10.29 : ETL and ELT in Python

Introduction to ETL and ELT Pipelines

Introduction to ETL and ELT Pipelines 00:00 - 00:24

Welcome! I'm Jake, your instructor for this course on ETL and ELT pipelines. As a Data Engineer, I work with tools like Python, SQL, and Airflow to build various data pipelines, and I'm here to help you do the same.

BI ML AI 00:24 - 00:54

You've likely heard buzzwords like business intelligence, machine learning, and AI. For many companies, these initiatives are central to driving innovation and growth. But before diving into them, we need data—organized, timely, and in the right format. Data pipelines help us achieve that!

Data pipelines 00:54 - 01:38

Data pipelines are designed to move data from a source to a destination, transforming it along the way. Sources can include CSV files, APIs, or databases. Once data is extracted from a source, the goal is to transform and load it into a destination, ready for business intelligence, machine learning, or AI applications. In this course, we'll explore ETL and ELT pipelines.

ETL and ELT Pipelines 01:38 - 02:39

ETL stands for *Extract, Transform, Load.* ETL pipelines first extract data, transform it, and then load it into a destination. This traditional data pipeline pattern often uses Python and libraries like pandas for data manipulation. ELT, on the other hand, stands for *Extract, Load, Transform,* and is commonly used with data warehouses, where the data is transformed after loading.

Extract, Transform, Load (ETL) 02:39 - 03:11

Extract, transform, load (ETL)

```
def load(data_frame, target_table):
    # Some custom-built Python logic to load data to SQL
    data_frame.to_sql(name=target_table, con=POSTGRES_CONNECTION)
    print(f"Loading data to the {target_table} table")

# Now, run the data pipeline
extracted_data = extract(file_name="raw_data.csv")
transformed_data = transform(data_frame=extracted_data)
load(data_frame=transformed_data, target_table="cleaned_data")
```

```
Extracting data from raw_data.csv
Transforming data to remove 'null' records
Loading data to the cleaned_data table
```

Here's an example of an ETL pipeline in Python. We use custom functions to extract, transform, and load data. The load function writes a pandas DataFrame to a SQL database, and the ETL functions are called in sequence to execute the pipeline.

Extract, Load, Transform (ELT) 03:11 - 03:38

Extract, load, transform (ELT)

```
...
def transform(source_table, target_table):
    data_warehouse.run_sql("""
        CREATE TABLE {target_table} AS
          SELECT
              <field-name>, <field-name>, ...
          FROM {source_table};
    """)

# Similar to ETL pipelines, call the extract, load, and transform functions
extracted_data = extract(file_name="raw_data.csv")
load(data_frame=extracted_data, table_name="raw_data")
transform(source_table="raw_data", target_table="cleaned_data")
```

For ELT, the extract, load, and transform functions are reordered. In this example, the transform function creates a table based on a *SELECT* statement. We call each function in sequence to complete the ELT pipeline.

We'll also take a look at... 03:38 - 04:01

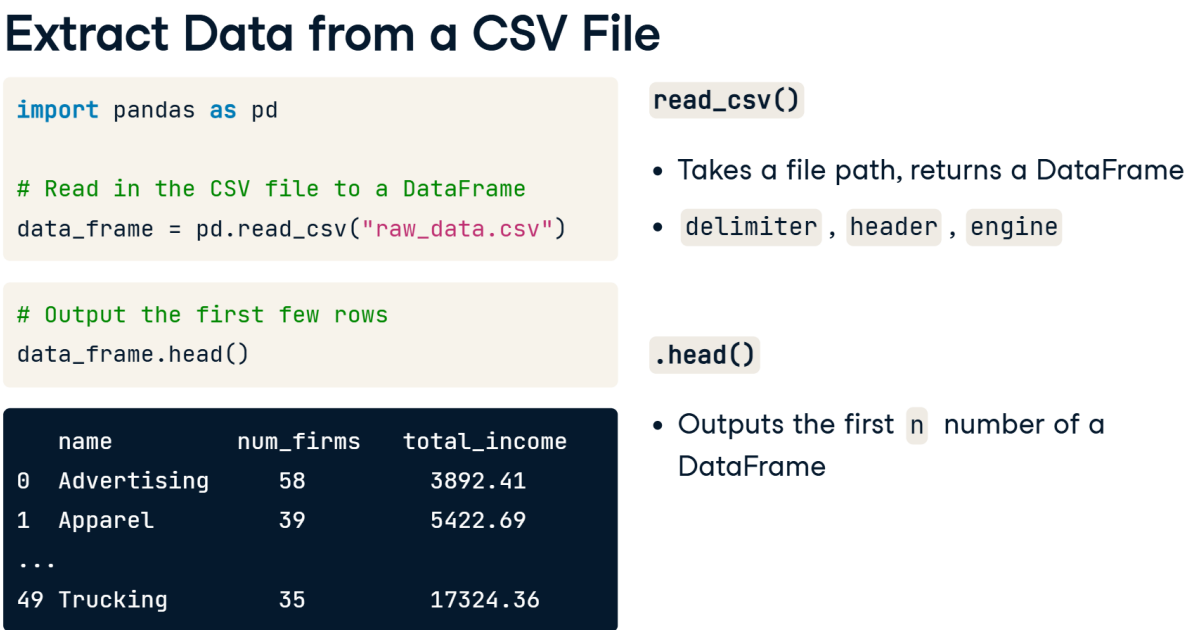
We'll explore working with both tabular and non-tabular data, using pandas for data transformations and to write data to disk and SQL databases. Stick around for the end, where we'll cover unit-testing, monitoring, and deploying data pipelines to production.

Building ETL and ELT Pipelines

Building ETL and ELT Pipelines 00:00 - 00:13

Welcome back! Earlier, we briefly discussed running ETL and ELT pipelines. Now, let's dive into the details of what this process entails.

Extract Data from a CSV File 00:13 - 01:01



First, we'll extract data from a CSV file using the pandas library. After importing pandas as `pd`, we call the `read_csv` function, passing in a file path to load the data into a DataFrame. Additional parameters, like `delimiter`, `header`, and `engine`, can customize how the file is read—especially helpful for non-standard file formats. To view the initial rows, we can use the `head` method, which shows the first five rows by default.

