Data Wrangling

1.1 Definition and Importance

Data wrangling—sometimes referred to as **data munging**—is the process of transforming and mapping raw data into a more suitable format for analysis and decision-making. Data wrangling typically encompasses:

- **Cleaning**: Identifying and correcting errors, inconsistencies, and missing values.
- **Structuring**: Organizing data to fit the needs of the analysis (e.g., pivoting or reshaping).
- **Enriching**: Adding context or combining multiple sources to get a more comprehensive dataset.

Why is it important?

- In the real world, data often arrives in inconsistent, incomplete, or unstructured formats.
- High-quality analysis depends on high-quality data. Incorrect, duplicated, or missing data leads to flawed insights.
- Data wrangling can account for up to 80% of a data professional's time, underscoring its critical role in analytics and data science.

1.2 Stages of Data Wrangling

1. **Data Discovery**: Understanding what data is available, where it comes from, and how it is structured.
2. **Data Structuring**: Ensuring data is in the correct format—often tabular—to facilitate analysis.
3. **Data Cleaning**: Handling missing values, removing or correcting erroneous entries, deduplicating records.
4. **Data Enrichment**: Combining data from multiple sources or adding derived features that enhance analysis.

5. **Validation**: Checking that the transformed data is correct, reliable, and fits the intended use.
6. **Documentation**: Recording the steps taken for reproducibility and transparency.

1.3 Real-World Examples

- Converting daily sales text files from different store branches into a single, consistent CSV file for consolidated reporting.
- Merging user profile data from an Excel spreadsheet with transaction records stored in a relational database.
- Parsing unstructured logs (e.g., system logs, web server logs) and transforming them into structured event data.

---

2. Data Loading & Storage

2.1 Common Data Sources

- **Flat files**: Comma-Separated Values (CSV), Tab-Separated Values (TSV), JSON, and XML.
- **Spreadsheets**: Microsoft Excel files (`.xlsx`), often used for manual data entry or small-scale storage.
- **Databases**: SQL (e.g., MySQL, PostgreSQL) or NoSQL (e.g., MongoDB, Cassandra).
- **Web Services and APIs**: Retrieving data via HTTP requests, often returning JSON or XML.
- **Cloud Services**: Storing and retrieving data from Amazon S3, Google Cloud Storage, Azure Blob, etc.

2.2 Local vs. Remote Storage

- **Local**: Data is stored on your personal computer or a single on-premises server.

    - *Pros*: Fast access (limited by hardware), easy to control environment.
    - *Cons*: Limited by local machine's storage and processing capabilities, collaboration can be more difficult.
- **Remote**: Data is stored on remote servers or cloud services.
    - *Pros*: Can handle large-scale storage, easier to share data among distributed teams.
    - *Cons*: Potential network latency, costs for data egress, reliance on an internet connection.

2.3 File Organization and Naming Conventions

Establish a structured file directory and naming system to streamline your workflow. For example:

- **Year/Month/Project**: `data/2024/12/marketing_campaign.csv`
- **Versioning**: `sales_data_v1.csv`, `sales_data_v2.csv`
- **Descriptive Filenames**: `customer_reviews_2024_Q4.csv` instead of `data_v3.csv`

2.4 Understanding File Formats and Their Trade-offs

- **CSV/TSV**:
    - *Pros*: Human-readable, widely supported.
    - *Cons*: Can be large (no compression), no built-in schema, potential issues with special characters or delimiters.
- **Excel**:
    - *Pros*: Common in business settings, user-friendly interface.
    - *Cons*: Not ideal for large datasets, multiple worksheets can complicate automation.
- **JSON/XML**:
    - *Pros*: Hierarchical, flexible, widely used by APIs.
    - *Cons*: Parsing can be more complex than CSV; can be verbose.
- **Binary formats** (like Parquet, Feather, HDF5) often provide better performance and compression than plain text.

3. Reading and Writing Text Files (CSV, TSV, etc.)

3.1 Structure of Text Files

Text files, especially CSV and TSV, follow a row-and-column structure using special characters to separate (or *delimit*) columns. For instance:

- **CSV** (Comma-Separated Values): Each line represents a row of data, and commas separate columns.
- **TSV** (Tab-Separated Values): Similar to CSV but uses tabs as delimiters.

Example CSV Content:

```
Name,Age,Occupation
Alice,30,Engineer
Bob,25,Designer
Charlie,35,Data Scientist
```

3.2 Delimiters, Quotes, and Special Characters

- **Delimiters**: Commas, tabs, semicolons, or even custom delimiters like `|`.
- **Quoting**: Text fields that contain delimiter characters or line breaks are often enclosed in quotes (e.g., `"New York, NY"`).
- **Escape characters**: Sometimes `\` is used to escape quotes or other special characters.

