Building a \*\*data testing framework\*\* in Python involves automating the validation, transformation, and manipulation of data. This kind of framework is essential for testing data pipelines, ensuring data integrity, performing regression tests, and validating business logic for datasets.

To create a data testing framework from scratch, you need to focus on several components, such as data validation, transformation, and comparison, as well as integration with databases, files (CSV, Excel), and data pipelines.

Here’s how to build a \*\*data testing framework\*\* in Python from scratch:

---

### 1. \*\*Core Components of a Data Testing Framework\*\*

- \*\*Data Source Integration\*\*: Integration with different data sources (databases, APIs, CSV, Excel).

- \*\*Data Transformation & Validation\*\*: Validation of data quality, transformation logic, or schema validation.

- \*\*Test Case Structure\*\*: Writing test cases for data consistency, missing data, outliers, or range validation.

- \*\*Reporting & Logging\*\*: Capturing test results, logging test steps, and generating reports.

- \*\*Data Comparison\*\*: Comparing expected data with actual data (data equality, sorting, etc.).

- \*\*Scalability\*\*: The framework should support large datasets and can integrate with tools like \*\*pandas\*\* and \*\*SQLAlchemy\*\* for handling larger data.

### 2. \*\*Required Libraries\*\*

- \*\*pandas\*\*: To handle and manipulate structured data (CSV, Excel, JSON, etc.).

- \*\*pytest\*\*: To write and run your test cases.

- \*\*SQLAlchemy\*\*: For database interaction.

- \*\*pyodbc / psycopg2 / mysql-connector\*\*: For connecting to different databases.

- \*\*openpyxl / xlrd\*\*: For reading/writing Excel files.

- \*\*faker\*\*: For generating fake test data (if needed).

- \*\*pytest-html\*\*: To generate HTML reports of your test results.

### 3. \*\*Directory Structure\*\*

```plaintext

data-testing-framework/

│

├── data/

│ ├── input\_data.csv # Sample input data for testing

│ ├── expected\_data.csv # Expected data for validation

│

├── tests/

│ ├── test\_data\_integrity.py # Test case for data integrity

│ ├── test\_data\_transformation.py # Test case for data transformation

│ └── test\_schema\_validation.py # Test case for schema validation

│

├── utils/

│ ├── data\_utils.py # Utility functions for data validation

│ └── database\_utils.py # Utility functions for database interaction

│

├── reports/ # Directory to store test reports

│

├── requirements.txt # List of dependencies

└── conftest.py # Pytest configurations

```

### 4. \*\*Install Required Libraries\*\*

Create a `requirements.txt` file for the dependencies:

```plaintext

pandas

pytest

sqlalchemy

psycopg2 # If using PostgreSQL

mysql-connector-python # If using MySQL

openpyxl # If working with Excel files

pytest-html # For generating HTML reports

```

Install the dependencies via `pip`:

```bash

pip install -r requirements.txt

```

### 5. \*\*Utility Functions (e.g., `data\_utils.py`)\*\*

In this file, you can define various utility functions to handle data validation, comparison, and transformation.

```python

# utils/data\_utils.py

import pandas as pd

import numpy as np

def load\_csv(file\_path):

""" Load a CSV file into a pandas DataFrame """

return pd.read\_csv(file\_path)

def validate\_data\_integrity(df, required\_columns):

""" Ensure the DataFrame has the required columns """

missing\_columns = [col for col in required\_columns if col not in df.columns]

assert not missing\_columns, f"Missing required columns: {missing\_columns}"

def validate\_no\_nulls(df):

""" Ensure there are no null values in the DataFrame """

assert df.isnull().sum().sum() == 0, "Null values found in the data"

def validate\_data\_range(df, column, min\_val, max\_val):

""" Validate that values in a column are within a specified range """

assert df[column].min() >= min\_val, f"Values in {column} are below the minimum range"

assert df[column].max() <= max\_val, f"Values in {column} exceed the maximum range"

def compare\_dataframes(df1, df2):

""" Compare two DataFrames for equality """

pd.testing.assert\_frame\_equal(df1, df2, check\_dtype=True, check\_index=True, check\_column\_type=True)

def transform\_data(df):

""" Example transformation function (e.g., date format, string manipulation) """

# Example: Convert a 'date' column to a datetime format

df['date'] = pd.to\_datetime(df['date'])

return df

