Vol 2

Mastering Pandas - From Fundamentals to Advanced – 2

Part 4: Combining & Reshaping Data

Part 5: Grouping, Aggregation & Window Functions

Part 6: Working with Dates & Times

https://in.linkedin.com/in/balajikudumu

Mastering Pandas - From Fundamentals to Advanced - 2

Part 4: Combining & Reshaping Data

30. Concatenation and Appending DataFrames
31. Merging and Joining DataFrames
32. Understanding Different Types of Joins
33. Combining Data on Index vs Columns
34. Reshaping with melt and pivot
35. Pivot Tables and Crosstab
36. Stacking and Unstacking
37. Wide vs Long Format Conversion
38. Exploding and Flattening Nested Data

Part 5: Grouping, Aggregation & Window Functions

39. GroupBy Operations and Aggregation
40. Transform and Filter with GroupBy
41. Multi-level Grouping and Named Aggregations
42. Window and Rolling Operations
43. Expanding and Exponentially Weighted Windows
44. Custom Aggregations and Lambda Functions

Part 6: Working with Dates & Times

45. Datetime Conversion and Parsing
46. DateTime Index and Resampling
47. Time Series Shifting and Lagging
48. Periods, Offsets, and Date Ranges
49. Time Zone Handling

Part 4: Combining & Reshaping Data

Below contains details of Part 4 of the "Mastering Pandas" series.

30. Concatenation and Appending DataFrames

pd.concat stacks multiple DataFrames either vertically (axis=0) or horizontally (axis=1). It can preserve original indices or reindex with ignore_index=True.
DataFrame.append() was commonly used but is deprecated, so **pd.concat** is preferred. The parameter keys allows creating a hierarchical index to track the origin of each piece.

```python
import pandas as pd

df1 = pd.DataFrame({'id':[1,2], 'A':[10,20]})
df2 = pd.DataFrame({'id':[3,4], 'A':[30,40]})

v = pd.concat([df1, df2], axis=0, ignore_index=True)
h = pd.concat([df1, df2], axis=1)
k = pd.concat([df1, df2], keys=['x','y'])

print("Vertical concat:\n", v)
print("\nHorizontal concat:\n", h)
print("\nConcat with keys:\n", k)
```

[1] ✓ 1.0s

```
Vertical concat:
    id   A
0   1  10
1   2  20
2   3  30
3   4  40

Horizontal concat:
    id   A  id   A
0   1  10   3  30
1   2  20   4  40

Concat with keys:
      id   A
x 0   1  10
  1   2  20
y 0   3  30
  1   4  40
```

31. Merging and Joining DataFrames

pd.merge performs database-style joins using key columns.
It can handle one-to-one, one-to-many, and many-to-many relationships.
It can also resolve overlapping column names by adding **suffixes**.

```python
import pandas as pd

# Sample DataFrames
left = pd.DataFrame({'id': [1, 2, 3], 'name': ['Alice', 'Bob', 'Charlie']})
right = pd.DataFrame({'id': [2, 3, 4], 'score': [85, 92, 78]})

print("Left DataFrame:")
print(left, "\n")

print("Right DataFrame:")
print(right, "\n")

# --- Inner join ---
inner = pd.merge(left, right, on='id', how='inner')
print("Inner join (only common ids):\n", inner, "\n")

# --- Left join ---
left_join = pd.merge(left, right, on='id', how='left')
print("Left join (all from left, match from right):\n", left_join, "\n")

# --- Right join ---
right_join = pd.merge(left, right, on='id', how='right')
print("Right join (all from right, match from left):\n", right_join, "\n")

# --- Outer join ---
outer = pd.merge(left, right, on='id', how='outer')
print("Outer join (all ids from both sides):\n", outer, "\n")

# --- Example with overlapping column names ---
left2 = pd.DataFrame({'id': [1, 2], 'value': ['X', 'Y']})
right2 = pd.DataFrame({'id': [2, 3], 'value': ['Z', 'W']})

overlap = pd.merge(left2, right2, on='id', how='outer', suffixes=('_left', '_right'))
print("Outer join with overlapping column names:\n", overlap)
```

```
Left DataFrame:
   id     name
0   1    Alice
1   2      Bob
2   3  Charlie

Right DataFrame:
   id  score
0   2     85
1   3     92
2   4     78

Inner join (only common ids):
    id     name  score
0   2      Bob     85
1   3  Charlie     92

Left join (all from left, match from right):
    id     name  score
0   1    Alice    NaN
1   2      Bob   85.0
2   3  Charlie   92.0

Right join (all from right, match from left):
    id     name  score
0   2      Bob     85
1   3  Charlie     92
2   4      NaN     78

Outer join (all ids from both sides):
    id     name  score
0   1    Alice    NaN
1   2      Bob   85.0
2   3  Charlie   92.0
3   4      NaN   78.0

Outer join with overlapping column names:
    id value_left value_right
0   1          X         NaN
1   2          Y           Z
2   3        NaN           W
```