Filtering a DataFrame 01:01 - 01:55

Filtering a DataFrame

```
   name          num_firms   total_income
0  Advertising      58          3892.41
1  Apparel          39          5422.69
...
49 Trucking         35          17324.36
```

```
        name          num_firms
1   Apparel          39
37  Apparel          61
```

```
# First, by rows
data_frame.loc[data_frame["name"] == "Apparel", :]

# Then, by columns
data_frame.loc[:, ["name", "num_firms"]]
```

`.loc`

- Filters a DataFrame
- `:` means "all"

Once the data is in a DataFrame, we can filter it. For example, we might start with an unfiltered DataFrame and aim to display only rows with the "name" "Apparel" and columns "name" and "num_firms." To achieve this, we use `loc` on the DataFrame. Here, we first filter rows by specifying `"Apparel"` for the "name" column, and we use `:` after the comma to include all columns. Lastly, we narrow down to "name" and "num_firms."

Write a DataFrame to a CSV File 01:55 - 02:33

Write a DataFrame to a CSV File

```
# Write a DataFrame to a .csv file
data_frame.to_csv("cleaned_data.csv")
```

`.to_csv()`

- Takes a `path`, creates DataFrame from file stored at that `path`
- Can take other parameters to customize the output

Other options, like:

`.to_json()`, `.to_excel()`, `.to_sql()`

To save data to a file, we use the `to_csv` method on a DataFrame and pass in a file path. This writes the DataFrame to the specified path, with several optional parameters to control how the data is stored. Along with `to_csv`, pandas offers methods like `to_json`, `to_excel`, and `to_sql` for various formats.

Running SQL Queries 02:33 - 03:02

Running SQL Queries

```
data_warehouse.execute(  # Use Python clients or other tools to run SQL queries
    """
    CREATE TABLE total_sales AS
        SELECT
            ds,
            SUM(sales)
        FROM raw_sales_data
        GROUP BY ds;
    """
)
```

- Tools like `.execute()` to run SQL queries

For SQL-based data transformations in a data warehouse, we can use multi-line strings to write SQL queries. Python clients like SQLAlchemy or the Snowflake Python Connector can execute these queries. By using the `execute` method, we can run a query, like creating a new table.

Putting it all together! 03:02 - 03:47

Putting it all together!

```
# Define extract(), transform(), and load() functions
...

def transform(data_frame, value):
    return data_frame.loc[data_frame["name"] == value, ["name", "num_firms"]]

# First, extract data from a .csv
extracted_data = extract(file_name="raw_data.csv")

# Then, transform the `extracted_data`
transformed_data = transform(data_frame=extracted_data, value="Apparel")

# Finally, load the `transformed_data`
load(data_frame=transformed_data, file_name="cleaned_data.csv")
```

Here's the complete workflow: add ETL logic into `extract`, `transform`, and `load` functions. First, call the `extract` function to load data from a file, then the `transform` function to filter rows with "Apparel", and finally, the `load` function to save the transformed DataFrame to a file called "cleaned_data.csv". This general structure will guide most of the ETL and ELT pipelines throughout the course!

Extracting data from structure sources

Extracting data from structured sources 00:00 - 00:08

In this video, we'll go through the first step of any data pipeline—extracting data from a source system.

Source systems 00:08 - 00:55

Data pipelines often start by extracting data from source systems, which can be structured files like CSV, parquet, or JSON files, dynamic data stores like SQL databases, and APIs from third-party providers. Other organizational sources include data lakes and warehouses, and advanced pipelines may even utilize web scraping. In this video, we'll focus on extracting data from tabular sources like parquet files and SQL databases.

Reading in parquet files 00:55 - 01:42

Reading in parquet files

Parquet files:

- Open source, column-oriented file format designed for efficient field storage and retrieval
- Similar to working with CSV files

```
import pandas as pd

# Read the parquet file into memory
raw_stock_data = pd.read_parquet("raw_stock_data.parquet", engine="fastparquet")
```

You may already be familiar with reading CSV files in pandas, but other formats, such as Parquet, are common. Apache Parquet is a column-oriented file format optimized for storage and retrieval efficiency. To load a parquet file, import pandas as `pd`, and use `pd.read_parquet()` with the file path, optionally specifying an engine like "fastparquet." This process is faster than reading CSV files and just as straightforward!

Connecting to SQL databases 01:42 - 02:56

Connecting to SQL databases

- Data can be pulled from SQL databases into a `pandas` DataFrame
- Requires a connection URI to build an engine, and connect to the database

```
import sqlalchemy
import pandas as pd

# Connection URI: schema_identifier://username:password@host:port/db
connection_uri = "postgresql+psycopg2://repl:password@localhost:5432/market"
db_engine = sqlalchemy.create_engine(connection_uri)
```

```
# Query the SQL database
raw_stock_data = pd.read_sql("SELECT * FROM raw_stock_data LIMIT 10", db_engine)
```

SQL databases are another frequently used tabular data source. To retrieve data from an SQL database into a pandas DataFrame, use the `pd.read_sql` function. First, establish a connection using `sqlalchemy`'s `create_engine` function, which takes a URI formatted with the schema identifier (like `postgresql+psycopg2`), username, password, host, port, and database name. Pass this connection along with a query to `pd.read_sql()` to load data into a DataFrame for use in your pipeline.

Modularity 02:56 - 03:55

Modularity

Separating logic into functions

- Increases readability within a pipeline
- Adheres to the principle "don't repeat yourself"

```
def extract_from_sql(connection_uri, query):
    # Create an engine, query data and return DataFrame
    db_engine = sqlalchemy.create_engine(connection_uri)
    return pd.read_sql(query, db_engine)

extract_from_sql("postgresql+psycopg2://.../market", "SELECT ... LIMIT 10;")
```

We've looked at `read_parquet` and `read_sql` for extracting tabular data. Modular code is essential for data pipeline development—it improves readability and reusability. Separate the "extract", "transform", and "load" logic into distinct functions to avoid redundancy, following the PEP-8 principle of "don't repeat yourself." Modular code streamlines development, saves time, and makes troubleshooting more efficient.

Transforming data with pandas

Transforming data with pandas 00:00 - 00:05

Now that you've extracted data from a source system, it's time to transform it.

Transforming data in a pipeline 00:05 - 00:48

Transforming data effectively is key to a successful data pipeline. Inaccurate transformations can lead to downstream issues. With pandas, transformation is straightforward and powerful. Rows can be filtered, columns created, and data types changed—all often in a single line of code. Here, we'll explore `.loc` and `to_datetime` for filtering and date transformations.

