Pandas

Interview Preparation Guide

for Data Engineers and ML Engineers

by Inder P Singh

3-Week Pandas & ML Interview Prep Workshop:

Live, hands-on, career-focused. Message “PANDAS” to reserve.

Software Testing Material - original reference

Kaggle (16 Badges & 16 Certifications) https://www.kaggle.com/inderpsingh
YouTube Channel: Software and Testing Training (364 Tutorials, 10 Million+ Views, 82 K Subscribers)
Blog: Fourth Industrial Revolution (42 posts) https://fourth-industrial-revolution.blogspot.com/

Copyright © 2026 All Rights Reserved.

Software and Testing Training – For personal use only

Copyright © 2026 All Rights Reserved.

Chapter 1: Series and DataFrames

Welcome to the foundation of data manipulation in Python. Before we can clean messy datasets, visualize trends, or train machine learning models, we must understand the containers that hold our data.

If you are from an Excel background, you are used to cells, columns, and sheets. If you are coming from SQL, you are used to tables and relations. In the Pandas ecosystem, these concepts are distilled into two primary data structures: the **Series** and the **DataFrame**.

This chapter explains these structures. We examine how they are created, why their axes are labeled, and the concept of index immutability that makes Pandas both fast and reliable in production systems.

Run Pandas Playbook (code + datasets): https://github.com/Inder-P-Singh/pandas-playbook

Quick check: clone the playbook and run the smoke script:

```
git clone https://github.com/Inder-P-Singh/pandas-playbook.git

python scripts/run_quick_tests.py  # expected: ALL_CHECKS_PASS
```

The Pandas Series

A Series is the fundamental building block of Pandas. It is a one-dimensional labeled array capable of holding data of any type, including integers, strings, floating-point numbers, and Python objects.

You can think of a Series as a hybrid of a Python list and a Python dictionary. Like a list, it maintains order and stores a sequence of values. Like a dictionary, it uses keys, called an index, which allow data retrieval by label instead of only by position.

Creation and Structure

To work with Pandas, we first import it using the standard convention. NumPy is also commonly imported alongside Pandas.

Copyright © 2026 All Rights Reserved.

```
import pandas as pd

import numpy as np
```

Creating a Series is straightforward. It can be created from a list, a NumPy array, or a dictionary.

Example: Creating a Series with a Default Index

```
data = [100, 200, 300, 400]

s = pd.Series(data)

print(s)
```

Output:

```
0    100
1    200
2    300
3    400
dtype: int64
```

The left column, 0 through 3, represents the Index. Because no index was explicitly provided, Pandas generated a default integer index.

The Power of Custom Indexing

The real strength of a Series appears when you define a custom index. A custom index attaches semantic meaning to the values, making the data easier to reason about and safer to manipulate.

```
revenue = [4500, 6200, 5800]
months = ['January', 'February', 'March']

revenue_series = pd.Series(revenue, index=months)
print(revenue_series)
print(f"\nRevenue for February: {revenue_series['February']}")
```


Copyright © 2026 All Rights Reserved.

Output:

```
January     4500
February    6200
March       5800
dtype: int64

Revenue for February: 6200
```

Unlike a NumPy array, where you must remember that index position 1 corresponds to February, a Pandas Series maintains this mapping internally. This dramatically reduces logical errors in analysis pipelines.

To get the latest AI and ML free documents, follow Inder P Singh (11.3K Followers, 1.8M Views) on LinkedIn at https://www.linkedin.com/in/inderpsingh/

The Pandas DataFrame

If a Series represents a single column, a DataFrame represents an entire table or spreadsheet. It is a two-dimensional, size-mutable, tabular data structure with labeled axes for both rows and columns.

Internally, a DataFrame can be understood as a collection of Series objects that all share the same index.

Anatomy of a DataFrame

A DataFrame is composed of three core components. The data, which holds the actual values; the index, which labels the rows; and the columns, which label the fields.

Creation from a Dictionary

The common construction method is using a dictionary of lists. The dictionary keys become column names, and the lists become column data.

Copyright © 2026 All Rights Reserved.

```
data = {
    'Product': ['Laptop', 'Mouse', 'Monitor'],
    'Price': [1200, 25, 300],
    'Stock': [50, 200, 75]
}
df = pd.DataFrame(data)
print(df)
```

Output:

	Product	Price	Stock
0	Laptop	1200	50
1	Mouse	25	200
2	Monitor	300	75

Each column accessed using df['Product'] or df['Price'] returns a Series, and all Series align automatically using the shared index.

1.3 Labeled Axes and Alignment

One of the defining advantages of Pandas over NumPy is automatic data alignment. When performing operations across Series or DataFrames, Pandas aligns data based on labels rather than positional order.

Example: Automatic Alignment

```
s1 = pd.Series([10, 20], index=['a', 'b'])
s2 = pd.Series([30, 40], index=['b', 'a'])
total = s1 + s2
print(total)
```


Copyright © 2026 All Rights Reserved.

Output:

```
a    50
b    50
dtype: int64
```

Even though the order of labels differs between s1 and s2, Pandas matches values by label. This prevents subtle and dangerous analytical errors when working with real-world datasets from multiple sources.

To get production-grade AI/ML code templates with playbooks, message Inder P Singh

Explore the Pandas Playbook (code + datasets):
https://github.com/Inder-P-Singh/pandas-playbook

```
git clone https://github.com/Inder-P-Singh/pandas-playbook.git

cd pandas-playbook

python3 -m venv venv

source venv/bin/activate  # On Windows, use `venv\\Scripts\\activate`

pip install -r requirements.txt

python scripts/run_quick_tests.py

# Expected output: ALL_CHECKS_PASS

python programs/01_quickstart_basic.py

python programs/02_cleaning_and_dtypes.py

python programs/03_groupby_merge_pivot.py

python programs/04_time_series_and_resample.py

python scripts/example_report.py
```


Copyright © 2026 All Rights Reserved.

Index Immutability Concept

Although the data inside a Series or DataFrame can be modified, the Index object itself is immutable. This architectural choice is deliberate and critical.

Immutability improves safety by preventing accidental index corruption and improves performance by allowing the index to be hashable, which enables fast lookups.

You can replace an index entirely, but you cannot modify individual index elements in place.

```
df = pd.DataFrame({'A': [1, 2]}, index=['x', 'y'])
df.index = ['a', 'b']
```

Attempting the following will raise a TypeError:

```
df.index[0] = 'z'
```

Understanding this distinction is essential when debugging index-related errors during data preprocessing.

Software and Testing Training | For personal reference

Copyright © 2026 All Rights Reserved.

To learn Pandas coding, use the Pandas Playbook on GitHub.

