Lesson 08: Basic Operations

Data Science with Python – BSc Course

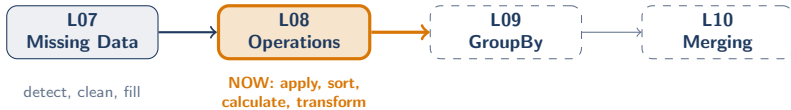
Data Science Program

BSc Course

45 Minutes

Where We Are

Last time: you cleaned the mess. Now: transform it.



Raw data is messy even after cleaning. You need to create new columns, calculate returns, sort by criteria, and apply custom functions. These operations turn raw data into actionable information.

Clean data + transformations = analysis-ready datasets

Learning Objectives

After this lesson, you will be able to:

- **Apply** custom functions with `apply()` and lambda expressions
- **Compute** arithmetic operations and derived columns
- **Sort** data by one or multiple columns with `sort_values()`
- **Summarize** categories with `value_counts()` and binning
- **Calculate** financial returns and rolling statistics

Finance Application: Compute daily returns, moving averages, Bollinger Bands, and rolling volatility.

These operations are the daily toolkit of every data analyst

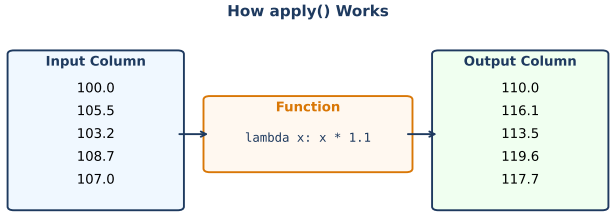
Raw Data Is Messy. These Operations Clean It.

From raw columns to derived insights

5 raw columns become 10+ derived features. Every quantitative analysis starts with column transformations.

Transformation = creating the columns your analysis actually needs

apply(): Custom Transformations




```
df['Price_Adjusted'] = df['Price'].apply(lambda x: x * 1.1)
```

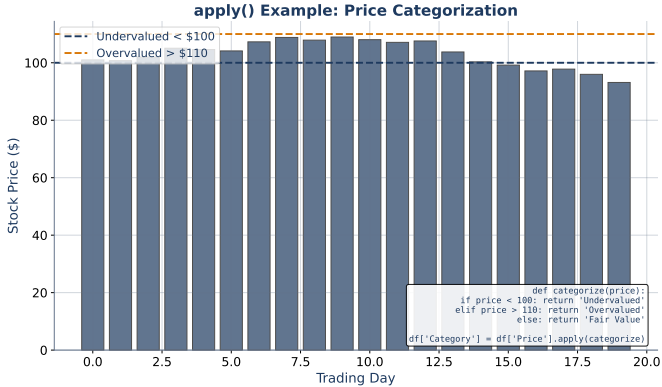
Types of apply():

Column-wise: df["col"].apply(func)
Row-wise: df.apply(func, axis=1)
Element-wise: df.applymap(func)

---

apply() runs your function on each element, row, or column

Lambda Functions with apply()



apply(lambda x: ...) is the most common pattern for custom transforms

String Operations

The .str accessor transforms text columns

```
# Extract year from date string
df['Year'] = df['Date'].str[:4]
# Clean ticker symbols
df['Ticker'] = df['Ticker'].str.upper().str.strip()
# Check for pattern
mask = df['Name'].str.contains('Tech')
```

Common methods:

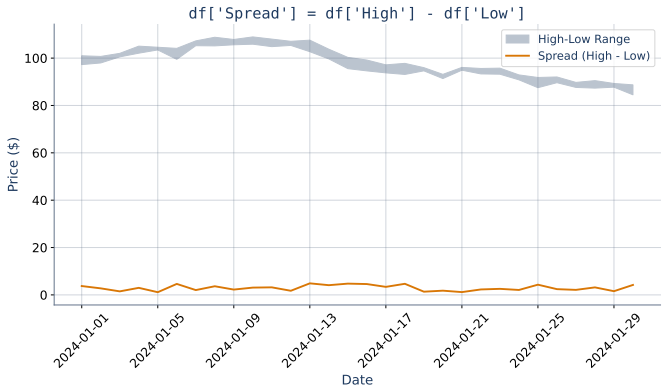
- `.str.upper()`, `.str.lower()`
- `.str.strip()`, `.str.replace()`
- `.str.contains()`, `.str.len()`