Filtering records with .loc[] 00:48 - 01:59

Filtering records with .loc[]

`.loc[]` allows for both dimensions of a DataFrame to be transformed

```
# Keep only non-zero entries
cleaned = raw_stock_data.loc[raw_stock_data["open"] > 0, :]
```

```
# Remove excess columns
cleaned = raw_stock_data.loc[:, ["timestamps", "open", "close"]]
```

```
# Combine into one step
cleaned = raw_stock_data.loc[raw_stock_data["open"] > 0, ["timestamps", "open", "close"]]
```

`.iloc[]` uses integer indexing to filter DataFrames

```
cleaned = raw_stock_data.iloc[[0:50], [0, 1, 2]]
```

The `.loc[]` function is used to filter DataFrames by row and column values. For example, calling `.loc` on `raw_stock_data` with `open > 0` keeps only rows where the "open" value is greater than zero, while the colon after the comma includes all columns. Another example filters columns by keeping only “timestamps”, “open”, and “close”. These can be combined into one statement to filter both rows and columns. `.iloc[]` is also available, using integer indexing to select data by row and column positions, like the first 50 rows and three columns in a DataFrame.

Altering data types 01:59 - 02:59

Altering data types

Data types often need to be converted for downstream use cases

- `.to_datetime()`

```
# "timestamps" column currectly looks like: "20230101085731"
# Convert "timestamps" column to type datetime
cleaned["timestamps"] = pd.to_datetime(cleaned["timestamps"], format="%Y%m%d%H%M%S")
```

```
Timestamp('2023-01-01 08:57:31')
```

```
# "timestamps" column currently looks like: 1681596000011
# Convert "timestamps" column to type datatime
cleaned["timestamps"] = pd.to_datetime(cleaned["timestamps"], unit="ms")
```

```
Timestamp('2023-04-15 22:00:00.011000')
```

Converting data types is common in pandas transformations, especially for timestamps. For instance, using `pd.to_datetime`, a "timestamps" column in string format (like "YYYYmmddHHMMSS") can be converted to `datetime`. It can also handle Unix timestamps (milliseconds since 1970), making it versatile for time-based data transformations. Pandas offers other tools for data type transformations beyond `to_datetime`.

Validating transformations 02:59 - 03:52

Validating transformations

Transforming data comes with risks:

- Losing information
- Creating faulty data

```
# Several ways to investigate a DataFrame
cleaned = raw_stock_data.loc[raw_stock_data["open"] > 0, ["timestamps", "open", "close"]]
print(cleaned.head())
```

```
# Return smallest and largest records
print(cleaned.nsmallest(10, ["timestamps"]))
print(cleaned.nlargest(10, ["timestamps"]))
```

Validation is essential to ensure accuracy and prevent data loss during transformation. The `head()` method is helpful to check the first few rows of the DataFrame (default five, but customizable), providing a quick "spot-check." The `nsmallest` and `nlargest` methods further allow inspection of the smallest or largest values, which is particularly useful when filtering by ranges. These tools make it easy to validate transformations visually and detect potential issues early.

Persisting data with pandas

Persisting data with pandas 00:00 - 00:11

Nice transformation! We've covered extracting and transforming data. Now, let's move on to loading data using pandas.

Persisting data in an ETL pipeline 00:11 - 00:48

Data persistence is a best practice that allows data to be accessible to consumers at various pipeline stages. While often done in the "load" phase of ETL, persisting data through files or checkpoints throughout the pipeline is beneficial, especially for recovery after failures or when data is challenging to reacquire.

Loading data to CSV files using pandas 00:48 - 01:22

Loading data to CSV files using pandas

`.to_csv()` method

```
import pandas as pd

# Data extraction and transformation
raw_data = pd.read_csv("raw_stock_data.csv")
stock_data = raw_data.loc[raw_data["open"] > 100, ["timestamps", "open"]]

# Load data to a .csv file
stock_data.to_csv("stock_data.csv")
```

- `.to_csv` called on the DataFrame
- Writes DataFrame to path `"stock_data.csv"`

To write a DataFrame to a file, we use pandas' `to_csv` method. In this example, stock market data is transformed and saved using `to_csv`, specifying a path like `stock_data.csv`. Additional arguments let you customize the storage format.

Customizing CSV file output 01:22 - 02:38

Customizing CSV file output

```
stock_data.to_csv("./stock_data.csv", header=True)
```

- Takes `True`, `False` or list of string values

```
stock_data.to_csv("./stock_data.csv", index=True)
```

- Takes `True` or `False`
- Determines whether `index` column is written to the file

```
stock_data.to_csv("./stock_data.csv", sep="|")
```

- Takes string value used to separate columns in the file
- The `|` character is a common option

Has counterparts:

- `.to_parquet()`
- `.to_json()`
- `.to_sql()`

https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_csv.html

The `header` argument allows you to include a header (default is `True`), omit it (`False`), or customize it with a list of column names. The `index` argument includes the DataFrame index in the file (default `True`), helpful if it has meaningful data, like transaction IDs. Otherwise, set `index=False`. The `sep` argument defines the column separator, with a comma as default, though alternatives like the pipe character are also common. Other persistence methods include `to_parquet`, `to_json`, and `to_sql` for database storage, among others.

Ensuring data persistence 02:38 - 03:22

Ensuring data persistence

Was the DataFrame correctly stored to the CSV file?

```
import pandas
import os  # Import the os module

# Extract, transform and load data
raw_data = pd.read_csv("raw_stock_data.csv")
stock_data = raw_data.loc[raw_data["open"] > 100, ["timestamps", "open"]]
stock_data.to_csv("stock_data.csv")

# Check that the path exists
file_exists = os.path.exists("stock_data.csv")
print(file_exists)
```

```
True
```

To verify successful persistence, the `os.path.exists` function checks if a file exists at a specified path. By storing this check result in a variable, we can confirm the data's existence (`True`) or detect potential errors if `False`, a valuable validation tool in the "load" phase of the pipeline.

Monitoring a data pipeline

Monitoring a data pipeline 00:00 - 00:10

Excellent progress! Now, let's explore methods to monitor pipeline performance and set up alerts for failures.

Monitoring a data pipeline 00:10 - 00:40

Once a pipeline is set up, monitoring becomes essential to handle unexpected changes or execution issues. Pipelines are susceptible to source system errors, data type changes, or tool updates that affect functionality. Effective monitoring and alerting give Data Engineers the visibility needed to resolve issues promptly, ideally before data consumers are impacted.