Pandas Playbook

This repository contains the Pandas Playbook, designed for data analysts and ML engineers (Beginner → Intermediate) to master essential Pandas operations.

Features

- **Reproducible Python Programs:** Step-by-step guides for data loading, cleaning, transformation, and visualization.
- **Sample Data:** Small, synthetic CSV files for hands-on practice.
- **Utility Scripts:** Examples for running basic data checks and generating reports outside of Programs.
- **Automated Tests:** A Pytest suite to verify core functionalities and generated outputs.
- **Comprehensive Playbook:** A markdown guide (`playbook.md`) covering concepts, lab exercises, and next steps.

Setup Instructions

1. Clone the Repository:

```
git clone https://github.com/Inder-P-Singh/pandas-playbook.git
cd pandas-playbook
```

2. Create a Virtual Environment (recommended):

```
python3 -m venv venv
source venv/bin/activate  # On Windows, use `venv\\Scripts\\activate`
```

3. Install Required Packages:

```
pip install -r requirements.txt
```

Running the Playbook

1. Run Quick Tests

Software and Testing

Copyright © 2026 All Rights Reserved.

Pandas Interview Questions on Series and DataFrames

Q1: What is the primary difference between a NumPy array and a Pandas Series?
Answer: A NumPy array relies on positional indexing, while a Pandas Series adds an explicit index that supports label-based access and automatic alignment during operations.

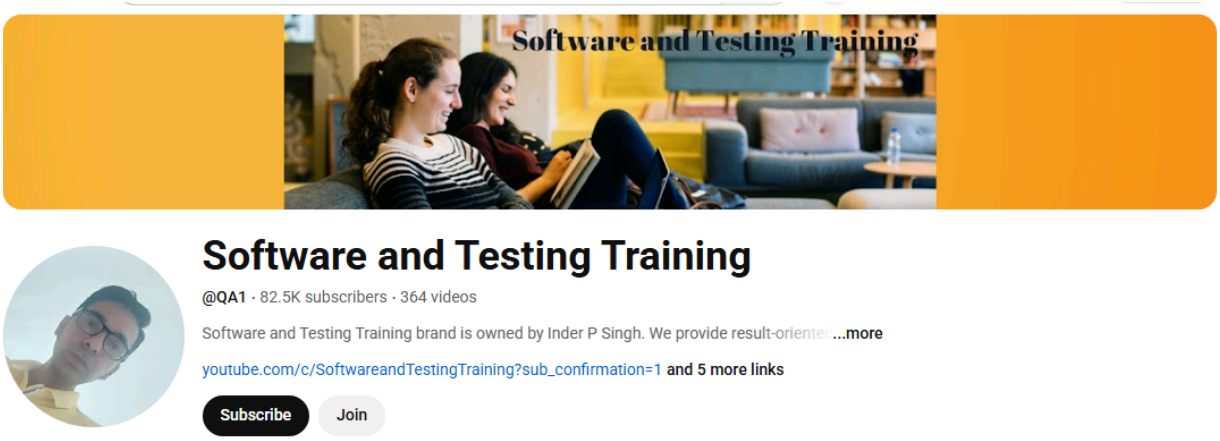
Q2: Can a Pandas DataFrame store multiple data types?
Answer: Yes. Each column is a Series and must be homogeneous for performance reasons, but different columns in a DataFrame can have different data types.

Q3: Why is index immutability important in Pandas?
Answer: It ensures index integrity, prevents side effects across shared objects, and allows Pandas to optimize data access using hash-based mechanisms.

Summary

In this chapter, we examined the architectural foundations of Pandas. We learned that a Series is a labeled one-dimensional structure, a DataFrame is a two-dimensional table composed of Series, and the Index acts as an immutable backbone that enables alignment and safe operations.

View Software and Testing Training YouTube channel (364 tutorials, since 2013)
https://youtube.com/c/SoftwareandTestingTraining/videos



Copyright © 2026 All Rights Reserved.

Chapter 2: Data Ingestion and Exploratory Analysis

In real-world systems, data almost never originates inside Python code. It typically arrives from CSV files, Excel workbooks, or enterprise databases. This chapter focuses on data ingestion and exploratory data analysis, the critical first step before cleaning or modeling.

Reading External Data

Pandas provides a rich set of I/O utilities, commonly prefixed with pd.read_. These functions allow precise control over parsing behavior during ingestion.

CSV Files

CSV is the most common data interchange format in analytics. It is lightweight, human-readable, and widely supported.

```
df = pd.read_csv('sales_data.csv')
```

Handling variations during ingestion:

```
df_custom = pd.read_csv(
    'sales_data_v2.csv',
    sep=';',
    index_col='ID',
    parse_dates=['Date']
)
```

If a UnicodeDecodeError occurs, the file encoding is likely non-UTF-8. Specifying encodings such as latin1 or cp1252 usually resolves this issue.

Excel Files

Excel remains dominant in business environments. Pandas reads Excel files through engines such as openpyxl.

Software and Testing Training _ For personal reference

Copyright © 2026 All Rights Reserved.

```
df_excel = pd.read_excel(
    'financial_report.xlsx',
    sheet_name='Q1_Results',
    header=0
)
```

SQL Databases

For enterprise-scale data, Pandas integrates with SQLAlchemy to execute queries directly against databases.

```
from sqlalchemy import create_engine

engine = create_engine('sqlite:///my_database.db')
query = "SELECT * FROM customers WHERE region = 'North America'"
df_sql = pd.read_sql(query, engine)
```

View AI / ML Tutorials
https://www.youtube.com/playlist?list=PLc3SzDYhhiGWsHYGhnGwGSX3C-EXUmotP

Inspection Techniques

After ingestion, inspection is mandatory. You need to understand size, structure, and data types before proceeding.

Using head and tail:
`print(df.head())`

`print(df.tail(3))`

Example: If you run `01_quickstart_basic.py` from the Pandas Playbook, the output will be:

Copyright © 2026 All Rights Reserved.

```
--- DataFrame Head ---
   order_id  customer_id  order_date  product     category  amount region
0      1000          105  2023-08-25  Monitor  Electronics  262.19   East
1      1001          114  2023-01-23    Mouse  Peripherals  231.83  South
2      1002          112  2023-07-28  Monitor  Electronics  163.78  North
3      1003          113  2023-06-25   Laptop  Electronics  283.47  South
4      1004          116  2023-01-08  Monitor  Electronics   73.10   West
```

Using shape:

```
rows, cols = df.shape

print(f"We have {rows} rows and {cols} columns.")
```

Using info:

```
df.info()
```

Example from Pandas Playbook:

```
--- DataFrame Info ---
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 55 entries, 0 to 54
Data columns (total 7 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   order_id     55 non-null     int64
 1   customer_id  55 non-null     Int64
 2   order_date   55 non-null     object
 3   product      55 non-null     object
 4   category     55 non-null     object
 5   amount       55 non-null     float64
 6   region       55 non-null     object
dtypes: Int64(1), float64(1), int64(1), object(4)
memory usage: 3.2+ KB
```

The info output immediately highlights missing values, incorrect data types, and memory usage, all of which inform downstream cleaning decisions.