Type conversion:

```
df['Price'] = df['Price']
    .str.replace('$', '')
    .astype(float)
```

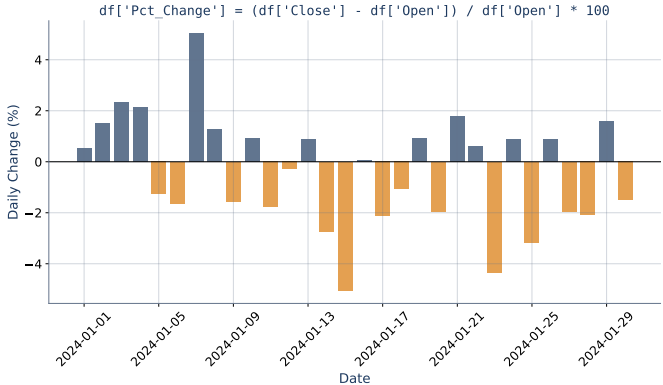
.str accessor unlocks all Python string methods for vectorized column operations

Arithmetic: Calculating Spread



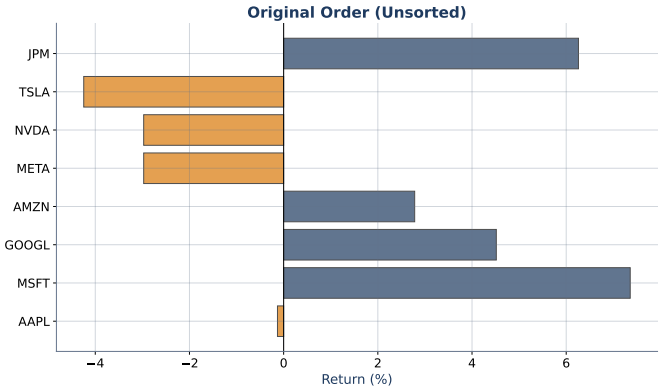
Spread = High − Low measures intraday price range

Percentage Change



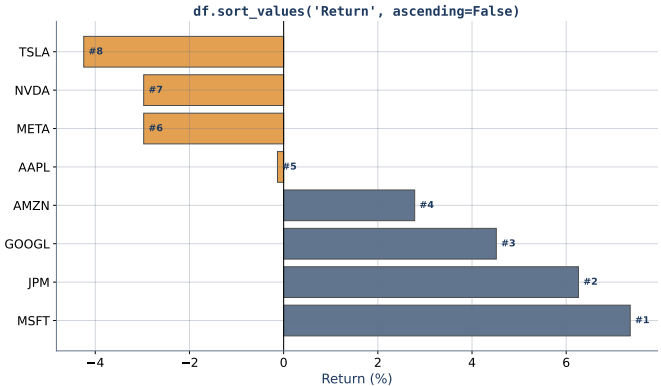
pct_change() calculates period-over-period percentage changes

Sorting with sort_values()



Raw data arrives in arbitrary order – sorting reveals patterns

Sorted by Returns



sort_values('Returns') ranks stocks from worst to best performance

Ranking and Binning

Turn continuous values into categories

Ranking:

```
df['Rank'] = df['Return']
    .rank(ascending=False)
# Percentile rank
df['Pctile'] = df['Return']
    .rank(pct=True)
```

Rank 1 = best performer

Binning with pd.cut():

```
df['Size'] = pd.cut(
    df['MarketCap'],
    bins=[0, 2e9, 10e9, 1e12],
    labels=['Small','Mid','Large'])
```

Custom bins from continuous data

pd.qcut() for equal-frequency bins: `pd.qcut(df['Return'], q=4, labels=['Q1','Q2','Q3','Q4'])`

Binning converts continuous data to categories for groupby analysis

Checkpoint: Test Your Understanding

Q1: What is the difference between `apply()` on a Series vs on a DataFrame?

Q2: `df.sort_values(['A','B'], ascending=[True,False])` – describe the sort order.