Logging data pipeline performance 00:40 - 01:53

Logging data pipeline performance

- Document performance at execution
- Provides a starting point when a solution fails

```
import logging
logging.basicConfig(format='%(levelname)s: %(message)s', level=logging.DEBUG)

# Create different types of logs
logging.debug(f"Variable has value {path}")
logging.info("Data has been transformed and will now be loaded.")
```

```
DEBUG: Variable has value raw_file.csv
INFO: Data has been transformed and will now be loaded.
```

Logs are a foundational tool for monitoring, providing insight into pipeline execution. Configurable with Python's logging module, logs capture different levels of severity, like debug, info, warning, and error. *Debug logs* assist in understanding data dimensions, types, and variables during development. *Info logs* mark key checkpoints, helping trace actions taken on data through the pipeline.

Logging warnings and errors 01:53 - 02:32

Warning logs alert engineers to unexpected occurrences, such as unexpected row counts or new data types, without halting execution. *Error logs*, however, document critical failures that require pipeline termination, such as missing or improperly formatted data. Effective logging makes troubleshooting more efficient by highlighting the root causes of pipeline issues.

Handling exceptions with try-except 02:32 - 02:59

Handling exceptions with try-except

```
try:
    # Execute some code here
    ...

except:
    # Logging about failures that occured
    # Logic to execute upon exception
    ...
```

- Provides a way to execute code if errors occur

Data pipelines should handle errors gracefully. Using Python's `try-except` structure, code runs within the `try` block, while the `except` block handles exceptions, allowing the pipeline to log issues without abruptly stopping.

Handling specific exceptions with try-except 02:59 - 03:55

Handling specific exceptions with try-except

Pass the specific exception in the `except` clause

```
try:
    # Try to filter by price_change
    clean_stock_data = transform(raw_stock_data)
    logging.info("Successfully filtered DataFrame by 'price_change'")

except KeyError as ke:
    # Handle the error, create new column, transform
    logging.warning(f"{ke}: Cannot filter DataFrame by 'price_change'")
    raw_stock_data["price_change"] = raw_stock_data["close"] - raw_stock_data["open"]
    clean_stock_data = transform(raw_stock_data)
```

For known errors, specific exceptions like `KeyError` can be targeted within `try-except` blocks. For example, the transform function attempts to filter rows based

on the `price_change` column. If a `KeyError` occurs, it is logged, and the column is created if needed. Leveraging `try-except` with logging offers a straightforward monitoring solution in Python, laying the groundwork for more sophisticated tracking in data workflows.

Extracting non-tabular data

Extracting non-tabular data 00:00 - 00:09

We've been working with tabular data sources so far. In this lesson, we'll dive into pipelines for non-tabular data.

Extracting non-tabular data 00:09 - 00:24

Using an architecture diagram for guidance, we'll explore tools to extract non-tabular data before transforming and loading it for downstream use.

Types of non-tabular data 00:24 - 00:58

Non-tabular, unstructured data makes up over 80% of generated data, including sources like text, audio, image, video, spatial, and IoT data. With non-tabular data, engineers frequently invest significant time extracting features and transforming raw data into a structured, tabular format.

Working with APIs and JSON data 00:58 - 02:04

APIs (Application Programming Interfaces) enable data retrieval and submission without direct database access, commonly delivering data in JSON format. APIs prevent direct interaction with databases, thus maintaining data integrity. JSON data, a non-tabular format storing information as key-value pairs, resembles Python dictionaries and is schema-less, making it a versatile choice.

Reading JSON files with pandas 02:04 - 02:57

Reading JSON files with pandas

```
{
    "timestamps": [863703000, 863789400, ...],
    "open": [0.121875, 0.098438, ...],
    "close": [...],
    "volume": [...]
}
```

Use the `.read_json()` function

```
# Read in a JSON file in the format above
raw_stock_data = pd.read_json("raw_stock_data.json", orient="columns")
```

https://pandas.pydata.org/docs/reference/api/pandas.read_json.html

To read JSON data stored in a file, we use `pd.read_json`, which transforms JSON data into a DataFrame. The `orient` parameter in `read_json` specifies how data is organized. For data stored in columns, `orient="columns"` is appropriate, while `orient="records"` fits a list of dictionaries, and `orient="index"` is for indices as keys with corresponding dictionary records.

Nested or unstructured JSON data 02:57 - 03:19

Some JSON files contain nested or complex structures, making them unsuitable for direct DataFrame conversion. In these cases, loading data into a dictionary first allows us to restructure it before converting it into a DataFrame.

Reading JSON files with json 03:19 - 03:58

The `json` library's `load` function reads JSON data into a Python dictionary. This dictionary format mirrors the JSON structure, making it easier to manipulate and transform it into a DataFrame-ready format. This approach is especially helpful when working with nested or unstructured JSON data.

Transforming non-tabular data

Transforming non-tabular data 00:08 - 00:24

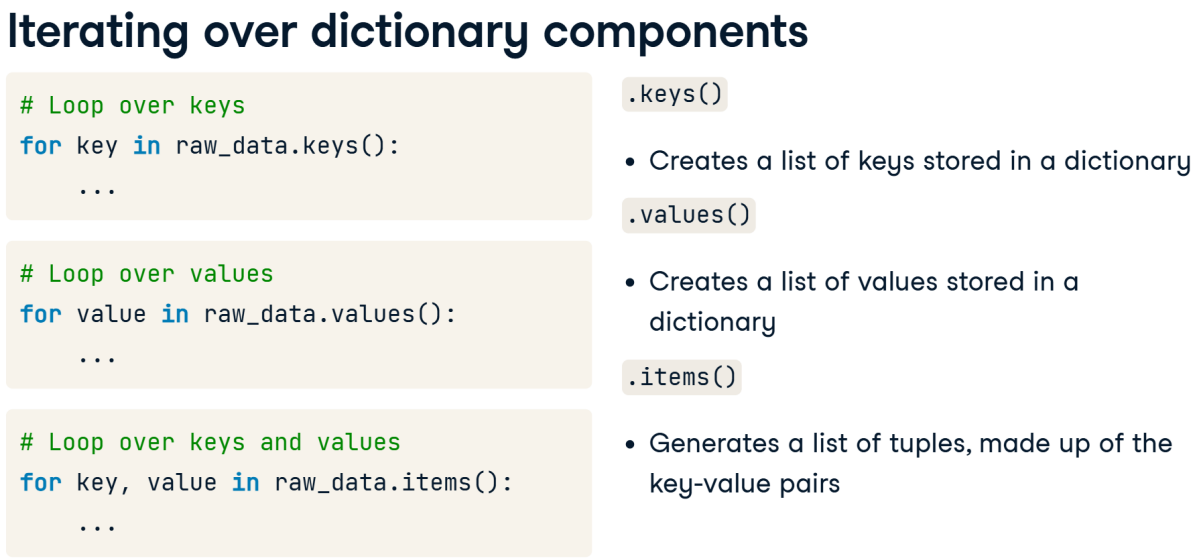
In the last lesson, we used `json.load` to store complex JSON data as a dictionary. Now, we'll explore tools to transform this data into a DataFrame-

friendly format.

