Hello

Pandas iterrows() - Complete Guide

Master row iteration in pandas DataFrames

Download This Guide as PDF

What is iterrows()?

`iterrows()` is a pandas method that allows you to iterate through a DataFrame row by row, returning both the index and the row data as a Series for each iteration.

Copy

```
// Basic Syntax for index, row in dataframe.iterrows(): # index: the row index (int, string, etc.) # row: pandas Series containing the row data # access data using row['column_name']
```

Basic Usage

Sample DataFrame

Name	Age	City	Salary
Alice	25	New York	50000
Bob	30	London	60000
Charlie	35	Tokyo	70000

Simple Iteration Example

Copy

```
import pandas as pd df = pd.DataFrame({ 'Name': ['Alice', 'Bob',
'Charlie'], 'Age': [25, 30, 35], 'City': ['New York', 'London',
'Tokyo'], 'Salary': [50000, 60000, 70000] }) for index, row in
df.iterrows(): print(f"Index: {index}") print(f"Name:
{row['Name']}, Age: {row['Age']}") print(f"City: {row['City']},
Salary: ${row['Salary']}") print("---")
```

Output:
Index: 0
Name: Alice, Age: 25
City: New York, Salary: $50000

Index: 1
Name: Bob, Age: 30
City: London, Salary: $60000

Index: 2
Name: Charlie, Age: 35
City: Tokyo, Salary: $70000

Accessing Data in iterrows()

Different Ways to Access Column Values

Copy

```
for index, row in df.iterrows(): # Method 1: Using column name
as key name = row['Name'] # Method 2: Using dot notation (if
column name is valid) age = row.Age # Method 3: Using get()
method (safe) city = row.get('City', 'Unknown') print(f"{name} -
{age} - {city}")
```

Working with Specific Data Types

Copy

```
for index, row in df.iterrows(): # Convert to appropriate types
if needed name = str(row['Name']) age = int(row['Age']) salary =
float(row['Salary']) # Perform calculations monthly_salary =
salary / 12 print(f"{name}: ${monthly_salary:.2f} per month")
```

Practical Examples

Example 1: Data Processing with Conditions

Copy

```
# Calculate bonuses based on conditions results = [] for index,
row in df.iterrows(): if row['Age'] > 30: bonus = row['Salary']
* 0.15 # 15% bonus for over 30 else: bonus = row['Salary'] *
0.10 # 10% bonus for others results.append({ 'Name':
row['Name'], 'Bonus': bonus, 'Total_Compensation': row['Salary']
+ bonus }) bonus_df = pd.DataFrame(results) print(bonus_df)
```

Example 2: Filtering and Collecting Data

Copy

```
# Find people in specific cities with high salaries high_earners
= [] for index, row in df.iterrows(): if row['Salary'] > 55000
and row['City'] in ['London', 'Tokyo']: high_earners.append({
'Name': row['Name'], 'City': row['City'], 'Salary':
row['Salary'] }) print("High earners:", high_earners)
```

Example 3: Real-world ATECO Data Processing

Copy

```
def analyze_ateco_codes(df_ateco): """Analyze ATECO codes using
iterrows()""" analysis = { 'main_categories': [],
'subcategories': [], 'empty_descriptions': [] } for index, row
```

```
in df_ateco.iterrows(): codice = str(row['Codice']).strip()
descrizione = str(row['Codice_desc']).strip() # Categorize by
code length if len(codice) == 2:
analysis['main_categories'].append(codice) elif len(codice) > 2:
analysis['subcategories'].append(codice) # Check for empty
descriptions if not descrizione:
analysis['empty_descriptions'].append(index) return analysis #
Usage # result = analyze_ateco_codes(df_Extra)
```

Performance Considerations

⚠️ **Performance Warning:**
`iterrows()` can be slow for large DataFrames. Always consider vectorized operations first.