```

### 6. \*\*Test Cases (e.g., `test\_data\_integrity.py`)\*\*

Here’s how you might structure your test cases using \*\*pytest\*\* to ensure data integrity, schema validation, and transformation.

```python

# tests/test\_data\_integrity.py

import pytest

from utils.data\_utils import load\_csv, validate\_data\_integrity, validate\_no\_nulls, validate\_data\_range

import pandas as pd

@pytest.fixture

def sample\_data():

"""Fixture to load the sample data for testing"""

return load\_csv("data/input\_data.csv")

def test\_column\_integrity(sample\_data):

""" Test to validate that required columns are present in the data """

required\_columns = ['id', 'name', 'age', 'date']

validate\_data\_integrity(sample\_data, required\_columns)

def test\_no\_nulls(sample\_data):

""" Test to check if there are any null values in the data """

validate\_no\_nulls(sample\_data)

def test\_age\_range(sample\_data):

""" Test to validate that the 'age' column values are within the range of 18-99 """

validate\_data\_range(sample\_data, 'age', 18, 99)

def test\_data\_transformation(sample\_data):

""" Test to validate data transformation logic (e.g., date formatting) """

transformed\_data = transform\_data(sample\_data)

assert pd.api.types.is\_datetime64\_any\_dtype(transformed\_data['date']), "Date column transformation failed"

```

### 7. \*\*Data Transformation Test (e.g., `test\_data\_transformation.py`)\*\*

For testing any specific transformations, you can have a separate test file that validates data transformation logic.

```python

# tests/test\_data\_transformation.py

from utils.data\_utils import transform\_data

import pytest

def test\_date\_transformation(sample\_data):

""" Test to ensure that the date column is properly transformed """

transformed\_data = transform\_data(sample\_data)

assert transformed\_data['date'].dtype == 'datetime64[ns]', "Date column transformation failed"

```

### 8. \*\*Schema Validation (e.g., `test\_schema\_validation.py`)\*\*

For schema validation, you could compare the actual schema of your dataset (data types, column names) against the expected schema.

```python

# tests/test\_schema\_validation.py

import pytest

import pandas as pd

from utils.data\_utils import load\_csv

@pytest.fixture

def expected\_schema():

"""Expected schema for validation (column names and types)"""

return {

'id': 'int64',

'name': 'object',

'age': 'int64',

'date': 'datetime64[ns]'

}

def test\_schema\_validation(sample\_data, expected\_schema):

""" Validate the schema of the data against the expected schema """

actual\_schema = sample\_data.dtypes.to\_dict()

for column, dtype in expected\_schema.items():

assert column in actual\_schema, f"Column {column} not found"

assert actual\_schema[column] == dtype, f"Column {column} has wrong dtype. Expected {dtype}, got {actual\_schema[column]}"

```

### 9. \*\*Database Interaction (`database\_utils.py`)\*\*

For testing data in databases, you can use \*\*SQLAlchemy\*\* to create a connection and run queries to fetch and validate data.

```python

# utils/database\_utils.py

from sqlalchemy import create\_engine

import pandas as pd

def connect\_to\_db(db\_url):

""" Connect to the database using SQLAlchemy """

engine = create\_engine(db\_url)

return engine.connect()

def fetch\_data\_from\_db(query, connection):

""" Execute a query and return the result as a pandas DataFrame """

return pd.read\_sql(query, connection)

```

### 10. \*\*Running the Tests\*\*

Once everything is set up, you can run the tests using \*\*pytest\*\*:

```bash

pytest --maxfail=3 --disable-warnings -v

```

You can also generate an HTML report:

```bash

pytest --maxfail=3 --disable-warnings -v --html=reports/test\_report.html

```

### 11. \*\*Example Data (input\_data.csv)\*\*

Here is an example of what the input data might look like:

```csv

id,name,age,date

1,John,28,2023-01-01

2,Alice,34,2023-01-02

3,Bob,45,2023-01-03

4,Charlie,23,2023-01-04

```

### Conclusion

This basic framework can be extended and customized based on your specific use case, such as:

- Testing large datasets with \*\*dask\*\* (for distributed computing).

- Adding support for more data sources like \*\*Parquet\*\*, \*\*JSON\*\*, or \*\*APIs\*\*.

- Creating more complex transformations, validations, or handling multiple schemas.

By using libraries like \*\*pandas\*\*, \*\*pytest\*\*, and \*\*SQLAlchemy\*\*, you can build a robust \*\*data testing framework\*\* in Python that ensures data quality, integrity, and correctness across different environments.