Storing data in dictionaries 00:24 - 00:46

In the dictionary, each key represents an index, with associated values stored as nested dictionaries. To make this DataFrame-ready, we'll need to reformat it, such as by converting it into a list of lists.

Iterating over dictionary components 00:46 - 01:25



To iterate through dictionary keys and values, we can use the `keys`, `values`, and `items` methods. `keys` and `values` create lists of the dictionary's keys or values, respectively, while `items` returns a list of tuples, each containing a key-value pair. This can be helpful for accessing both key and value in a loop.

Parsing data from dictionaries 01:25 - 02:27

Parsing data from dictionaries

```
entry = {
    "volume": 1443120000,
    "price": {
        "open": 0.12187,
        "close": 0.09791,
    }
}
```

```
# Parse data from dictionary using .get()
volume = entry.get("volume")
```

```
ticker = entry.get("ticker", "DCMP")
```

```
# Call .get() twice to return the nested "open" value
open_price = entry.get("price").get("open", 0)
```

For individual value extraction, the `get` method is very useful. `get` takes a key and returns its value if it exists, or `None` if it doesn't. This method allows us to set a default return if a key is missing, ideal for handling non-standard JSON formats. For nested dictionaries, calling `get` twice allows deeper access—for instance, `price` and `open` in sequence to retrieve nested values.

Creating a DataFrame from a list of lists 02:27 - 02:51

Creating a DataFrame from a list of lists

Pass a list of lists to `pd.DataFrame()`

```
# Pass a list of lists to pd.DataFrame
raw_data = pd.DataFrame(flattened_rows)
```

Set column headers using `.columns`

```
# Create columns
raw_data.columns = ["timestamps", "open", "close", "volume"]
```

Set an index using `.set_index()`

```
# Set the index column to be "timestamps"
raw_data.set_index("timestamps")
```

After converting dictionary data into a list of lists, we can pass it to `pd.DataFrame` to create a DataFrame. Column names can be set by assigning a list of names to the `columns` attribute. To set an index, we use the `set_index` method.

Transforming stock data 02:51 - 03:33

Transforming stock data

```
parsed_stock_data = []

# Loop through each key-value pair of the raw_stock_data dictionary
for timestamp, ticker_info in raw_stock_data.items():
    parsed_stock_data.append([
        timestamp,
        ticker_info.get("price", {}).get("open", 0),  # Parse the opening price
        ticker_info.get("price", {}).get("close", 0),  # Parse the closing price
        ticker_info.get("volume", 0)  # Parse the volume
    ])
```

```
# Create a DataFrame, assign column names, and set an index
transformed_stock_data = pd.DataFrame(parsed_stock_data)
transformed_stock_data.columns = ["timestamps", "open", "close", "volume"]
transformed_stock_data = transformed_stock_data.set_index("timestamps")
```

To transform stock data, we iterate through `raw_stock_data` using `items`. In each loop, we parse `open`, `close`, and `volume` values from the nested dictionary using `get`, then append these as a list to `parsed_stock_data`. Finally, we convert `parsed_stock_data` to a DataFrame, assigning column names and setting an index.

Advanced data transformation with pandas

Advanced data transformation with pandas 00:00 - 00:11

Leveraging advanced pandas functionality can streamline data transformations in a pipeline. Let's revisit our architecture to see how these techniques fit in.

Advanced data transformation with pandas 00:11 - 00:27

In the past lessons, we transformed non-tabular data into a DataFrame. Now, we'll explore advanced techniques like handling missing values, grouping, and applying custom transformations.

Filling missing values with pandas 00:27 - 01:03

```
timestamps           volume      open     close
1997-05-15 13:30:00  1443120000  0.121875  0.097917
1997-05-16 13:30:00   294000000  NaN       0.086458
1997-05-19 13:30:00   122136000  0.088021  NaN
```

```
# Fill all NaN with value 0
clean_stock_data = raw_stock_data.fillna(value=0)
```

```
timestamps           volume      open     close
1997-05-15 13:30:00  1443120000  0.121875  0.097917
1997-05-16 13:30:00   294000000  0.000000  0.086458
1997-05-19 13:30:00   122136000  0.088021  0.000000
```

Encountering missing values is common, represented as NaN. The `fillna` method helps fill NaNs with a specified value. Here, missing "open" and "close" values are filled with zero.

Filling missing values with pandas 01:03 - 01:33

```
timestamps           volume      open      close
1997-05-15 13:30:00  1443120000  0.121875  0.097917
1997-05-16 13:30:00   294000000  NaN       0.086458
1997-05-19 13:30:00   122136000  0.088021  NaN
```

```
# Fill NaN values with specific value for each column
clean_stock_data = raw_stock_data.fillna(value={"open": 0, "close": .5}, axis=1)
```

```
timestamps           volume      open      close
1997-05-15 13:30:00  1443120000  0.121875  0.097917
1997-05-16 13:30:00   294000000  0.000000  0.086458
1997-05-19 13:30:00   122136000  0.088021  0.500000
```

To fill specific columns, we can pass a dictionary to `fillna`. Setting `axis=1`, the keys represent column names with values to fill missing entries. Here, missing "open" values are filled with zero, and "close" values with 0.5.

Filling missing values with pandas 01:33 - 01:58

```
timestamps           volume      open      close
1997-05-15 13:30:00  1443120000  0.121875  0.097917
1997-05-16 13:30:00   294000000  NaN       0.086458
1997-05-19 13:30:00   122136000  0.088021  NaN
```

```
# Fill NaN value using other columns
raw_stock_data["open"].fillna(raw_stock_data["close"], inplace=True)
```

```
timestamps           volume      open      close
1997-05-15 13:30:00  1443120000  0.121875  0.097917
1997-05-16 13:30:00   294000000  0.086458  0.086458
1997-05-19 13:30:00   122136000  0.088021  NaN
```

A column can also be used to fill NaNs in another column. For example, missing "open" values are filled with the corresponding "close" values. With `inplace=True`, the DataFrame is updated directly without needing to assign a new variable.