Pandas Interview Questions and Answers: Ingestion and Inspection

Copyright © 2026 All Rights Reserved.

Q1: Which is faster, read_csv or read_excel, and why?
Answer: read_csv is faster because it parses plain text, whereas read_excel must process complex spreadsheet formats.

Q2: Why do all values appear in a single column after loading a CSV?
Answer: The delimiter in the file does not match the default comma delimiter. The correct separator must be specified using the sep parameter.

Q3: How do you process a CSV file that does not fit into memory?
Answer: Use the chunksize parameter to iterate over the file in manageable portions.

```
for chunk in pd.read_csv('massive_file.csv', chunksize=10000):
    process(chunk)
```

You now know how to ingest data from external sources and validate its structure using inspection tools. These steps form the foundation for reliable data pipelines.

Note: If you want working AI POC, dataset or AI demo for your portfolio, message Inder P Singh

Software and Testing Training | Fourth Industrial Revolution

Copyright © 2026 All Rights Reserved.

Chapter 3: Extraction – Selection and Filtering

After ingestion and inspection, meaningful analysis requires extracting specific subsets of data. This chapter focuses on precise data selection using Pandas indexing and filtering mechanisms.

The Indexing Dilemma: Explicit vs Implicit

Basic bracket and dot notation are limited. For robust two-dimensional selection, Pandas provides loc and iloc.

Label-Based Indexing with loc

loc selects data based on index and column labels.

```
data = {
    'Department': ['IT', 'HR', 'IT', 'Sales', 'Marketing'],
    'Salary': [85000, 60000, 78000, 92000, 65000],
    'Status': ['Active', 'Active', 'On Leave', 'Active', 'Contract']
}


df = pd.DataFrame(data, index=['Alice', 'Bob', 'Charlie', 'Diana', 'Evan'])
charlie_data = df.loc['Charlie', ['Salary', 'Status']]
```

Slicing with loc is inclusive of the stop label.

Example from Pandas Playbook: `print(sales_df.loc[0:2])`

```
First 3 rows (labels 0 to 2 inclusive) using .loc:
   order_id  customer_id  order_date  product     category  amount region
0      1000          105  2023-08-25  Monitor  Electronics  262.19   East
1      1001          114  2023-01-23    Mouse  Peripherals  231.83  South
2      1002          112  2023-07-28  Monitor  Electronics  163.78  North
```


Copyright © 2026 All Rights Reserved.

Integer-Based Indexing with iloc

iloc selects data purely by position.

```
subset = df.iloc[0:3, 0:2]
```

Here, slicing is exclusive of the stop bound, following standard Python behavior.

Example from Pandas Playbook: `print(sales_df.iloc[[0, 2, 4]])`

```
Rows at index positions 0, 2, 4 using .iloc:
   order_id  customer_id  order_date  product     category  amount region
0      1000          105  2023-08-25  Monitor  Electronics  262.19   East
2      1002          112  2023-07-28  Monitor  Electronics  163.78  North
4      1004          116  2023-01-08  Monitor  Electronics   73.10   West
```

For Gen AI and ML understanding, view Fourth Industrial Revolution blog articles at https://fourth-industrial-revolution.blogspot.com/

Conditional Filtering

Boolean indexing allows selection based on conditions.

```
high_earners = df[df['Salary'] > 70000]
```

Multiple conditions require bitwise operators:

```
it_active = df[(df['Department'] == 'IT') & (df['Status'] == 'Active')]
```


Copyright © 2026 All Rights Reserved.

The query Method

query provides SQL-like readability for complex filters.

```
it_active = df.query("Department == 'IT' and Status == 'Active'")

min_salary = 80000

high_salary_sales = df.query("Department == 'Sales' and Salary > @min_salary")
```

Pandas Interview Questions: Selection and Filtering

Q1: What is the slicing difference between loc and iloc?
Answer: loc includes the stop label, while iloc excludes the stop index.

Q2: Why does using and instead of & fail in Pandas filters?
Answer: and evaluates entire objects, which is ambiguous for Series. Pandas requires element-wise operators such as & and |.

Q3: How do you filter rows based on multiple allowed values?
Answer: Use the isin method for clean and efficient membership testing.