Comparison: iterrows() vs Vectorized Operations

Copy

```
# SLOW: Using iterrows() for large datasets results = [] for
index, row in large_df.iterrows():
results.append(complex_calculation(row)) # FAST: Vectorized
operations (when possible) results =
large_df['column'].apply(lambda x: complex_calculation(x)) # OR
results = complex_calculation(large_df['column'])
```

When to Use iterrows()

- Small to medium-sized DataFrames
- Complex row-wise logic that can't be vectorized
- When you need both index and row data
- Data validation and cleaning tasks

When to Avoid iterrows()

- Large DataFrames (100,000+ rows)

- Simple mathematical operations
- Operations that can be done with apply()
- Performance-critical applications

Common Pitfalls and Solutions

Pitfall 1: Modifying Data Incorrectly

```
Copy
# WRONG: This doesn't modify the original DataFrame for index, row in df.iterrows(): row['Salary'] = row['Salary'] * 1.1 # Only modifies the copy! # CORRECT: Use .at or .loc for index, row in df.iterrows(): df.at[index, 'Salary'] = row['Salary'] * 1.1 # OR: Create new DataFrame new_data = [] for index, row in df.iterrows(): new_row = row.copy() new_row['Salary'] = row['Salary'] * 1.1 new_data.append(new_row) df_updated = pd.DataFrame(new_data)
```

Pitfall 2: Assuming Column Data Types

```
Copy
# Always check/convert data types for index, row in df.iterrows(): # Safe approach name = str(row['Name']) age = int(row['Age']) if pd.notna(row['Age']) else 0 salary = float(row['Salary']) if pd.notna(row['Salary']) else 0.0
```

Pitfall 3: Ignoring Missing Values

```
Copy
# Handle missing values properly for index, row in df.iterrows(): if pd.isna(row['Salary']): print(f"Missing salary for {row['Name']} at index {index}") continue # Skip or handle appropriately # Process non-missing values processed_salary = row['Salary'] * 1.1
```

Advanced Techniques

Using with enumerate()

Copy

```
# Add a counter with enumerate for counter, (index, row) in enumerate(df.iterrows()): print(f"Processing row {counter + 1}: {row['Name']}") if counter >= 5: # Limit to first 6 rows break
```

Combining with Filtering

Copy

```
# Filter first, then iterate (more efficient) filtered_df = df[df['Age'] > 28] print("People over 28:") for index, row in filtered_df.iterrows(): print(f" - {row['Name']} ({row['Age']} years)")
```

Error Handling in Loops

Copy

```
# Robust error handling for index, row in df.iterrows(): try: # Potentially problematic operations complex_result = some_risky_operation(row) results.append(complex_result) except Exception as e: print(f"Error processing row {index}: {e}") # Log error or handle appropriately error_rows.append(index)
```

Best Practices Summary

✅ **DO:**

- Use for small to medium DataFrames
- Handle missing values with pd.isna()
- Convert data types explicitly

- Use .at or .loc for modifications
- Add error handling for robustness

❌ **DON'T:**

- Use for large DataFrames (>100K rows)
- Modify the row Series directly
- Assume data types
- Ignore performance implications
- Use when vectorized operations are possible

Performance Tips

1. **Pre-filter** your DataFrame before iterating
2. **Use itertuples()** for better performance when you don't need the index
3. **Batch process** when possible
4. **Consider alternatives** like apply() with axis=1
5. **Profile your code** to identify bottlenecks

Alternative Methods

itertuples() - Faster Alternative

Copy

```
# Faster than iterrows(), uses namedtuples for row in df.itertuples(): print(f"Name: {row.Name}, Age: {row.Age}") # Note: row.Index for index, not row.index
```

apply() - Functional Approach

Copy

```
# Apply function to each row def process_row(row): return f"{row['Name']} earns ${row['Salary']}" results = df.apply(process_row, axis=1)
```

Vectorized Operations - Best Performance

Copy

```
# Always prefer vectorized operations when possible
df['Monthly_Salary'] = df['Salary'] / 12 df['Bonus'] =
df['Salary'] * 0.1
```

World!