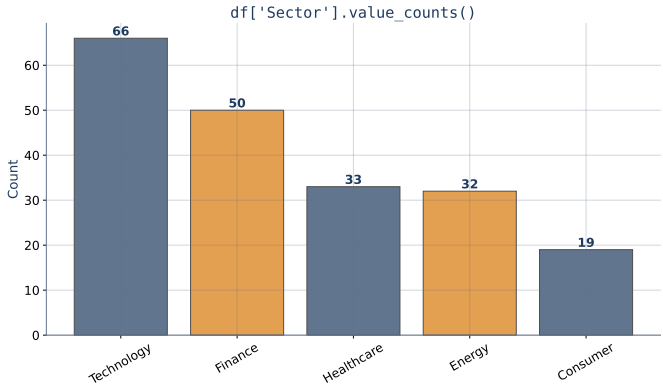
Q3: When would you use `pd.cut()` vs `pd.qcut()`?

Think for 30 seconds. Q1: Series → element-wise; DataFrame → row/column-wise. Q2: A ascending, B descending within ties. Q3: cut = fixed bins; qcut = equal-frequency bins.

---

apply() is powerful but slower than vectorized operations – use built-ins when possible

value_counts(): Category Frequencies



value_counts() is the fastest way to summarize categorical columns

Replace and Rename

Clean up values and column names

Replace values:

```
# Map old values to new
df['Sector'].replace({
    'IT': 'Technology',
    'Fin':  'Financials'})
# Replace by condition
df['Signal'] = np.where(
    df['Return'] > 0,
    'Buy', 'Sell')
```

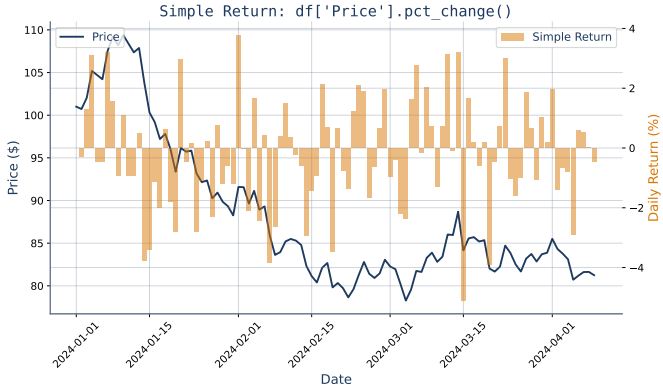
Rename columns:

```
df.rename(columns={
    'Adj Close':  'AdjClose',
    'Volume':  'Vol'})
# Rename with function
df.columns = df.columns
    .str.lower()
    .str.replace(' ', '_')
```

Type conversion: `df['Price'] = df['Price'].astype(float)` | `pd.to_datetime(df['Date'])`

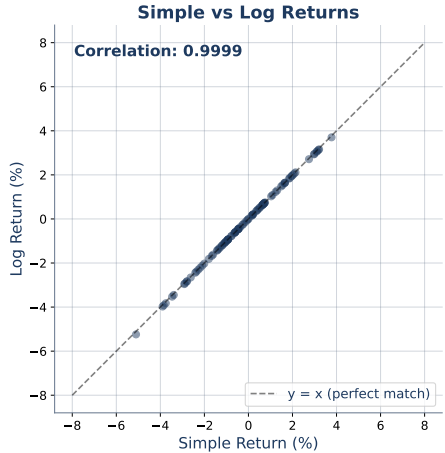
Clean column names and consistent values make downstream analysis easier

Finance: Price and Returns



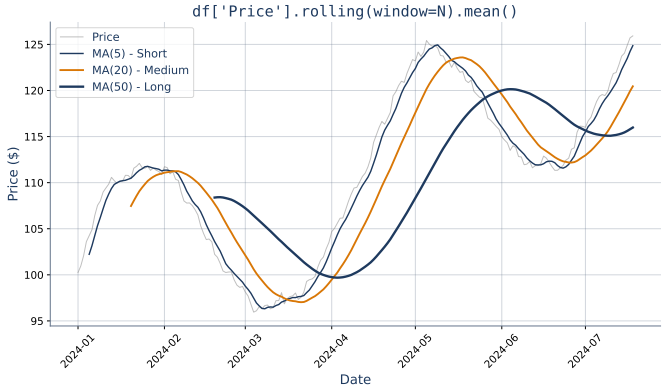
Returns = relative changes. Prices = absolute levels. Analyze returns, not prices.