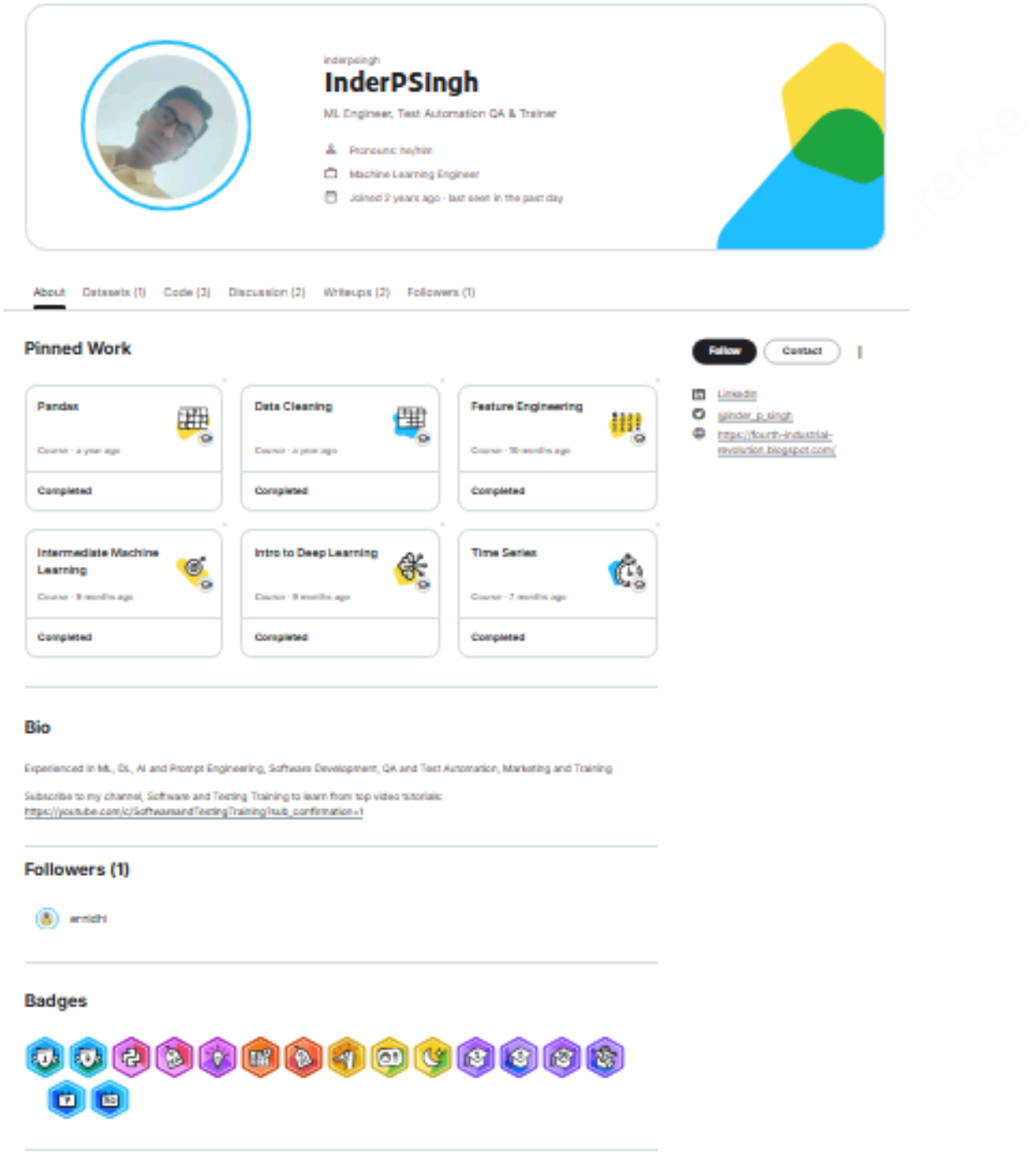
Summary

You now have precise control over what data enters your analysis. Mastery of loc, iloc, boolean masking, and query is essential for scalable analytics and machine learning workflows.

Software and Testing Training - For personal reference

Copyright © 2026 All Rights Reserved.

To get Pandas notebooks and other notebooks and datasets, follow Inder P Singh on Kaggle (16 Badges & 16 Certifications)



Copyright © 2026 All Rights Reserved.

Chapter 4: Data Cleaning and Sanitation

In Chapter 3, we mastered slicing and filtering. The uncomfortable truth is that real-world data is messy. Dates saved as text, prices with currency symbols, duplicate customer records, and missing values routinely derail analysis pipelines. View the types of Data Quality issues in the Data Quality video. The principle of GIGO—Garbage In, Garbage Out—applies: a model trained on flawed inputs produces flawed outputs. This chapter focuses on data sanitation, the disciplined process of identifying and correcting errors so datasets become reliable inputs for downstream models and reports.

4.1 The Invisible Enemy: Missing Data (NaN)

In Pandas, missing scalar values are typically represented as `NaN` for floating types or `<NA>` / `None` for nullable dtypes. These placeholders indicate absent information.

Detecting missing values is the first step. Use the `isna()` or `isnull()` methods to create a boolean mask, and aggregate with `sum()` to see counts per column.

Example: Detect missing values and counts

```
import pandas as pd

import numpy as np


data = {

    'Name': ['Alice', 'Bob', 'Charlie', 'Alice', np.nan],

    'Age': [25, np.nan, 30, 25, 22],

    'Salary': ['$50,000', '$60,000', np.nan, '$50,000', '$45,000'],

    'Join_Date': ['2023-01-01', '2023-01-15', 'Unknown', '2023-01-01',
'2023-02-20']

}
```


Copyright © 2026 All Rights Reserved.

```
df = pd.DataFrame(data)


print(df.isna())          # Boolean mask of missing values

print(df.isna().sum())    # Count missing per column
```

Once you locate missing values, decide whether to drop rows or impute values. Deletion is appropriate when missing values are rare or the missing field is critical and irrecoverable. Imputation is required when the data is scarce or the missing variable can be reasonably estimated.

Example: Dropping and filling

```
# Drop any row containing at least one missing value

df_clean_drop = df.dropna()


# Drop rows only if ALL columns are missing

df_clean_drop_all = df.dropna(how='all')


# Fill missing Age with the mean

mean_age = df['Age'].mean()

df['Age'] = df['Age'].fillna(mean_age)


# Forward fill (useful for time series)

df['Age'] = df['Age'].ffill()
```


Copyright © 2026 All Rights Reserved.

Example from Pandas Playbook

```
DataFrame head after initial cleaning:
   order_id  customer_id  order_date  product     category  amount region  amount_filled region_filled
0      1000          105  2023-08-25  Monitor  Electronics  262.19   East         262.19          East
1      1001          114  2023-01-23    Mouse  Peripherals  231.83  South         231.83         South
2      1002          112  2023-07-28  Monitor  Electronics  163.78  North         163.78         North
3      1003          113  2023-06-25   Laptop  Electronics  283.47  South         283.47         South
4      1004          116  2023-01-08  Monitor  Electronics   73.10   West          73.10          West
```

Note: To get deep dive AI/Gen AI/ML hands-on training and interview coaching, message Inder P Singh (6 years experience in AI & ML) on LinkedIn.

Handling Duplicates

Duplicate records inflate counts and bias statistics. Identify duplicates with `duplicated()` and remove them with `drop_duplicates()`.

Example: Identify and remove duplicates

```python
duplicate_count = df.duplicated().sum()
print(f"Number of duplicate rows: {duplicate_count}")

# Remove duplicates, keeping the first occurrence
df_no_dupes = df.drop_duplicates(keep='first')

# Drop duplicates based on subset of columns, e.g., Name only
df_no_dupes_name = df.drop_duplicates(subset=['Name'])
```

Subtle duplicates occur when identifiers differ but the meaningful fields match. Choose the subset argument carefully and consider a deterministic deduplication rule (for example, keep the most recent record or the one with the most complete fields).

Correcting Data Types

A column that looks numeric may be stored as `object` because of symbols or mixed values. You cannot perform arithmetic on strings; convert columns to appropriate numeric or datetime types after cleaning.

Copyright © 2026 All Rights Reserved.

`astype()` works when values are already compatible with the target dtype. For messy strings or dates, use helper functions like `pd.to_numeric()` and `pd.to_datetime()` with `errors='coerce'` to transform unparseable values into missing indicators.

