Day 2, Session 2: Data Manipulation

Three-Day Data Analysis with Python Course

Session Overview

1. **Data Cleaning**
2. **Data Transformation**
3. **Basic Aggregations**

Part 1: Data Cleaning

Why Data Cleaning?

Real-world data is messy:

- Missing values
- Duplicate records
- Incorrect data types
- Inconsistent formatting
- Outliers and errors

Spend 80% of your time cleaning, 20% analyzing.

Data cleaning is not glamorous, but it's essential!

Missing Values in Pandas

Missing data is represented as `NaN` (Not a Number) or `None`.

Common causes:

- Data not collected
- Data lost or corrupted
- Not applicable for that record
- Merge/join operations

Pandas provides powerful tools to handle missing data.

Detecting Missing Values

```python
# Check for missing values
customers.isna()        # Returns DataFrame of True/False
customers.isnull()      # Same as isna()

# Count missing values per column
customers.isna().sum()

# Total missing values
customers.isna().sum().sum()

# Percentage of missing values
(customers.isna().sum() / len(customers)) * 100
```

Detecting Missing Values (Example)

```python
import pandas as pd

# Sample data with missing values
data = {
    'name': ['Alice', 'Bob', None, 'David', 'Eve'],
    'age': [25, 30, 35, None, 28],
    'city': ['NYC', 'LA', 'Chicago', 'NYC', None],
    'salary': [50000, 60000, 55000, 65000, 52000]
}
df = pd.DataFrame(data)

print("Missing values per column:")
print(df.isna().sum())
```

Handling Missing Values: Strategy 1 - Drop

Remove rows or columns with missing data:

```
# Drop rows with ANY missing values
df_clean = df.dropna()

# Drop rows where ALL values are missing
df_clean = df.dropna(how='all')

# Drop rows with missing values in specific columns
df_clean = df.dropna(subset=['age', 'city'])

# Drop columns with missing values
df_clean = df.dropna(axis=1)
```

Be careful: You might lose valuable data!

Handling Missing Values: Strategy 2 - Fill

Replace missing values with something meaningful:

```python
# Fill with a constant value
df['age'].fillna(0)

# Fill with mean (for numerical columns)
df['age'].fillna(df['age'].mean())

# Fill with median
df['age'].fillna(df['age'].median())

# Fill with mode (most common value)
df['city'].fillna(df['city'].mode()[0])

# Forward fill (use previous value)
df['age'].fillna(method='ffill')
```

Handling Missing Values (Example)

```python
# Start with our messy data
print("Original data:")
print(df)
print(f"\nMissing values:\n{df.isna().sum()}")

# Strategy: Fill age with median, city with 'Unknown'
df_clean = df.copy()
df_clean['age'].fillna(df['age'].median(), inplace=True)
df_clean['city'].fillna('Unknown', inplace=True)
df_clean['name'].fillna('Anonymous', inplace=True)

print("\nCleaned data:")
print(df_clean)
print(f"\nMissing values:\n{df_clean.isna().sum()}")
```

Removing Duplicates

Duplicate records can skew analysis:

```
# Check for duplicates
df.duplicated()              # Returns True/False for each row
df.duplicated().sum()        # Count duplicates

# Remove duplicates
df_unique = df.drop_duplicates()

# Remove duplicates based on specific columns
df_unique = df.drop_duplicates(subset=['name', 'email'])

# Keep first occurrence (default)
df_unique = df.drop_duplicates(keep='first')

# Keep last occurrence
df_unique = df.drop_duplicates(keep='last')
```

Removing Duplicates (Example)

```python
# Data with duplicates
transactions = pd.DataFrame({
    'transaction_id': [1, 2, 3, 2, 4, 5, 3],
    'customer': ['Alice', 'Bob', 'Carol', 'Bob', 'David', 'Eve', 'Carol'],
    'amount': [100, 150, 200, 150, 120, 180, 200]
})

print("Original transactions:")
print(transactions)
print(f"\nDuplicates: {transactions.duplicated().sum()}")

# Remove duplicates based on transaction_id
clean_transactions = transactions.drop_duplicates(subset=['transaction_id'])
print("\nCleaned transactions:")
print(clean_transactions)
```