Finance: Simple vs Log Returns



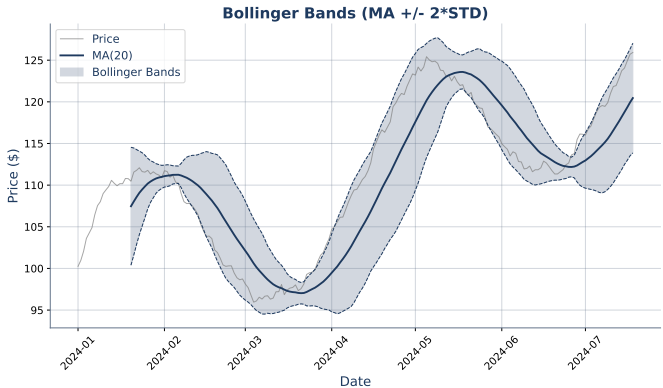
Log returns: $\ln(P_t/P_0) = \sum \ln(P_i/P_{i-1})$ – time-additive

Finance: Moving Averages



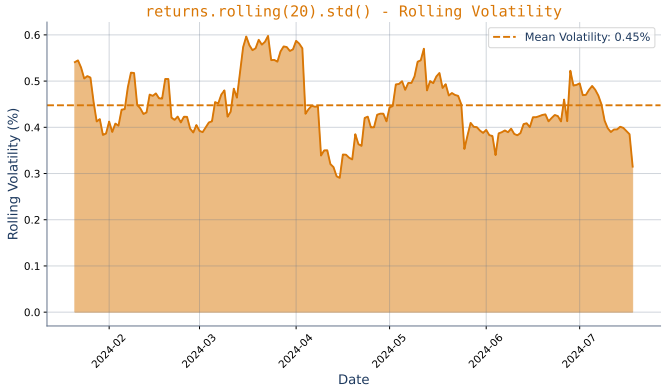
rolling(N).mean() smooths noise – compare 20-day vs 50-day trends

Finance: Bollinger Bands



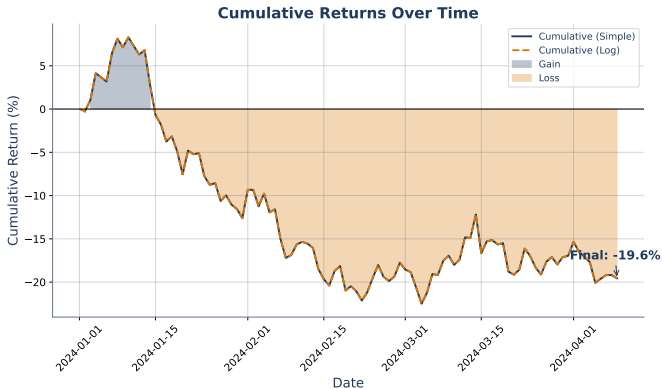
Bollinger Bands = MA $\pm$ 2σ. Price outside bands signals unusual volatility.

Finance: Rolling Volatility



rolling(20).std() measures recent price variability – clusters in time

Finance: Cumulative Returns



(1 + returns).cumprod() − 1 gives cumulative performance over time

Hands-on Exercise (25 min)

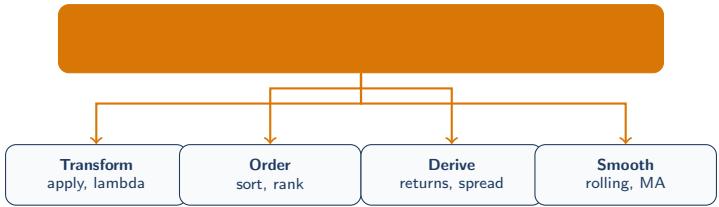
Transform stock data into analysis-ready features:

1. Load stock data. Create a 'Returns' column: `df['Close'].pct_change()`
2. Create a 'Spread' column: `df['High'] - df['Low']`
3. Add a signal: `np.where(df['Returns'] > 0, 'Up', 'Down')`
4. Sort by returns descending. Show top 5 with `nlargest()`
5. Calculate 20-day and 50-day moving averages
6. Plot Bollinger Bands (MA $\pm$ 2 $\times$ rolling std)
7. Use `pd.cut()` to bin returns into 'Loss', 'Flat', 'Gain'

Bonus: Calculate the Sharpe ratio: `returns.mean() / returns.std() * np.sqrt(252)`

These operations form the basis of every quantitative finance workflow

The Big Idea



Raw columns → derived features. Every analysis starts here.