32. Understanding Different Types of Joins

An **inner join** keeps only matching keys.
A **left join** keeps all keys from the left table and adds matches from the right.
A **right join** keeps all keys from the right table and adds matches from the left.
An **outer join** keeps all keys from both tables and fills missing values with NaN.

```
L = pd.DataFrame({'key':[1,2,3], 'Lval':['a','b','c']})
R = pd.DataFrame({'key':[2,3,4], 'Rval':[10,20,30]})

for how in ['inner','left','right','outer']:
    print(f"{how} join:")
    print(pd.merge(L, R, on='key', how=how), "\n")
```

[3] ✓ 0.0s

```
inner join:
   key Lval  Rval
0    2    b    10
1    3    c    20

left join:
   key Lval  Rval
0    1    a   NaN
1    2    b  10.0
2    3    c  20.0

right join:
   key Lval  Rval
0    2    b    10
1    3    c    20
2    4  NaN    30

outer join:
   key Lval  Rval
0    1    a   NaN
1    2    b  10.0
2    3    c  20.0
3    4  NaN  30.0
```

33. Combining Data on Index vs Columns

Merges can happen using index values instead of columns.
The join method joins on index by default.
You can also set **left_index=True** and **right_index=True** in **pd.merge**.

```python
import pandas as pd

# Create sample DataFrames with indexes
left = pd.DataFrame({'Lval': [1, 2]}, index=['a', 'b'])
right = pd.DataFrame({'Rval': [10, 20]}, index=['b', 'a'])

print("Left DataFrame:")
print(left, "\n")

print("Right DataFrame:")
print(right, "\n")

# --- Using join() ---
print("Join using .join() with outer join:")
print(left.join(right, how='outer'), "\n")

# --- Using pd.merge() with index ---
print("Merge using pd.merge() with indexes (outer join):")
print(pd.merge(left, right, left_index=True, right_index=True, how='outer'))
```

✓ 0.0s

```
Left DataFrame:
   Lval
a     1
b     2

Right DataFrame:
   Rval
b    10
a    20

Join using .join() with outer join:
   Lval  Rval
a     1    20
b     2    10

Merge using pd.merge() with indexes (outer join):
   Lval  Rval
a     1    20
b     2    10
```