Example: Convert salary strings to numeric and parse dates

```
# Remove currency symbols and thousands separators
df['Salary'] = df['Salary'].str.replace('$', '', regex=False).str.replace(',', '', regex=False)

# Convert to numeric; invalid parsing becomes NaN
df['Salary'] = pd.to_numeric(df['Salary'], errors='coerce')

# Parse dates; unparseable values become NaT
df['Join_Date'] = pd.to_datetime(df['Join_Date'], errors='coerce')
```

Avoid applying `astype(int)` directly when missing values may exist; use `pd.to_numeric(..., downcast=...)` or nullable integer dtypes where appropriate.

Standardizing Text

Text fields vary in casing and whitespace; normalization reduces cardinality and improves merge success. Use the `.str` accessor to vectorize string operations across a Series.

Example: Normalize name strings and split full names

```
# Standardize casing and strip whitespace
df['Name'] = df['Name'].str.title().str.strip()

# If Full_Name exists, split into First and Last
# df[['First', 'Last']] = df['Full_Name'].str.split(' ', expand=True)
```

Other common operations include `str.replace()` for pattern cleanup, `str.contains()` for filtering, and `str.extract()` for regex captures. When working with large text corpora, consider using optimized backends or converting low-cardinality strings to `category` to reduce memory.

Interview Questions and Answers: Data Cleaning and Sanitation

Copyright © 2026 All Rights Reserved.

Q1: When should you drop rows with missing target values instead of imputing them?
Answer: Drop rows with missing target values when the target is essential for supervised learning and cannot be reliably inferred. If the proportion of missing targets is small and missingness is unrelated to predictors (MCAR), dropping is safe. If missingness is systematic, dropping can bias the model; consider strategies like modeling missingness or collecting labels.

Q2: How do nullable dtypes affect imputation strategies?
Answer: Nullable dtypes (for example `Int64`, `Boolean`, `String`) allow preserving the semantic dtype while representing missing values as `<NA>`. This lets you distinguish between missing and valid sentinel values and choose imputation methods that respect original dtypes, avoiding implicit dtype promotion.

Q3: What is a safe strategy to deduplicate records when no single primary key exists?
Answer: Define a deterministic rule that ranks candidate duplicates (for example, completeness score, most recent timestamp), group by the fields that define identity, and retain the top-ranked row per group using `.sort_values(...).drop_duplicates(subset=..., keep='first')`.

Software and Testing Training

Copyright © 2026 All Rights Reserved.

Chapter 5: Grouping, Merging, and Reshaping

In Chapter 4 we cleaned the inputs. Clean data is necessary but not sufficient. Often you must summarize, combine, or reshape data into the form that addresses the analytical question. This chapter covers three transformation families: aggregation via grouping, combining multiple datasets, and reshaping data for reporting or modeling.

The Split-Apply-Combine Strategy

The most common analytical question asks about groups: average salary by department or total sales by region. Pandas implements this through the Split-Apply-Combine pattern with `.groupby()`.

Example: Basic grouping and aggregation

```
import pandas as pd

data = {

    'Department': ['Sales', 'Tech', 'Sales', 'Tech', 'HR'],

    'Employee': ['Alice', 'Inder', 'Charlie', 'David', 'Eve'],

    'Revenue': [5000, 7000, 4500, 8000, 2000]

}

df = pd.DataFrame(data)


# Group by Department and sum Revenue

dept_revenue = df.groupby('Department')['Revenue'].sum()

print(dept_revenue)


# Multiple aggregations
```

Software and Testing Training - For personal reference

Copyright © 2026 All Rights Reserved.

```
summary = df.groupby('Department')['Revenue'].agg(['sum', 'mean',
'count'])

print(summary)
```

Remember that `groupby` often returns the grouping column as an index. Use `as_index=False` or `.reset_index()` to keep it as a column when needed.

Example from Pandas Playbook

```
Regional Sales Summary:
    region  total_sales  order_count
0  Central      1095.14            7
1     East      2198.29           12
2    North      1547.05           10
3    South      2098.45           11
4     West      2304.05           15
```

Combining Datasets: Merging and Concatenation

Data frequently lives across files. Use `pd.concat()` to stack DataFrames with the same columns and `pd.merge()` to join on keys, analogous to SQL joins. Specify the `how` argument explicitly and consider `validate` to enforce expected cardinalities.

Example: Concatenate and merge

```
# Concatenation (stack monthly reports)

df_jan = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})

df_feb = pd.DataFrame({'A': [5, 6], 'B': [7, 8]})

yearly_df = pd.concat([df_jan, df_feb], axis=0, ignore_index=True)


# Merge (SQL-like join)
```

Software Testing Training - For personal reference

Copyright © 2026 All Rights Reserved.

```
users = pd.DataFrame({'UserID': [1, 2, 3], 'Name': ['Alice', 'Bob',
'Charlie']})

orders = pd.DataFrame({'OrderID': [101, 102], 'UserID': [1, 2],
'Amount': [250, 150]})

merged_df = pd.merge(orders, users, on='UserID', how='left')
```

When merging, always reason about missing matches and whether `left`, `inner`, or `outer` is appropriate. Use `validate='m:1'` or similar to catch unexpected multiplicities that could silently inflate rows.

Example from [Pandas Playbook]

```
Merged (Left Join) sample:
   order_id  customer_id  order_date  product     category  amount region          name signup_date segment
0      1000          105  2023-08-25  Monitor  Electronics  262.19   East  Customer 105  2022-02-23  Retail
1      1001          114  2023-01-23    Mouse  Peripherals  231.83  South  Customer 114  2022-09-08  Retail
2      1002          112  2023-07-28  Monitor  Electronics  163.78  North  Customer 112  2022-03-10  Retail
3      1003          113  2023-06-25   Laptop  Electronics  283.47  South  Customer 113  2023-06-03     SMB
4      1004          116  2023-01-08  Monitor  Electronics   73.10   West  Customer 116  2022-05-09  Retail
Rows (left join): 55
Missing customer names (left join): 5

Merged (Inner Join) sample:
   order_id  customer_id  order_date  product     category  amount region          name signup_date segment
0      1000          105  2023-08-25  Monitor  Electronics  262.19   East  Customer 105  2022-02-23  Retail
1      1001          114  2023-01-23    Mouse  Peripherals  231.83  South  Customer 114  2022-09-08  Retail
2      1002          112  2023-07-28  Monitor  Electronics  163.78  North  Customer 112  2022-03-10  Retail
3      1003          113  2023-06-25   Laptop  Electronics  283.47  South  Customer 113  2023-06-03     SMB
4      1004          116  2023-01-08  Monitor  Electronics   73.10   West  Customer 116  2022-05-09  Retail
Rows (inner join): 50
Saved 'outputs/sales_customer_merged_inner.csv'
```

Example: Pivot and melt

```
df_sales = pd.DataFrame({

    'Date': ['2025-01-01', '2025-01-01', '2025-01-02', '2025-01-02'],

    'Region': ['North', 'South', 'North', 'South'],

    'Sales': [100, 200, 150, 250]

})
```


Copyright © 2026 All Rights Reserved.

```
pivot = df_sales.pivot_table(values='Sales', index='Date',
columns='Region', aggfunc='sum')

melted = pivot.reset_index().melt(id_vars='Date', value_vars=['North',
'South'])
```

Example from [Pandas Playbook]

```
Pivot Table: Total Sales by Region and Category
category  Accessories  Electronics  Peripherals
region
Central         94.97       227.40       772.77
East           271.69      1406.54       520.06
North          327.46       770.43       449.16
South          226.40       900.55       971.50
West           229.54      1775.11       299.40
```

If duplicates exist for a given pivot index/column pair, `.pivot()` raises an error; use `.pivot_table()` with `aggfunc` to aggregate duplicates.