Grouping data 01:58 - 02:18

```sql
SELECT
    ticker,
    AVG(volume),
    AVG(open),
    AVG(close)
FROM raw_stock_data
GROUP BY ticker;
```

The `.groupby()` method can recreate the query above, using `pandas`

Similar to SQL's "GROUP BY," the `groupby` method in pandas groups data by specific columns. Here, grouping by "ticker" and taking the average of other columns mimics SQL's grouping capabilities.

Grouping data with pandas 02:18 - 02:54

```
ticker  volume        open        close
AAPL    1443120000    0.121875    0.097917
AAPL     297000000    0.098146    0.086458
AMZN     124186000    0.247511    0.251290
```

```python
# Use Python to group data by ticker, find the mean of the reamining columns
grouped_stock_data = raw_stock_data.groupby(by=["ticker"], axis=0).mean()
```

```
          volume        open        close
ticker
AAPL      1.149287e+08    34.998377  34.986851
AMZN      1.434213e+08    30.844692  30.830233
```

Can use `.min()`, `.max()` and `.sum()` to aggregate data

In a single line, `groupby` groups `raw_stock_data` by the "ticker" column and calculates the mean of other columns. Setting `axis=0` (the default) groups by rows. The result is stored in `grouped_stock_data`. Other aggregation methods like `min`, `max`, and `sum` are also available.

Applying advanced transformations to DataFrames 02:54 - 03:47

The `.apply()` method can handle more advanced transformations

```
def classify_change(row):
    change = row["close"] - row["open"]
    if change > 0:
        return "Increase"
    else:
        return "Decrease"
```

```
# Apply transformation to DataFrame
raw_stock_data["change"] = raw_stock_data.apply(
    classify_change,
    axis=1
)
```

Before transformation

ticker	...	open	close
AAPL		0.121875	0.097917
AAPL		0.098146	0.086458
AMZN		0.247511	0.251290

After transformation

ticker	...	open	close	change
AAPL		0.121875	0.097917	Decrease
AAPL		0.098146	0.086458	Decrease
AMZN		0.247511	0.251290	Increase

For complex logic, the `apply` method accepts custom functions. For example, a function `classify_change` is defined to categorize price movements by comparing "open" and "close" values, returning "Increase" or "Decrease." Applying this function with `axis=1` processes each row individually, writing the output to a new "change" column.

Loading data to a SQL database with pandas

Loading data to a SQL database with pandas 00:00 - 00:07

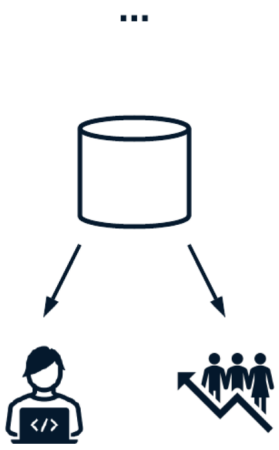
Before exploring SQL database loading, let's take a final look at our architecture diagram.

Load data to a SQL database with pandas 00:07 - 00:21

Having extracted and transformed non-tabular data, we're ready to address the final ETL component: loading data to a SQL database.

Loading data into a SQL database with pandas 00:21 - 00:58

Loading data into a SQL database with pandas



`pandas` provides `.to_sql()` to persist data to SQL

- `name`
- `con`
- `if_exists`
- `index`
- `index_label`

SQL databases are a common endpoint for transformed data in a pipeline, as they integrate well with visualization tools and are familiar to data professionals. To load a DataFrame to a PostgreSQL database, pandas offers the `to_sql` method. This method requires several parameters, such as `name`, `con`, `if_exists`, `index`, and `index_label`.

Persisting data to Postgres with pandas 00:58 - 02:41

Persisting data to Postgres with pandas

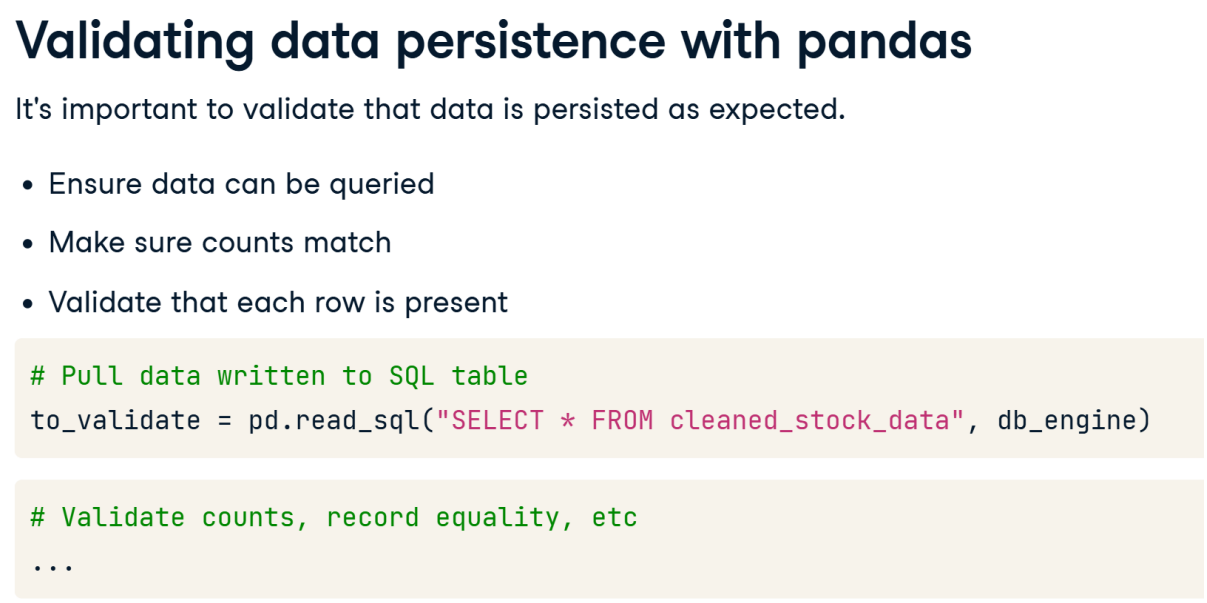
```
# Create a connection object
connection_uri = "postgresql+psycopg2://repl:password@localhost:5432/market"
db_engine = sqlalchemy.create_engine(connection_uri)
```

```
# Use the .to_sql() method to persist data to SQL
clean_stock_data.to_sql(
    name="filtered_stock_data",
    con=db_engine,
    if_exists="append",
    index=True,
    index_label="timestamps"
)
```

First, create a connection object to load data into SQL by forming a connection URI with SQLAlchemy. Here, the URI includes engine details like username, password, host, port, and the database name "market". Calling `to_sql` on `clean_stock_data` writes it to the "filtered_stock_data" table in the "market"

database. The `if_exists` parameter controls table behavior with options like `"append"` to add or `"replace"` to overwrite. Setting `index=True` includes the DataFrame index in the table, and `index_label` names the SQL index column, here as "timestamps".