String Operations with `.str` Accessor

Pandas provides vectorized string operations:

```
# Convert to lowercase/uppercase
df['name'].str.lower()
df['name'].str.upper()

# Strip whitespace
df['name'].str.strip()

# Replace strings
df['city'].str.replace('NYC', 'New York City')

# Check if contains substring
df['email'].str.contains('@gmail.com')

# Split strings
df['name'].str.split(' ')
```

String Operations (Example)

```python
# Messy customer data
customers = pd.DataFrame({
    'name': [' Alice ', 'bob', 'CAROL ', ' david'],
    'email': ['ALICE@GMAIL.COM', 'bob@yahoo.com', 'carol@GMAIL.COM', 'david@outlook.com'],
    'phone': ['123-456-7890', '234.567.8901', '345 678 9012', '456-567-8901']
})

print("Original:")
print(customers)

# Clean up
customers['name'] = customers['name'].str.strip().str.title()
customers['email'] = customers['email'].str.lower()
customers['phone'] = customers['phone'].str.replace('[.-\\s]', '', regex=True)

print("\nCleaned:")
print(customers)
```

Type Conversion

Ensure columns have the correct data type:

```python
# Convert to numeric
df['age'] = pd.to_numeric(df['age'], errors='coerce')

# Convert to datetime
df['date'] = pd.to_datetime(df['date'])

# Convert to string
df['id'] = df['id'].astype(str)

# Convert to category (memory efficient!)
df['category'] = df['category'].astype('category')

# Using .astype() for explicit conversion
df['price'] = df['price'].astype(float)
```

Type Conversion (Example)

```
# Data with wrong types
sales = pd.DataFrame({
    'date': ['2024-01-15', '2024-01-16', '2024-01-17'],
    'amount': ['1000', '1500', '1200'],
    'category': ['A', 'B', 'A']
})

print("Original types:")
print(sales.dtypes)

# Convert to correct types
sales['date'] = pd.to_datetime(sales['date'])
sales['amount'] = pd.to_numeric(sales['amount'])
sales['category'] = sales['category'].astype('category')

print("\nConverted types:")
print(sales.dtypes)
```

Part 2: Data Transformation

Adding New Columns

Create new columns from existing ones:

```
# Simple calculation
df['total_price'] = df['quantity'] * df['unit_price']

# Conditional column
df['status'] = 'Active'

# Based on another column
df['price_category'] = df['price'] > 100

# Multiple columns
df['profit'] = df['revenue'] - df['cost']
df['margin'] = (df['profit'] / df['revenue']) * 100
```

Adding New Columns (Example)

```python
# Sales data
sales = pd.DataFrame({
    'product': ['Laptop', 'Mouse', 'Keyboard', 'Monitor'],
    'quantity': [5, 50, 30, 10],
    'unit_price': [999.99, 29.99, 79.99, 299.99],
    'cost': [700, 15, 40, 200]
})

# Calculate derived columns
sales['total_revenue'] = sales['quantity'] * sales['unit_price']
sales['total_cost'] = sales['quantity'] * sales['cost']
sales['profit'] = sales['total_revenue'] - sales['total_cost']
sales['margin_pct'] = (sales['profit'] / sales['total_revenue']) * 100

print(sales)
```

Removing Columns

Drop columns you don't need:

```
# Drop single column
df_clean = df.drop('column_name', axis=1)

# Drop multiple columns
df_clean = df.drop(['col1', 'col2'], axis=1)

# Drop columns in place (modifies original)
df.drop('column_name', axis=1, inplace=True)

# Alternative: select columns you want to keep
df_clean = df[['col1', 'col2', 'col3']]
```