Interview Questions and Answers: Transformation

Q1: What is the difference between `merge()` and `concat()`?
Answer: `concat()` appends DataFrames along a given axis, suitable when structures align. `merge()` joins DataFrames by column keys, equivalent to SQL joins, and combines rows based on matching key values.

Q2: How do you handle duplicate index/column combinations when pivoting?
Answer: Use `pivot_table()` with an `aggfunc` parameter to aggregate duplicates, for example `aggfunc='sum'` or a custom aggregation function.

Q3: What is the difference between `transform()` and `apply()` on a `groupby` object?
Answer: `transform()` returns an output with the same index length as the original dataframe, enabling adding group-level statistics back to rows. `apply()` can return a reduced-dimensional result or custom structure, and is more flexible but often less performant.

Summary of Chapter 5

Copyright © 2026 All Rights Reserved.

You can now convert row-level events into grouped summaries, combine disparate sources safely, and reshape data for downstream requirements.

© Inder P Singh (401+ professionals trained including AI and ML) is available for practical coaching and project guidance.

Software and Testing Training – For personal reference

Copyright © 2026 All Rights Reserved.

Chapter 6: Pandas 2.0, PyArrow, and Optimization

Chapter 5 taught structural transformations. At scale, performance becomes the gating factor. Pandas 2.0 and the Arrow ecosystem introduced new backends and dtypes that address long-standing memory and speed limitations. This chapter explains the practical implications and optimization strategies for production workloads.

PyArrow

Historically, Pandas stored strings as Python objects, incurring large memory overhead and slow vectorized string operations. Apache Arrow provides a columnar in-memory representation optimized for modern CPUs and language interoperability. Pandas 2.0 exposes Arrow-backed dtypes and a `dtype_backend` option that can materially reduce memory usage and speed up string operations.

Example: Read CSV with PyArrow backend

```
import pandas as pd


# Standard read (NumPy-backed)

df_numpy = pd.read_csv("massive_dataset.csv")


# Arrow-backed read for better memory and string performance

# Requires pandas>=2.0 and pyarrow installed; fallback to default
engine otherwise.

df_arrow = pd.read_csv("massive_dataset.csv", dtype_backend="pyarrow")

print(df_arrow.dtypes)

# Example output: customer_id int64[pyarrow], customer_name
string[pyarrow], purchase_amount double[pyarrow]
```


Copyright © 2026 All Rights Reserved.

Using `dtype_backend="pyarrow"` can reduce memory usage significantly for string-heavy datasets and accelerate many text operations.

Nullable Data Types (Solving the NaN Problem)

NumPy-based Pandas forced integer columns with missing values to `float64`. Nullable dtypes such as `Int64`, `Float64`, `Boolean`, and `String` preserve logical types while representing missing values as `<NA>`. This avoids unintended dtype promotion and keeps identifiers or categorical integers intact.

Example: Nullable integer series

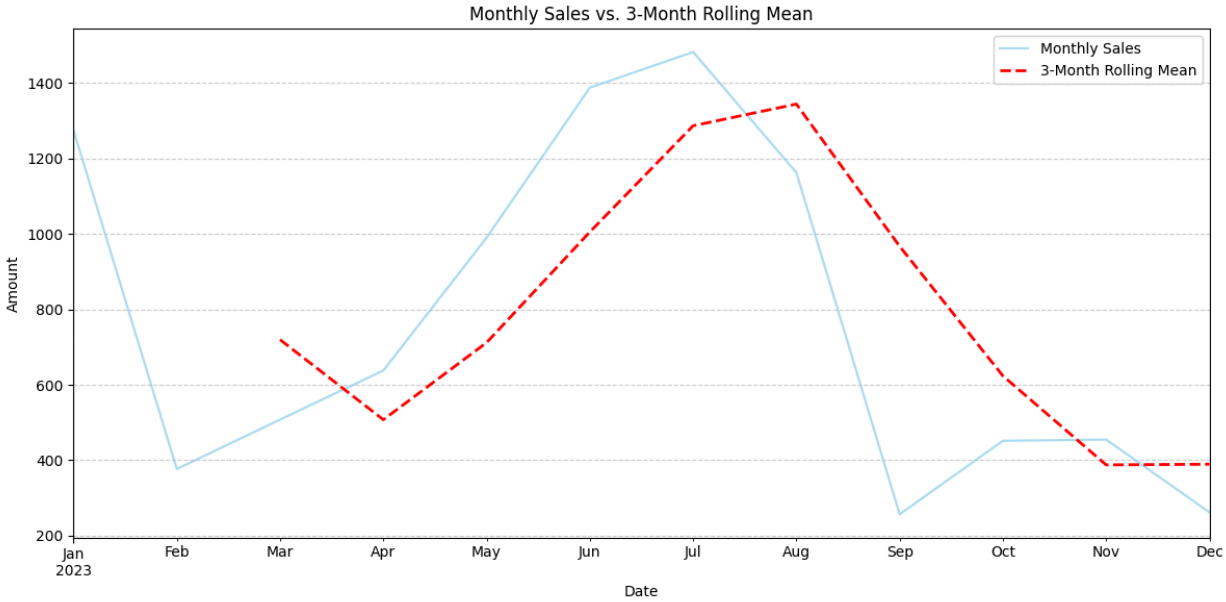
```
s_new = pd.Series([1, 2, 3], dtype="Int64")

s_new[0] = None

print(s_new)

# 0    <NA>

# 1       2

# 2       3

# dtype: Int64
```

Nullable dtypes improve semantic correctness and reduce surprises in downstream processing.

Software and Testing Training – For personal reference only

Copyright © 2026 All Rights Reserved.

Example from [Pandas Playbook]



Copy-on-Write (CoW)

Ambiguity between views and copies led to silent bugs and the `SettingWithCopyWarning`. Copy-on-Write makes slice semantics predictable: slices share memory until a write occurs, at which point a deep copy is created just-in-time.