34. Reshaping with melt and pivot

melt converts wide format into long format.
pivot converts long format into wide format.
pivot_table allows aggregations when duplicates exist.

```python
import pandas as pd
# Wide format DataFrame
wide = pd.DataFrame({
    'id': [1, 2],
    'sales_2019': [100, 150],
    'sales_2020': [120, 160]
})
print("Wide format DataFrame:")
print(wide, "\n")

# --- Melt: wide -> long ---
long = pd.melt(
    wide,
    id_vars=['id'],            # columns to keep fixed
    var_name='year',           # new column for former headers
    value_name='sales'         # new column for values
)
print("Long format (after melt):")
print(long, "\n")

# --- Pivot: long -> wide ---
pivoted = long.pivot(
    index='id',                # unique identifier
    columns='year',            # new headers
    values='sales'             # values to fill
)
print("Wide format (after pivot):")
print(pivoted, "\n")

# --- Pivot Table (handles duplicates with aggregation) ---
# Example with duplicate entries
long_with_dupes = pd.DataFrame({
    'id': [1, 1, 2, 2],
    'year': ['sales_2019', 'sales_2019', 'sales_2020', 'sales_2020'],
    'sales': [100, 120, 150, 160]
})

pivot_tabled = pd.pivot_table(
    long_with_dupes,
    index='id',
    columns='year',
    values='sales',
    aggfunc='mean'   # handles duplicates with aggregation
)

print("Pivot table (with aggregation):")
print(pivot_tabled)
```
✓ 0.0s

```
Wide format DataFrame:
   id  sales_2019  sales_2020
0   1         100         120
1   2         150         160

Long format (after melt):
   id        year  sales
0   1  sales_2019    100
1   2  sales_2019    150
2   1  sales_2020    120
3   2  sales_2020    160

Wide format (after pivot):
year  sales_2019  sales_2020
id
1            100         120
2            150         160

Pivot table (with aggregation):
year  sales_2019  sales_2020
id
1          110.0         NaN
2            NaN       155.0
```

35. Pivot Tables and Crosstab

pivot_table aggregates values with a function such as sum or mean.
crosstab produces frequency tables and can normalize values.

```
df = pd.DataFrame({
    'region':['East','East','West','West'],
    'product':['A','B','A','B'],
    'sales':[100,200,150,120]
})
pt = pd.pivot_table(df, values='sales', index='region', columns='product', aggfunc='sum')
ct = pd.crosstab(df['region'], df['product'])
print(pt)
print(ct)
```

✓ 0.0s

```
product    A    B
region
East     100  200
West     150  120
product  A  B
region
East     1  1
West     1  1
```

36. Stacking and Unstacking

stack pivots columns into the row index.
unstack pivots a row index level into columns.

```python
import pandas as pd
df = pd.DataFrame({
    'city': ['X', 'X', 'Y', 'Y'],
    'year': [2019, 2020, 2019, 2020],
    'value': [100, 150, 200, 250]
})
print("Original DataFrame:")
print(df, "\n")

wide = df.pivot(index='city', columns='year', values='value')
print("Wide format (pivoted):")
print(wide, "\n")

# --- Stack: columns → row index (long format) ---
stacked = wide.stack()
print("Stacked (columns turned into row index):")
print(stacked, "\n")

# --- Unstack: row index → columns ---
unstacked = stacked.unstack()
print("Unstacked (row index level back to columns):")
print(unstacked)
```

✓ 0.0s

```
Original DataFrame:
  city  year  value
0    X  2019    100
1    X  2020    150
2    Y  2019    200
3    Y  2020    250

Wide format (pivoted):
year  2019  2020
city
X      100   150
Y      200   250

Stacked (columns turned into row index):
city  year
X     2019    100
      2020    150
Y     2019    200
      2020    250
dtype: int64

Unstacked (row index level back to columns):
year  2019  2020
city
X      100   150
Y      200   250
```


37. Wide vs Long Format Conversion

Wide format has one row per observation with multiple measure columns.
Long format has one row per observation-measure pair.
You use **melt** to convert wide to long.
You use **pivot** to convert long back to wide.

```python
import pandas as pd

# --- Wide format DataFrame ---
wide = pd.DataFrame({'person': ['Alice', 'Bob'], 'height_cm': [165, 180], 'weight_kg': [60, 80]})
print("Wide format DataFrame:")
print(wide, "\n")

# --- Convert wide → long using melt ---
long = pd.melt(
    wide,
    id_vars=['person'],         # column(s) to keep
    var_name='measure',         # new column for former headers
    value_name='value'          # new column for values
)
print("Long format (after melt):")
print(long, "\n")

# --- Convert long → wide using pivot ---
wide_again = long.pivot(
    index='person',             # identifier column
    columns='measure',          # former variable column becomes headers
    values='value'              # values to fill
).reset_index()                 # optional: reset index to make 'person' a column again

print("Wide format (after pivot):")
print(wide_again)
```

