Extracting data from structure sources

INTRODUCTION TO DATA PIPELINES

Source systems

In this course:

- CSV files
- Parquet files
- JSON files
- SQL databases

Data is also sourced from:

- APIs
- Data lakes
- Data warehouses
- Web scraping
- ... and so many more!

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

```python
import pandas as pd


# Read the parquet file into memory
raw stock data = pd.read parquet("raw stock data.parquet", engine="fastparquet")
```

[1] https://www.databricks.com/glossary/what-is-parquet

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)
```

Modularity

Separating logic into functions

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

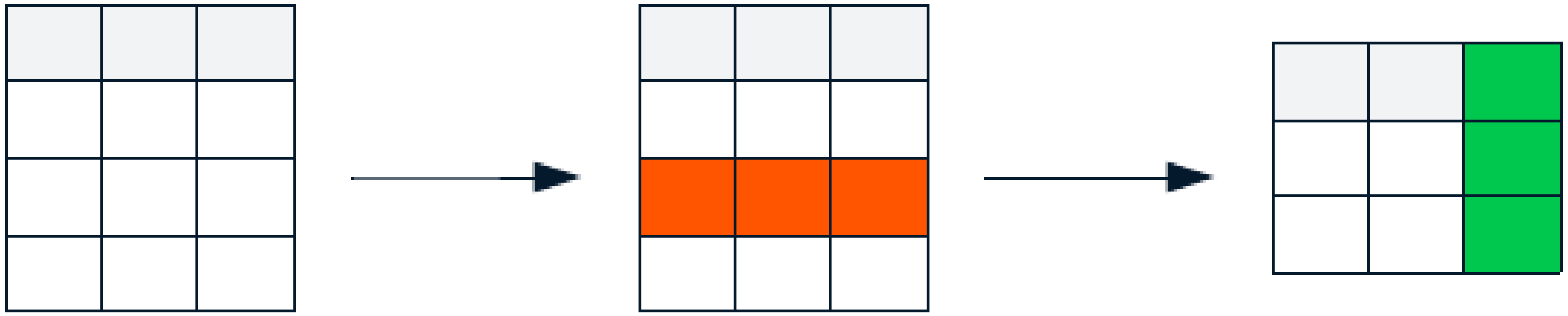
```python
def extract_from_sql(connection_uri, query):
    # Create an engine, query data and return DataFrame
    db_engine = sqlalchemy.create_engine(connection_uri)
    return pd.read_sql(query, db_engine)

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

Transforming data with pandas

INTRODUCTION TO DATA PIPELINES

Transforming data in a pipeline

Data must be properly transformed to ensure value is provided to downstream users



`pandas` provides powerful tools to transform tabular data

- `.loc[]`
- `.to_datetime()`

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]]
```

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')
```

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"]))
```

Persisting data with pandas

INTRODUCTION TO DATA PIPELINES

Persisting data in an ETL pipeline

Loading data to a file:

- Ensures data consumers have stable access to transformed data
- Occurs as a final step in an ETL process, **as well as** between discrete steps
- Captures a "snapshot" of the data

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"`

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()`

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

Ensuring data persistence

Was the DataFrame correctly stored to the CSV file?

```python
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
```

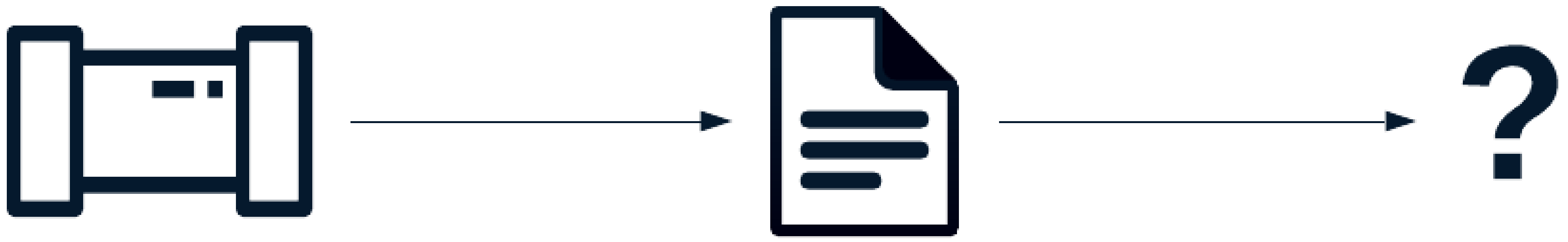
Monitoring a data pipeline

INTRODUCTION TO DATA PIPELINES

Monitoring a data pipeline

Data pipelines should be monitored for changes to data and failures in execution

- Missing data
- Shifting data types
- Package deprecation or functionality change



Logging data pipeline performance

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

```python
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.
```

Logging warnings and errors

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

# Create different types of logs
logging.warning("Unexpected number of rows detected.")
logging.error("{ke} arose in execution.")
```

```
WARNING: Unexpected number of rows detected.
ERROR: KeyError arose in execution.
```

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

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
```