Example: Enable Copy-on-Write and illustrate safety

```
pd.options.mode.copy_on_write = True


df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})

subset = df[["A"]]

subset.iloc[0, 0] = 99


print(df.iloc[0, 0])     # 1 (original unchanged)

print(subset.iloc[0, 0]) # 99 (subset modified)
```


Copyright © 2026 All Rights Reserved.

CoW improves both memory efficiency and safety by ensuring that modifications to slices do not unexpectedly mutate the original.

Optimization Strategies for Big Data

Performance is a multi-faceted problem that includes I/O, memory representation, and algorithmic choices. Recommended practical tactics include using Arrow-backed I/O, converting low-cardinality strings to `category`, downcasting numeric types when appropriate, selecting useful columns at load time with `usecols`, and processing in chunks.

Example: Read large file with PyArrow engine and chunking

```
# Faster reading using PyArrow engine where available

df = pd.read_csv("large_file.csv", engine="pyarrow")


# Chunking for files larger than memory

chunk_iter = pd.read_csv('massive_file.csv', chunksize=10000)

for chunk in chunk_iter:

    process(chunk)
```

Also consider `memory_usage(deep=True)` to diagnose where memory is spent and adjust dtypes accordingly.

Interview Questions: The Modern Era

Q1: What is the difference between `NaN` and `<NA>`?
Answer: `NaN` is the IEEE floating-point missing value used by NumPy; it coerces integer columns to floats. `<NA>` is the missing-value scalar used by Pandas nullable dtypes (for example `Int64`), which preserves the logical dtype while indicating missingness.

Q2: Why prefer `string[pyarrow]` over `object` dtype for text?
Answer: `object` stores pointers to Python strings with high memory overhead and poor cache

Copyright © 2026 All Rights Reserved.

locality. `string[pyarrow]` stores text in a contiguous, columnar memory layout with efficient vectorized operations and lower memory use.

Q3: How does Copy-on-Write affect chained assignment and the `SettingWithCopyWarning`?

Answer: Copy-on-Write defers actual copying until a write occurs. It ensures a slice behaves as a reference until modified, at which point Pandas safely copies data for the write. This removes ambiguity and eliminates the need for the `SettingWithCopyWarning` in typical scenarios.

Q4: When is `dtype_backend="pyarrow"` not the right choice?

Answer: When you require maximum compatibility with older code that depends on NumPy-backed internal representations or third-party libraries that expect NumPy dtypes. Also evaluate performance trade-offs per workload; for some numeric-only workloads, the default backend remains adequate.

Summary of Chapter 6

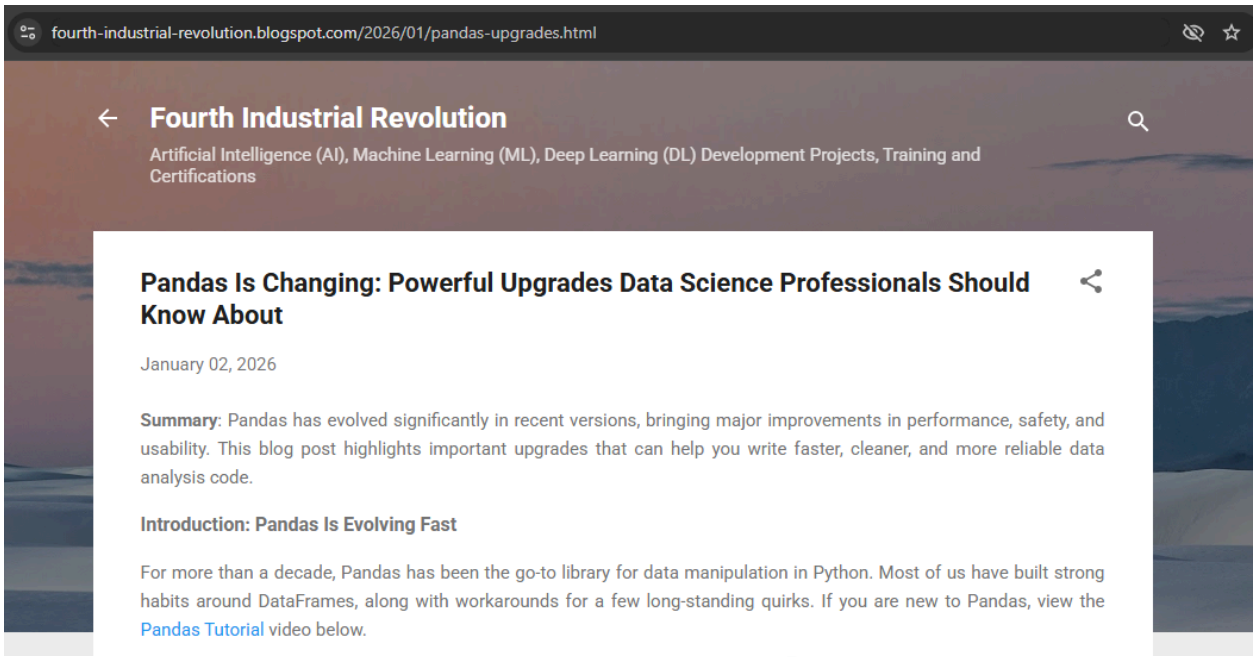
Pandas 2.0 and the Arrow ecosystem move Pandas from a single-process, NumPy-centric tool toward a more modern, memory-efficient, and interoperable platform. Nullable dtypes, Arrow-backed strings, and CoW are production-grade primitives for large-scale tabular processing.

© Inder P Singh (10 AI ML projects successfully delivered) offers practical interview and project coaching.

Software and Testing Training (for personal use only)

Copyright © 2026 All Rights Reserved.

Read more about Pandas Upgrades on the blog.



Software and Testing Training

Copyright © 2026 All Rights Reserved.

Chapter 7: Interview Questions and Best Practices

This chapter synthesizes technical mastery into interview readiness. Interviewers evaluate technical correctness, efficiency, and the candidate's ability to reason about trade-offs. Demonstrating an understanding of when to use a technique is as important as knowing how to use it.

Conceptual Architecture

Vectorization is more than "faster." It enables compiled execution, SIMD use, improved cache locality, and avoids Python-level type checks per element. When asked about vectorization, explain these performance mechanisms.

The view versus copy distinction matters. A View is a window to the same memory; a Copy duplicates data. Best practice: use .loc for assignment clarity and .copy() when a true independent copy is required.

Practical Coding Scenarios

Selecting the top-performing item per group is a common interview question. Efficient approaches avoid Python loops and use groupby with nlargest, transform, or sort_values().groupby(...).head(n).