Validating data persistence with pandas 02:41 - 03:50



After loading data, validating persistence is essential to maintain data integrity and trust. Start by ensuring the data is queryable and that record counts match between transformed and persisted data. With pandas' `read_sql` function, retrieve the persisted data and compare it with the original transformed DataFrame. This validation process strengthens your pipeline by providing data quality checks, fostering robust data solutions

---

Manually testing a data pipeline

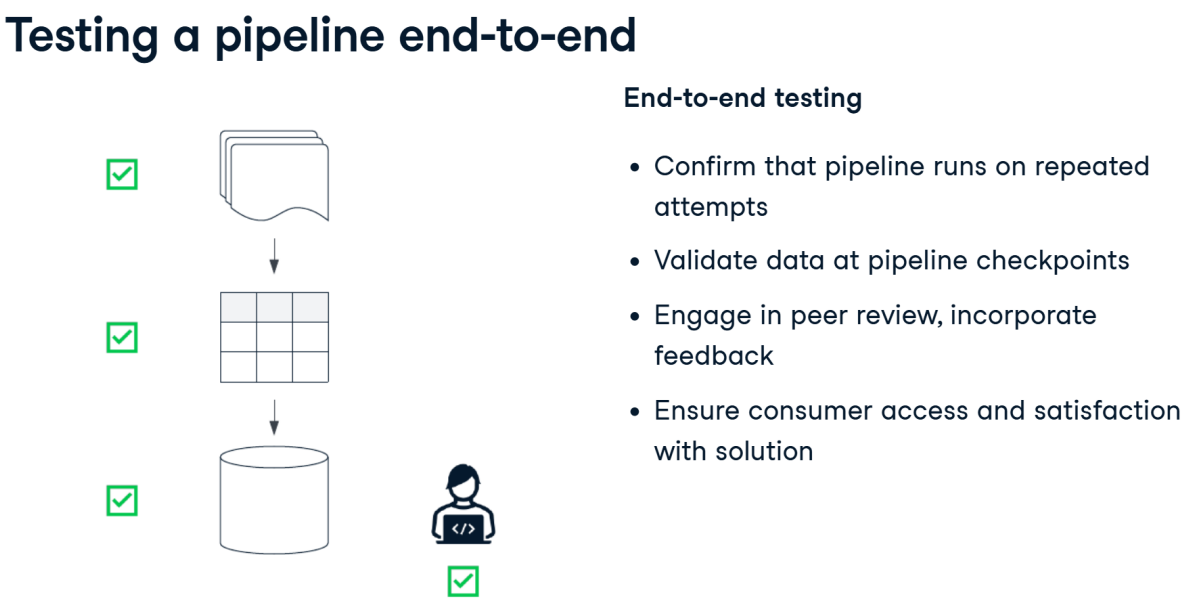
Testing data pipelines 00:07 - 00:59

Testing data pipelines is essential for production-readiness, ensuring data is correctly extracted, transformed, and loaded, which in turn reduces maintenance efforts and improves data reliability. Testing is often overlooked but plays a critical role in identifying data quality issues. Testing can be challenging due to dependencies or data variability. In this lesson, we'll explore end-to-end testing, checkpoint validation, and later, unit testing.

Testing and production environments 00:59 - 01:33

Testing and production environments are standard in data and software engineering. Testing environments allow experimentation with sample data without impacting data consumers. In this safe setting, developers can refine pipeline functions without affecting live data.

Testing a pipeline end-to-end 01:33 - 02:16



End-to-end testing validates pipeline performance by processing sample data through extraction, transformation, and loading stages in a testing environment. This process verifies consistent execution and facilitates checkpoint validation, enabling peer review and consumer feedback on test data accessibility and solution effectiveness.

Validating pipeline checkpoints 02:16 - 03:15

Validating pipeline checkpoints

```
# Extract, transform, and load data as part of a pipeline
...

# Take a look at the data made available in a Postgres database
loaded_data = pd.read_sql("SELECT * FROM clean_stock_data", con=db_engine)
print(loaded_data.shape)
```

```
(6438, 4)
```

```
print(loaded_data.head())
```

```
         timestamps      volume      open     close
1997-05-15 13:30:00  1443120000  0.121875  0.097917
1997-05-16 13:30:00   294000000  0.098438  0.086458
1997-05-19 13:30:00   122136000  0.088021  0.085417
```

Monitoring checkpoints within the pipeline is crucial to prevent data loss. A checkpoint, such as the one after loading, allows validation of the final dataset's integrity. For example, after writing `clean_stock_data` to a Postgres database, we can verify correctness by querying the data into a DataFrame called `loaded_data` and checking its shape and content for completeness and accuracy.

Validating DataFrames 03:15 - 03:45

Validating DataFrames

```
# Extract, transform, and load data, as part of a pipeline
...

# Take a look at the data made available in a Postgres database
loaded_data = pd.read_sql("SELECT * FROM clean_stock_data", con=db_engine)

# Compare the two DataFrames.
print(clean_stock_data.equals(loaded_data))
```

```
True
```

To confirm that `loaded_data` matches `clean_stock_data`, use the `.equals` method. This method returns True if both DataFrames are identical, ensuring no data was lost during loading. This simple, reliable tool reinforces data consistency within the pipeline.