Renaming Columns

Make column names more meaningful:

```
# Rename specific columns
df_renamed = df.rename(columns={
    'old_name1': 'new_name1',
    'old_name2': 'new_name2'
})

# Rename in place
df.rename(columns={'old': 'new'}, inplace=True)

# Rename all columns at once
df.columns = ['name1', 'name2', 'name3']

# Clean up column names (lowercase, no spaces)
df.columns = df.columns.str.lower().str.replace(' ', '_')
```

Applying Functions: `.apply()`

Apply custom functions to columns or rows:

```python
# Apply function to a Series (column)
df['price_squared'] = df['price'].apply(lambda x: x ** 2)

# Apply function to DataFrame (row-wise)
def calculate_discount(row):
    if row['total'] > 1000:
        return row['total'] * 0.10
    else:
        return 0

df['discount'] = df.apply(calculate_discount, axis=1)
```

`.apply()` Example

```python
# Customer data
customers = pd.DataFrame({
    'name': ['Alice', 'Bob', 'Carol', 'David'],
    'age': [25, 45, 35, 60],
    'purchases': [5, 15, 8, 25]
})

# Categorize customers by age
def age_group(age):
    if age < 30:
        return 'Young'
    elif age < 50:
        return 'Middle'
    else:
        return 'Senior'

customers['age_group'] = customers['age'].apply(age_group)
print(customers)
```

Mapping Values: `.map()`

Replace values based on a mapping:

```
# Map with dictionary
category_map = {'A': 'Active', 'I': 'Inactive', 'P': 'Pending'}
df['status_full'] = df['status'].map(category_map)

# Map with function
df['price_level'] = df['price'].map(lambda x: 'High' if x > 100 else 'Low')
```

Difference from `.apply()`:

- `.map()` - Works on Series, one-to-one mapping
- `.apply()` - Works on Series or DataFrame, more flexible

Sorting Data

```python
# Sort by single column (ascending)
df_sorted = df.sort_values('age')

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

# Sort by multiple columns
df_sorted = df.sort_values(['age', 'name'])

# Different sort order for each column
df_sorted = df.sort_values(
    ['age', 'salary'],
    ascending=[True, False]
)

# Sort by index
df_sorted = df.sort_index()
```

Sorting Example

```
# Sales data
sales = pd.DataFrame({
    'region': ['East', 'West', 'East', 'North', 'West'],
    'sales': [1200, 1800, 900, 1500, 2100],
    'profit': [200, 350, 150, 280, 420]
})

print("Original:")
print(sales)

# Sort by sales (highest first)
print("\nSorted by sales:")
print(sales.sort_values('sales', ascending=False))

# Sort by region, then profit
print("\nSorted by region and profit:")
print(sales.sort_values(['region', 'profit'], ascending=[True, False]))
```

Part 3: Basic Aggregations

GroupBy Operations

Like SQL's `GROUP BY` - split, apply, combine:

SQL:

```sql
SELECT region, AVG(sales)
FROM data
GROUP BY region;
```

Pandas:

```python
df.groupby('region')['sales'].mean()
```

GroupBy Fundamentals

The GroupBy operation has three steps:

1. **Split** - Divide data into groups based on criteria
2. **Apply** - Apply a function to each group
3. **Combine** - Combine results into a data structure

```python
# Basic groupby
grouped = df.groupby('category')

# Apply aggregation
result = grouped['sales'].sum()
```

Simple GroupBy Examples

```python
# Sales by region
sales_by_region = df.groupby('region')['sales'].sum()

# Average salary by department
avg_salary = df.groupby('department')['salary'].mean()

# Count of customers by city
customer_count = df.groupby('city').size()

# Or using count
customer_count = df.groupby('city')['customer_id'].count()
```

GroupBy Example

```python
# Transaction data
transactions = pd.DataFrame({
    'date': ['2024-01-15', '2024-01-15', '2024-01-16', '2024-01-16', '2024-01-17'],
    'region': ['East', 'West', 'East', 'West', 'East'],
    'product': ['A', 'B', 'A', 'A', 'B'],
    'sales': [1000, 1500, 1200, 900, 1100],
    'quantity': [10, 15, 12, 9, 11]
})

# Total sales by region
print("Sales by region:")
print(transactions.groupby('region')['sales'].sum())

# Average quantity by product
print("\nAverage quantity by product:")
print(transactions.groupby('product')['quantity'].mean())
```