✓ 0.0s

```
Wide format DataFrame:
  person  height_cm  weight_kg
0  Alice        165         60
1    Bob        180         80

Long format (after melt):
  person    measure  value
0  Alice  height_cm    165
1    Bob  height_cm    180
2  Alice  weight_kg     60
3    Bob  weight_kg     80

Wide format (after pivot):
measure person  height_cm  weight_kg
0        Alice        165         60
1          Bob        180         80
```

38. Exploding and Flattening Nested Data

explode expands list-like entries into separate rows.
json_normalize flattens dictionary-like or JSON data into columns.

```
import pandas as pd
# Sample DataFrame with list and dictionary columns
df = pd.DataFrame({
    'id': [1, 2],
    'tags': [['red', 'blue'], ['green']],
    'info': [{'a': 1, 'b': 2}, {'a': 3, 'b': 4}]
})
print("Original DataFrame:")
print(df, "\n")

# --- Explode: expand list-like column into separate rows ---
exploded = df.explode('tags').reset_index(drop=True)
print("After exploding 'tags' column:")
print(exploded, "\n")

# --- Flatten dictionary-like column using json_normalize ---
info_flat = pd.json_normalize(df['info'])
df_flattened = pd.concat([df.drop(columns=['info']), info_flat], axis=1)
print("After flattening 'info' column with json_normalize:")
print(df_flattened)
```

✓ 0.0s

```
Original DataFrame:
   id          tags              info
0   1  [red, blue]  {'a': 1, 'b': 2}
1   2      [green]  {'a': 3, 'b': 4}

After exploding 'tags' column:
   id   tags              info
0   1    red  {'a': 1, 'b': 2}
1   1   blue  {'a': 1, 'b': 2}
2   2  green  {'a': 3, 'b': 4}

After flattening 'info' column with json_normalize:
   id          tags  a  b
0   1  [red, blue]  1  2
1   2      [green]  3  4
```

Part 5: Grouping, Aggregation & Window Functions

Below contains details of Part 5 of the "Mastering Pandas" series.

39. GroupBy Operations and Aggregation

GroupBy in pandas is used to split data into groups based on some criteria. Once grouped, we can aggregate, transform, or filter the groups. Common aggregation functions include **sum**, **mean**, **count**, **min**, and **max**.

```python
import pandas as pd

data = {
    'Department': ['HR', 'HR', 'IT', 'IT', 'Finance', 'Finance'],
    'Employee': ['Alice', 'Bob', 'Charlie', 'David', 'Eva', 'Frank'],
    'Salary': [50000, 55000, 60000, 65000, 70000, 72000],
    'Experience': [2, 3, 5, 4, 6, 7]
}
df = pd.DataFrame(data)

grouped = df.groupby('Department').agg({'Salary': 'sum', 'Experience': 'mean'})
print(grouped)
```

✓ 0.5s

```
            Salary  Experience
Department
Finance     142000         6.5
HR          105000         2.5
IT          125000         4.5
```


40. Transform and Filter with GroupBy

Transform returns a DataFrame of the same shape as the original, which is useful for normalization or adding group-based values. Filter allows filtering groups based on a condition.

```python
df['Salary_relative'] = df.groupby('Department')['Salary'].transform(lambda x: x - x.mean())
print(df)

filtered = df.groupby('Department').filter(lambda x: x['Experience'].mean() > 4)
print(filtered)
```

✓ 0.0s

```
  Department Employee  Salary  Experience  Salary_relative
0         HR    Alice   50000           2          -2500.0
1         HR      Bob   55000           3           2500.0
2         IT  Charlie   60000           5          -2500.0
3         IT    David   65000           4           2500.0
4    Finance      Eva   70000           6          -1000.0
5    Finance    Frank   72000           7           1000.0
  Department Employee  Salary  Experience  Salary_relative
2         IT  Charlie   60000           5          -2500.0
3         IT    David   65000           4           2500.0
4    Finance      Eva   70000           6          -1000.0
5    Finance    Frank   72000           7           1000.0
```