Four power tools: apply for custom logic, sort for order, arithmetic for features, rolling for trends

Lesson Summary

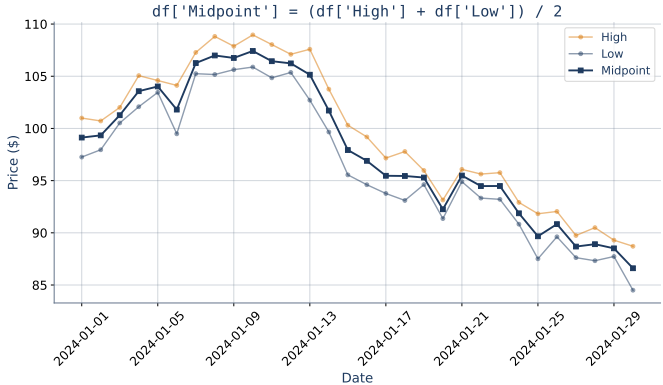
Key Takeaways:

- `apply()` runs custom functions; prefer vectorized operations when possible
- `sort_values()` orders data; multi-column sort for hierarchical ordering
- `pct_change()` for returns; `np.log()` for log returns
- `rolling(N).mean()` and `.std()` for smoothed trends and volatility
- `pd.cut()` / `pd.qcut()` convert continuous to categorical

Up Next: L09 – GroupBy Operations
You can transform individual columns. But how do you compute the average return *by sector*? GroupBy is the answer.

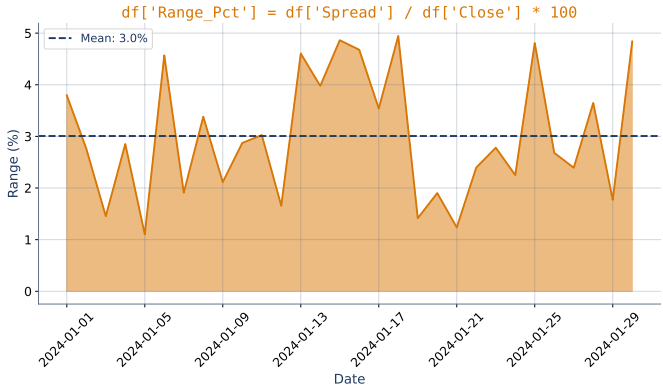
L08 transforms columns. L09 transforms groups. Together: full data wrangling.

Arithmetic: Midpoint Calculation



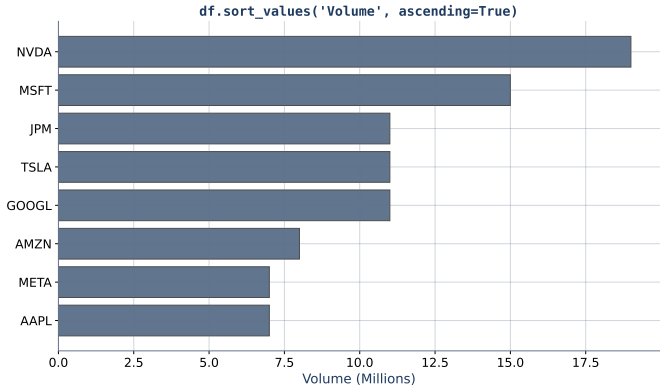
Midpoint = (High + Low) / 2 represents the typical price

Arithmetic: Range as Percentage



Range % = (High − Low) / Open normalizes volatility across price levels

Sorted by Volume



sort_values('Volume') identifies most actively traded stocks

Multi-Column Sorting

Multi-Column Sorting

```python
# Sort by single column
df.sort_values('Price')
```

```python
# Sort descending
df.sort_values('Price', ascending=False)
```

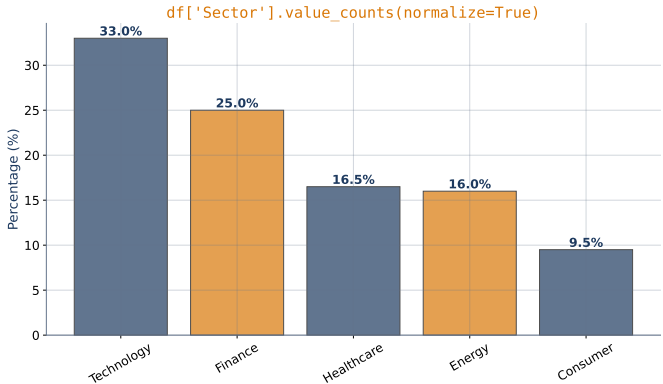
```python
# Sort by multiple columns
df.sort_values(['Sector', 'Return'],
               ascending=[True, False])
```

```python
# Sort by index
df.sort_index()
```

```python
# Reset index after sort
df.sort_values('Price').reset_index(drop=True)
```