Example: Top performer per region

```
import pandas as pd


data = {
    'Region': ['North', 'North', 'North', 'South', 'South', 'South'],
    'Product': ['A', 'B', 'C', 'A', 'B', 'C'],
    'Sales': [100, 150, 90, 200, 120, 300]
}
```

Software and Testing Training – For personal reference

Copyright © 2026 All Rights Reserved.

```
df = pd.DataFrame(data)


# Readable approach

top_performers = df.sort_values('Sales',
ascending=False).groupby('Region').head(1)


# Using nlargest

top_performers_alt = df.groupby('Region',
group_keys=False).apply(lambda x: x.nlargest(1, 'Sales'))
```

Joining DataFrames safely requires explicit how and, where applicable, validate to ensure merges meet expected cardinalities and avoid accidental multiplicative joins.

Optimization and Memory Management
When the dataset exceeds RAM, apply chunking, selective column loading with usecols, dtype optimization (categorical conversion and numeric downcasting), and consider Arrow-backed reading and computation. Also profile memory with memory_usage(deep=True) and test on representative subsets.

Best Practices for Production Code
Method chaining improves readability and reduces intermediate variables. Idempotency matters: cleaning logic should be repeatable without changing semantics on multiple runs. Avoid data leakage by computing imputations and scalers only on training data during ML workflows.

Interview Questions and Answers: Career Readiness

Q1: Explain vectorization and why it matters beyond raw speed.
Answer: Vectorization delegates work to optimized C routines and leverages CPU features such

Copyright © 2026 All Rights Reserved.

as SIMD and cache-friendly memory layouts. It eliminates Python-level per-element overhead and reduces runtime type checks, resulting in both faster and more predictable performance.

Q2: How would you prevent data leakage when imputing missing values for a machine learning model?
Answer: Fit imputation parameters (for example, mean or median) on the training set only. Apply those learned parameters to validation and test sets. Implement the imputation step in a pipeline so the same transformations are consistently applied and retraining does not leak information from validation/test sets.

Q3: What steps would you take when merging datasets and you notice row counts expand unexpectedly?
Answer: Re-evaluate the join key and its cardinality in both tables. Use validate in merge() to assert expected relationships, inspect duplicate keys, and consider deduplicating or aggregating prior to the merge. Choose the appropriate join type (left, inner, outer) for the business requirement.

Q4: Describe code simplicity choices you would make.
Answer: Write clear, well-commented code; prefer explicitness over cleverness; use method chaining thoughtfully with comments explaining complex chains; include minimal reproducible tests or assertions; and deliver a small README describing assumptions, performance considerations, and how to run the code.

Summary of Chapter 7

Interviews probe both technical commands and judgment. Pause to clarify constraints, discuss trade-offs, and demonstrate the reasoning behind chosen solutions. When you show why a method is appropriate for the data size, latency, and downstream use, you shift from coder to architect.

© Fourth Industrial Revolution blog provides additional reading for conceptual grounding in interdisciplinary approaches to AI and ML.

Copyright © 2026 All Rights Reserved.

Appendix I: AI and ML free videos

Free Artificial Intelligence AI Project in Python - Generative AI Chatbot (7 min.) https://youtu.be/nVTu6IjUbNY

LLM Concepts Deep Dive: Tokens, Transformers, Embeddings & RAG Explained (8 min.) https://youtu.be/xfD1nkM3BJI

Generative AI: Builder's Journey - Build LLMs, RAG, Prompting and Deployment in Gen AI (7 min.) https://youtu.be/R1qcoHwb5ss

Prompt Engineering & Prompting Techniques: Ways to Unlock Your AI's True Potential (7 min.) https://youtu.be/LxQ2ouKK-k4

New AI App: CodeCoach Multimodal Code Review & Explanation, Interview Q&A, Resume, LinkedIn, GitHub (2 min.) https://youtu.be/GTaZcqM0Ckg

LLMs in Python Explained: How to Build, Run & Deploy LLM Apps — LoRA, FastAPI, Exception Handling (7 min.) https://youtu.be/lHHYWqqxsR0

RAG Explained: Retrieval-Augmented Generation for LLMs—Reduce Hallucinations & Build Reliable Apps (7 min.) https://youtu.be/4hra0k5GMH8

Fine-Tuning LLMs Explained: Prompting vs RAG vs Fine Tuning | Cost, PEFT, RLHF (8 min.) https://youtu.be/EVxeU1Dw7jg

NumPy Explained in 8 Minutes | ndarrays, Vectorization, Broadcasting & Memory (8 min.) https://youtu.be/miOWA8VHkt0

Pandas Tutorial for Beginners | DataFrames, Cleaning, GroupBy & Interviews (6 min.) https://youtu.be/ZzCu5YddpEU

Matplotlib Tutorial for Beginners | Plots, OO API, Backends & Crashes Explained (8 min.) https://youtu.be/hdojjaMNoFg

Copyright © 2026 All Rights Reserved.

Connect on LinkedIn and message Inder P Singh for the below AI and ML guide documents:

Large Language Models LLMs Concepts
Generative AI With Large Language Models
Prompt Engineering
Introduction To LLMs In Python
Retrieval Augmented Generation RAG Framework In LLMs
Fine Tuning Large Language Models
LangChain With Python
Vector Databases
Numpy

Software and Testing Training – For personal reference

Copyright © 2026 All Rights Reserved.

Appendix II: SERVICES and CONTACT

This document is for personal reference only — redistribution, resale, or commercial use requires permission from the author.

Further reading & reproducible examples: Explore the free Pandas Playbook (code + datasets) on GitHub. For consulting, POCs, or bespoke training, contact Inder P Singh

3-Week Pandas & ML Interview Prep Workshop: Live, hands-on, career-focused. Pandas Interview Accelerator (Live + Labs, 3 weeks) — ₹9,000 INR / $108 USD. Message "PANDAS" to reserve.

How can I help?

1. TEMPLATES: Production-grade AI/ML code templates with playbooks
2. POC: AI working POC, dataset or AI demo for your portfolio
3. UPSKILLING: Deep-dive AI/ML/DL training courses (hands-on) starting regularly
4. PROJECT SUPPORT: AI architecture, system design and coding comprehensive guidance
5. RESUME SERVICES: I'm frequently approached by AI/ML hiring recruiters for suitable candidates.

Contact Info

© Kaggle (16 Badges & 16 Certifications) https://www.kaggle.com/inderpsingh

LinkedIn Profile **https://www.linkedin.com/in/inderpsingh/**

YouTube Channel **https://youtube.com/c/SoftwareandTestingTraining**

© Inder P Singh

© Fourth Industrial Revolution

Copyright © 2026 All Rights Reserved.