Unit-testing a data pipeline

Unit-testing a data pipeline 00:00 - 00:09

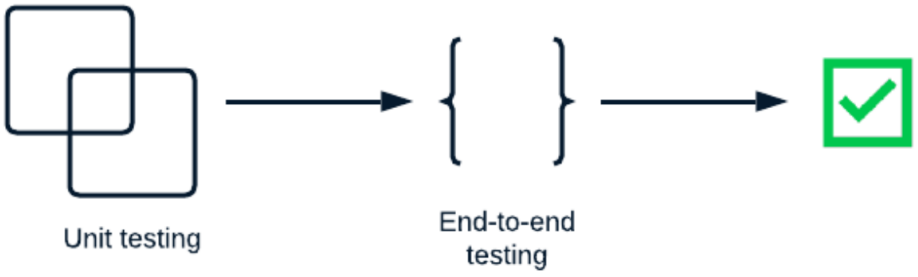
In addition to checkpoint and end-to-end testing, unit tests are essential for ensuring data pipeline components work correctly.

Validating a data pipeline with unit tests 00:09 - 00:42

Validating a data pipeline with unit tests

Unit tests:

- Commonly used in software engineering workflows
- Ensure code works as expected
- Help to validate data



Unit tests verify the functionality of specific parts of code and can confirm that each component of a data pipeline works as intended. Often written and run before end-to-end validation, unit tests help ensure both the code and resulting data meet expectations.

pytest for unit testing 00:42 - 01:31

pytest for unit testing

```
from pipeline import extract, transform, load

# Build a unit test, asserting the type of clean_stock_data
def test_transformed_data():
    raw_stock_data = extract("raw_stock_data.csv")
    clean_stock_data = transform(raw_data)
    assert isinstance(clean_stock_data, pd.DataFrame)
```

```
> python -m pytest

test_transformed_data .                                                  [100%]
================================ 1 passed in 1.17s ================================
```

The pytest library is a popular tool for Python unit testing. It automatically detects functions that start with “test” and runs them as tests. For instance, `test_transformed_data` confirms that `clean_stock_data` is a DataFrame. When running `python -m pytest`, tests are executed, and successful tests output a success message.

assert and isinstance 01:31 - 02:25

assert and isinstance

```
pipeline_type = "ETL"

# Check if pipeline_type is an instance of a str
isinstance(pipeline_type, str)
```

```
True
```

```
# Assert that the pipeline does indeed take value "ETL"
assert pipeline_type == "ETL"
```

```
# Combine assert and isinstance
assert isinstance(pipeline_type, str)
```

`isinstance` verifies that an object is of a specified type. When used with `assert`, it confirms a boolean expression is True and raises an AssertionError if it is not. For example, `assert isinstance(clean_stock_data, pd.DataFrame)` confirms the type of `clean_stock_data` without errors if True.

AssertionError 02:25 - 02:44

AssertionError

```
pipeline_type = "ETL"

# Create an AssertionError
assert isinstance(pipeline_type, float)
```

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
AssertionError
```

When an assertion is False, an AssertionError is raised. For example, asserting a string as a float type raises this error, which would cause a unit test to fail.

Mocking data pipeline components with fixtures 02:44 - 03:22

Mocking data pipeline components with fixtures

```
import pytest

@pytest.fixture()
def clean_data():
    raw_stock_data = extract("raw_stock_data.csv")
    clean_stock_data = transform(raw_data)
    return clean_stock_data
```

```
def test_transformed_data(clean_data):
    assert isinstance(clean_data, pd.DataFrame)
```

pytest fixtures are shared functions that simplify test setups by providing consistent test data across tests. In this example, `clean_data` returns a cleaned

DataFrame, accessible within a test function like `test_transformed_data`.

Unit testing DataFrames 03:22 - 04:05

Unit testing DataFrames

```
def test_transformed_data(clean_data):
    # Include other assert statements here
    ...

    # Check number of columns
    assert len(clean_data.columns) == 4

    # Check the lower bound of a column
    assert clean_data["open"].min() >= 0

    # Check the range of a column by chaining statements with "and"
    assert clean_data["open"].min() >= 0 and clean_data["open"].max() <= 1000
```

Data quality tests can confirm DataFrame content. For instance, a fixture can validate that `clean_data` has four columns, and using `min` and `max` functions, ensure values in a column meet specified requirements. This approach checks data quality and helps identify data issues before deployment.

Running a data pipeline in production

Running a data pipeline in production 00:00 - 00:17

Now that we've explored how to build and test a data pipeline, let's focus on best practices for deploying one in production.

Data pipeline architecture patterns 00:17 - 01:21

Data pipeline architecture patterns

```
# Define ETL function
...
def load(clean_data):
...

# Run the data pipeline
raw_stock_data = extract("raw_stock_data.csv")
clean_stock_data = transform(raw_stock_data)
load(clean_stock_data)
```

```
> ls
 etl_pipeline.py
```

```
# Import extract, transform, and load functions
from pipeline_utils import extract, transform, load

# Run the data pipeline
raw_stock_data = extract("raw_stock_data.csv")
clean_stock_data = transform(raw_stock_data)
load(clean_stock_data)
```

```
> ls
 etl_pipeline.py
 pipeline_utils.py
```

To deploy a pipeline in a production setting, a common approach is to create a script that triggers the extract, transform, and load processes. One way to structure the code is by placing function definitions and the execution logic within the same file, but this can become cumbersome. A better approach is to separate these components: function definitions are stored in a utilities file (e.g., `pipeline_utils.py`), which are then imported into the execution script as needed. This separation improves code readability, maintainability, and debugging.

Running a data pipeline end-to-end 01:21 - 02:13

Running a data pipeline end-to-end

```
import logging
from pipeline_utils import extract, transform, load

logging.basicConfig(format='%(levelname)s: %(message)s', level=logging.DEBUG)
```

```
try:
    # Extract, transform, and load data
    raw_stock_data = extract("raw_stock_data.csv")
    clean_stock_data = transform(raw_stock_data)
    load(clean_stock_data)

    logging.info("Successfully extracted, transformed and loaded data.")  # Log success message

# Handle exceptions, log messages
except Exception as e:
    logging.error(f"Pipeline failed with error: {e}")
```

To run a complete pipeline, we can import functions from `pipeline_utils.py`, configure logging, and encapsulate the ETL process within a try-except block for basic monitoring and alerting. This setup can be executed as a Python script in a production environment to perform the pipeline end-to-end. However, production pipelines often need additional features such as automated scheduling and error handling.

Orchestrating data pipelines in production 02:13 - 03:46

Production-grade data pipelines benefit from orchestration tools that automate scheduling, resource scaling, and error handling. Apache Airflow is a popular open-source tool used in over forty percent of data pipeline implementations for these features, with competitors like Prefect and Dagster offering similar benefits. Choosing the right orchestration tool can expedite pipeline development and enhance functionality, making the pipeline more resilient and efficient.