41. Multi-level Grouping and Named Aggregations

You can group by multiple columns and give custom names for aggregated columns. Named aggregations provide clarity when multiple aggregation functions are applied.

```python
df['Gender'] = ['F', 'M', 'M', 'M', 'F', 'M']

multi_grouped = df.groupby(['Department', 'Gender']).agg(
    Total_Salary=('Salary', 'sum'),
    Avg_Experience=('Experience', 'mean')
)
print(multi_grouped)
```

✓ 0.0s

```
                   Total_Salary  Avg_Experience
Department Gender
Finance    F              70000             6.0
           M              72000             7.0
HR         F              50000             2.0
           M              55000             3.0
IT         M             125000             4.5
```

42. Window and Rolling Operations

Window functions compute statistics over a moving set of rows. **Rolling** allows moving or rolling calculations like moving average.

```
df_sorted = df.sort_values('Employee')

df_sorted['Salary_Rolling_Mean'] = df_sorted['Salary'].rolling(window=2).mean()
print(df_sorted)
```

✓ 0.0s

```
  Department Employee  Salary  Experience  Salary_relative Gender  \
0         HR    Alice   50000           2          -2500.0      F
1         HR      Bob   55000           3           2500.0      M
2         IT  Charlie   60000           5          -2500.0      M
3         IT    David   65000           4           2500.0      M
4    Finance      Eva   70000           6          -1000.0      F
5    Finance    Frank   72000           7           1000.0      M

   Salary_Rolling_Mean
0                  NaN
1              52500.0
2              57500.0
3              62500.0
4              67500.0
5              71000.0
```

43. Expanding and Exponentially Weighted Windows

Expanding computes cumulative metrics over rows. Exponentially weighted moving average gives more weight to recent data points.

```
df_sorted['Salary_Expanding_Mean'] = df_sorted['Salary'].expanding().mean()
df_sorted['Salary_EWMA'] = df_sorted['Salary'].ewm(alpha=0.5).mean()
print(df_sorted)
```

✓ 0.0s

```
  Department Employee  Salary  Experience  Salary_relative Gender  \
0         HR    Alice   50000           2          -2500.0      F
1         HR      Bob   55000           3           2500.0      M
2         IT  Charlie   60000           5          -2500.0      M
3         IT    David   65000           4           2500.0      M
4    Finance      Eva   70000           6          -1000.0      F
5    Finance    Frank   72000           7           1000.0      M

   Salary_Rolling_Mean  Salary_Expanding_Mean   Salary_EWMA
0                  NaN                50000.0  50000.000000
1              52500.0                52500.0  53333.333333
2              57500.0                55000.0  57142.857143
3              62500.0                57500.0  61333.333333
4              67500.0                60000.0  65806.451613
5              71000.0                62000.0  68952.380952
```

44. Custom Aggregations and Lambda Functions

Custom aggregations allow using lambda functions or user-defined functions in **agg** for advanced calculations. **Lambda** functions are useful for computing metrics like range or difference within groups.

```
import pandas as pd

data = {
    'Department': ['HR', 'HR', 'IT', 'IT', 'Finance', 'Finance'],
    'Employee': ['Alice', 'Bob', 'Charlie', 'David', 'Eva', 'Frank'],
    'Salary': [50000, 55000, 60000, 65000, 70000, 72000],
    'Experience': [2, 3, 5, 4, 6, 7]
}

df = pd.DataFrame(data)
# Correct custom aggregation using tuples for named aggregation
custom_agg = df.groupby('Department').agg(
    Salary_Range=('Salary', lambda x: x.max() - x.min()),
    Total_Experience=('Experience', lambda x: x.sum())
)
print(custom_agg)
```

✓ 0.0s

```
            Salary_Range  Total_Experience
Department
Finance             2000                13
HR                  5000                 5
IT                  5000                 9
```

Part 6: Working with Dates & Times

Below contains details of Part 6 of the "Mastering Pandas" series.