Common Aggregation Functions

Function	Description
`.sum()`	Sum of values
`.mean()`	Average
`.median()`	Median
`.min()`	Minimum
`.max()`	Maximum
`.count()`	Count non-null values
`.std()`	Standard deviation
`.var()`	Variance

Multiple Aggregations with `.agg()`

```python
# Multiple aggregations on one column
df.groupby('region')['sales'].agg(['sum', 'mean', 'count'])

# Different aggregations for different columns
df.groupby('region').agg({
    'sales': ['sum', 'mean'],
    'profit': ['sum', 'mean'],
    'quantity': 'sum'
})

# Custom names for aggregations
df.groupby('region')['sales'].agg([
    ('total', 'sum'),
    ('average', 'mean'),
    ('count', 'count')
])
```

`.agg()` Example

```python
# Sales data
sales = pd.DataFrame({
    'region': ['East', 'West', 'East', 'West', 'East', 'West'],
    'sales': [1000, 1500, 1200, 1800, 900, 2100],
    'profit': [200, 350, 250, 400, 180, 450],
    'quantity': [10, 15, 12, 18, 9, 21]
})

# Multiple aggregations
summary = sales.groupby('region').agg({
    'sales': ['sum', 'mean', 'count'],
    'profit': ['sum', 'mean'],
    'quantity': 'sum'
})

print(summary)
```

GroupBy with Multiple Columns

Group by multiple columns:

```
# Group by region and product
summary = df.groupby(['region', 'product'])['sales'].sum()

# Multiple aggregations
summary = df.groupby(['region', 'product']).agg({
    'sales': 'sum',
    'quantity': 'sum'
})

# Reset index to turn multi-index into columns
summary_flat = summary.reset_index()
```

Practical Example: Sales Analysis

```python
# Sales data across regions and products
sales = pd.DataFrame({
    'region': np.random.choice(['East', 'West', 'North', 'South'], 100),
    'product': np.random.choice(['A', 'B', 'C'], 100),
    'category': np.random.choice(['Electronics', 'Clothing', 'Home'], 100),
    'sales': np.random.randint(500, 2000, 100),
    'quantity': np.random.randint(5, 20, 100)
})

print(sales.head())
```

Practical Example (Continued)

```python
# Sales by region and product
regional_product = sales.groupby(['region', 'product'])['sales'].sum()
print("Sales by region and product:")
print(regional_product)

# Product performance across all regions
product_summary = sales.groupby('product').agg({
    'sales': ['sum', 'mean', 'count'],
    'quantity': 'sum'
})
print("\nProduct performance:")
print(product_summary)

# Best performing region
best_region = sales.groupby('region')['sales'].sum().idxmax()
print(f"\nBest performing region: {best_region}")
```

Filtering Groups

Filter groups based on aggregate properties:

```
# Keep only groups where total sales > 5000
high_sales = df.groupby('region').filter(
    lambda x: x['sales'].sum() > 5000
)

# Keep groups with more than 10 records
large_groups = df.groupby('category').filter(
    lambda x: len(x) > 10
)
```

Session 2 Summary

✓ **Data Cleaning**: Handle missing values, remove duplicates, fix types
✓ **Transformation**: Add/remove columns, apply functions, sort data
✓ **Aggregation**: GroupBy operations, multiple aggregations with `.agg()`

Key Takeaways

1. **Clean first, analyze later** - Bad data = bad insights
2. `.isna()`, `.fillna()`, `.dropna()` - Your missing data toolkit
3. `.apply()` **and** `.map()` - Transform data flexibly
4. `.groupby()` - The most powerful aggregation tool
5. `.agg()` - Multiple aggregations in one call

These operations form the foundation of data analysis!

Questions?

Lunch break! 🍕

See you for the practical session!