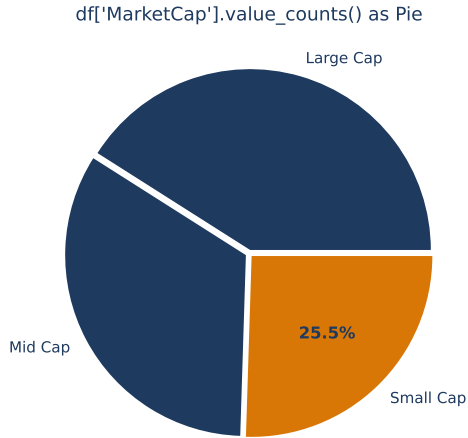
sort_values(['Sector', 'Returns']) for hierarchical ordering

Sector Distribution (Percentages)



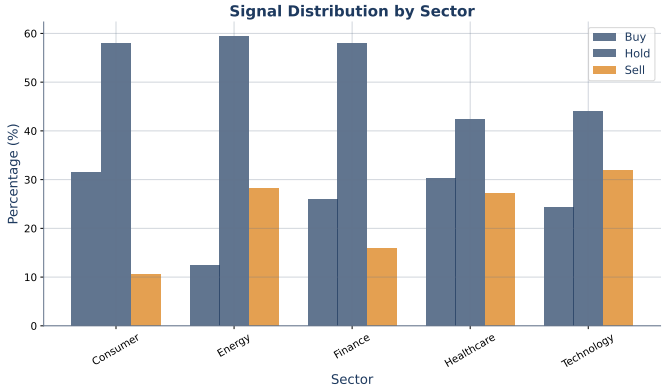
value_counts(normalize=True) converts counts to proportions

Market Cap Distribution



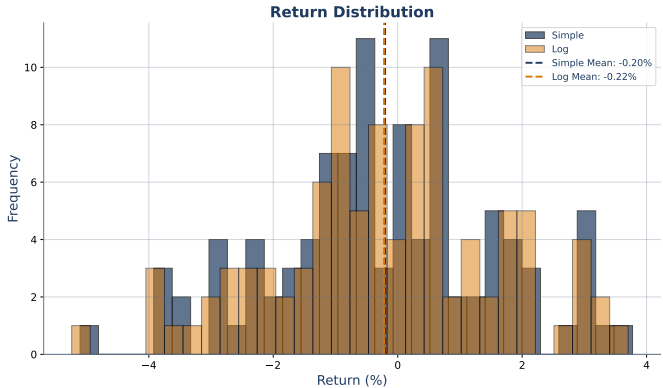
Pie charts visualize categorical proportions – use sparingly

Trading Signals by Sector



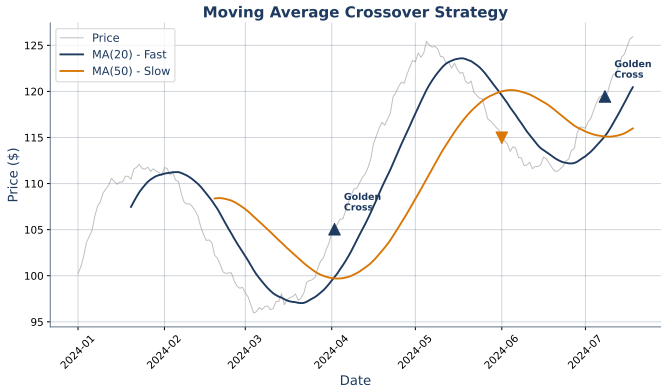
Cross-tabulate signals with sectors to find sector-specific patterns

