, and products above/below average.

Exercise 2: Student Grade Management
Given a Series of student grades, convert letter grades to numeric scores, calculate class statistics, identify honor roll students (top 20%), and generate a grade distribution report.

Exercise 3: Temperature Data Cleaning
Load temperature data with some missing values (NaN). Detect missing data, fill missing values with the average, identify days with extreme temperatures, and calculate weekly trends.

Next Lecture Preview

In Lecture 17, we'll extend Series to two dimensions with pandas DataFrames - the spreadsheet-like structure that enables multi-variable data analysis. You'll learn how DataFrames are collections of aligned Series columns, how to perform operations across rows and columns, boolean indexing with multiple conditions, and how to build complete data analysis pipelines. Everything you learned about Series will transfer directly to working with DataFrame columns.