45. Datetime Conversion and Parsing

Datetime conversion is important for handling date and time data in pandas. We often need to convert strings to **datetime** objects for analysis. **pd.to_datetime()** is commonly used.

```
import pandas as pd

# Sample DataFrame with string dates
data = {
    'Event': ['A', 'B', 'C', 'D'],
    'Date': ['2025-01-10', '2025/02/15', 'March 20, 2025', '2025-04-25 14:30']
}

df = pd.DataFrame(data)
# Convert Date column to datetime (mixed formats)
df['Date'] = pd.to_datetime(df['Date'], format='mixed', errors='raise')
print(df)
```

✓ 0.0s

```
  Event                Date
0     A 2025-01-10 00:00:00
1     B 2025-02-15 00:00:00
2     C 2025-03-20 00:00:00
3     D 2025-04-25 14:30:00
```

46. DateTime Index and Resampling

Setting a datetime column as an **index** allows time-based operations like resampling. Resampling aggregates data over a new time frequency.

```
date_rng = pd.date_range(start='2025-01-01', end='2025-01-10', freq='D')
ts_data = pd.DataFrame({'Date': date_rng, 'Sales': [100, 120, 130, 90, 150, 160, 170, 180, 200, 210]})
ts_data.set_index('Date', inplace=True)
print(ts_data)
```

✓ 0.0s

```
            Sales
Date
2025-01-01    100
2025-01-02    120
2025-01-03    130
2025-01-04     90
2025-01-05    150
2025-01-06    160
2025-01-07    170
2025-01-08    180
2025-01-09    200
2025-01-10    210
```

47. Time Series Shifting and Lagging

Shifting moves data forward or backward in time. It is useful for creating lag features in time series analysis.

```
# Shift sales forward by 1 day
ts_data['Sales_Lag1'] = ts_data['Sales'].shift(1)
print(ts_data)
```

✓ 0.0s

```
            Sales  Sales_Lag1
Date
2025-01-01    100         NaN
2025-01-02    120       100.0
2025-01-03    130       120.0
2025-01-04     90       130.0
2025-01-05    150        90.0
2025-01-06    160       150.0
2025-01-07    170       160.0
2025-01-08    180       170.0
2025-01-09    200       180.0
2025-01-10    210       200.0
```

48. Periods, Offsets, and Date Ranges

Pandas can handle **periods** (month, quarter, year) and **offsets** for custom time intervals. **pd.date_range** creates a range of dates with different frequencies.

```
# Create monthly period range
periods = pd.period_range(start='2025-01', end='2025-06', freq='M')
print(periods)

# Custom offset: every 2 days
custom_range = pd.date_range(start='2025-01-01', periods=5, freq='2D')
print(custom_range)
```

✓ 0.0s

```
PeriodIndex(['2025-01', '2025-02', '2025-03', '2025-04', '2025-05', '2025-06'], dtype='period[M]')
DatetimeIndex(['2025-01-01', '2025-01-03', '2025-01-05', '2025-01-07',
               '2025-01-09'],
              dtype='datetime64[ns]', freq='2D')
```

49. Time Zone Handling

Pandas supports time zones for datetime objects. You can **localize** naive datetimes or **convert** between time zones.

```python
# Sample naive datetime
ts = pd.Series(pd.date_range('2025-01-01 08:00', periods=3, freq='D'))
print("Naive datetime:\n", ts)

# Localize to UTC
ts_utc = ts.dt.tz_localize('UTC')
print("\nUTC datetime:\n", ts_utc)

# Convert to US/Eastern
ts_est = ts_utc.dt.tz_convert('US/Eastern')
print("\nUS/Eastern datetime:\n", ts_est)
```

✓ 0.1s

```
Naive datetime:
 0   2025-01-01 08:00:00
1   2025-01-02 08:00:00
2   2025-01-03 08:00:00
dtype: datetime64[ns]

UTC datetime:
 0   2025-01-01 08:00:00+00:00
1   2025-01-02 08:00:00+00:00
2   2025-01-03 08:00:00+00:00
dtype: datetime64[ns, UTC]

US/Eastern datetime:
 0   2025-01-01 03:00:00-05:00
1   2025-01-02 03:00:00-05:00
2   2025-01-03 03:00:00-05:00
dtype: datetime64[ns, US/Eastern]
```