Return Distribution



Stock returns are approximately normally distributed with fat tails

Moving Average Crossover Strategy



Golden cross (short MA > long MA) signals potential uptrend

pandas Operations Quick Reference

Column Creation:
`df['new'] = values`

`df.assign(new=val)`

Sorting:
`df.sort_values('col')`

`df.sort_index()`

Arithmetic:
`df['A'] + df['B']`

`df['A'].pct_change()`

Aggregation:
`df['A'].value_counts()`

`df['A'].nunique()`

Transformations:
`df['A'].apply(func)`

`df.map(func)`

Rolling Stats:
`df.rolling(N).mean()`

`df.rolling(N).std()`

Finance: `pct_change()` | `np.log(P/P.shift(1))` | `rolling(20).std()`

Keep this reference handy for data manipulation

Creating New Columns: Four Methods

1. Direct Assignment:

```
df['Returns'] = df['Close'].pct_change()
```

2. Using assign():

```
df = df.assign(Returns=df['Close'].pct_change())
```

3. Arithmetic:

```
df['Spread'] = df['High'] - df['Low']
```

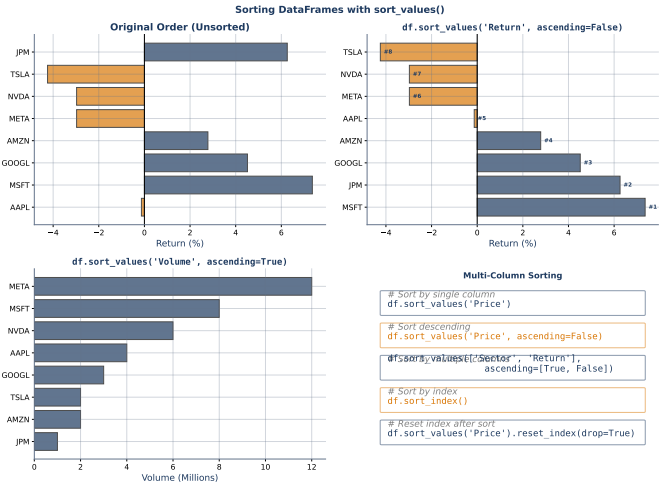
4. Conditional:

```
df['Signal'] = np.where(df['Returns']>0, 'Up', 'Down')
```

Direct assignment is simplest; assign() enables method chaining.

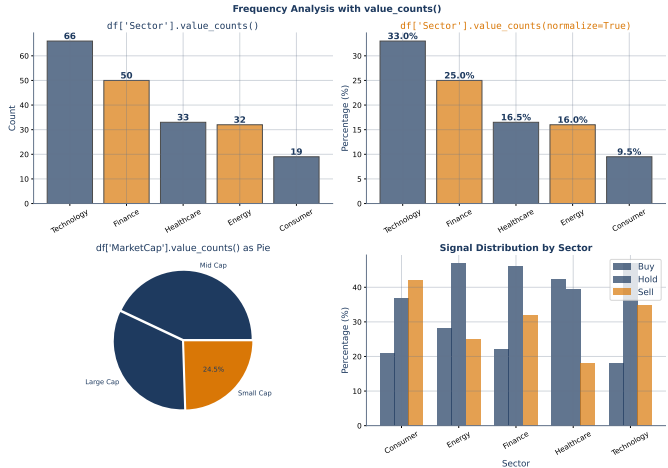
Choose the method that makes your code most readable

Sorting Overview



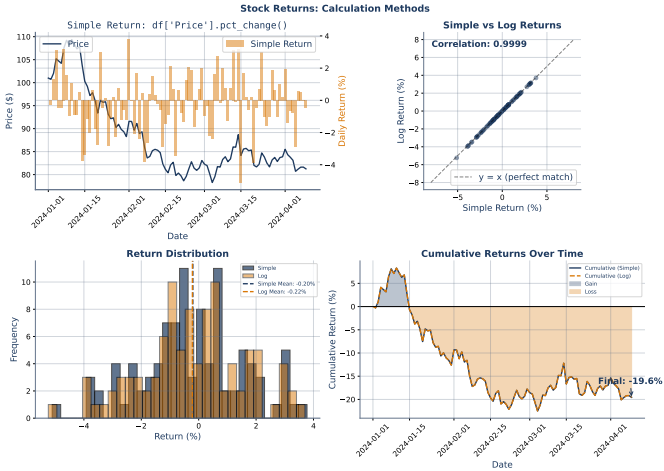
Sorting comparison: unsorted, by return, by volume, multi-column

Value Counts Overview



Value counts: absolute, percentage, pie chart, cross-tabulation

Returns Comparison



Returns overview: price vs returns, simple vs log, distribution, cumulative