Potential Pitfalls:

- Inconsistent use of delimiters (some lines might use commas, others might use tabs).
- Special characters in data (like commas in addresses) can break parsing if not quoted properly.
- Character encoding issues (e.g., CSV saved as `Latin-1` vs. `UTF-8`).

3.3 Using Programming Tools to Read and Write Text Files

While there are many programming languages and libraries for working with CSV/TSV files, Python’s **pandas** library is one of the most common choices.

1. **Reading CSV Files**:

   ```
   import pandas as pd

   # Basic usage
   df = pd.read_csv('data.csv')

   # Custom delimiter (tab)
   df_tab = pd.read_csv('data.tsv', sep='\t')

   # Handling headers and naming columns
   df_no_header = pd.read_csv('data_no_header.csv',
   header=None, names=['col1', 'col2', 'col3'])
   ```

   - Parameters like `header`, `names`, `skiprows`, and `nrows` allow for fine-grained control.
   - `encoding='utf-8'` (or another encoding) may be needed to properly read files containing special characters.

2. **Writing CSV Files**:

   ```
   # Write to a CSV file (default comma-delimited)
   df.to_csv('output.csv', index=False)

   # Write with a different delimiter
   df.to_csv('output.tsv', index=False, sep='\t')
   ```

   - You can omit the DataFrame’s index if it isn’t meaningful as a separate column in the final output (`index=False`).

3.4 Handling Large Files

For very large text files (e.g., several gigabytes or more), reading the entire file into memory at once can be inefficient or impossible on limited-memory systems. Strategies include:

- **Chunking**: Read the file in smaller parts using the `chunksize` parameter in `pd.read_csv()`, then process each chunk sequentially.
- **Sampling**: If you only need a quick preview of the data, load the first few rows (`nrows=100`) to understand the structure.
- **Database Integration**: Consider loading very large files into a database for more efficient querying and partial reads.

---

4. Key Pitfalls & Performance Tips

4.1 Data Type Inference

- **Auto-detection**: Libraries like pandas will attempt to guess data types. Numerical columns might default to `float64`, while anything not recognized becomes a string (`object` in pandas).
- **Explicitly Setting Types**: Use parameters like `dtype` when calling `pd.read_csv()` to avoid incorrect guessing.
- *Example*: `df = pd.read_csv('data.csv', dtype={'id': str, 'price': float})`

4.2 Encoding Issues

- **Common Encodings**: UTF-8, Latin-1, ISO-8859-1, etc.
- **Decoding Errors**: If a file is not correctly read due to wrong encoding, you may see `UnicodeDecodeError` or garbled text.
- **Tip**: Use `df = pd.read_csv('data.csv', encoding='latin-1')` if your system can’t handle UTF-8 or if you know the file is in another encoding.

4.3 Memory Constraints

- **Symptom**: “Out of memory” errors or system slowdown when reading large datasets.
- **Solutions**:
  - Use data types that require less memory (e.g., `float32` instead of `float64`, or `int8` if the data range is small).
  - Filter out unnecessary columns upon reading if you only need a subset.
  - Consider chunk processing or distributed computing frameworks (like Dask or PySpark) for extremely large datasets.

4.4 Dealing with Inconsistent or Dirty Data

- **Duplicates**: Identify them with `.duplicated()` or remove them with `.drop_duplicates()`.
- **Missing Values**: Some CSVs may encode missing values as `NA`, `N/A`, `null`, or `?`. Use parameters like `na_values=['?', 'NA', 'n/a']` to handle them explicitly.
- **Irregular Rows**: Rows with fewer or more columns than expected can cause parse errors.

4.5 Performance Optimization

- **Compression**: If space or I/O speed is a concern, consider reading/writing compressed files: `df.to_csv('output.csv.gz', compression='gzip')`.
- **Indexing**: If the data is frequently queried on a particular column, consider setting it as an index for faster lookups.
- **Hardware Considerations**: SSDs over HDDs can drastically improve read/write speeds.

Review

1. **Data Wrangling** is essential for turning raw data into actionable insights.

2. **Data Loading & Storage** covers everything from local file management to handling remote sources.
3. **Text File Formats** (CSV, TSV) are among the most common and straightforward ways to store and share data, but they come with their own quirks, especially with delimiters, quotes, and encodings.
4. **Key Pitfalls** such as large file handling, memory constraints, and data type/encoding issues should be addressed proactively for efficient, error-free wrangling.

Web Scraping and Binary Data Formats

**2.1 Introduction to Web Scraping

2.1.1 What is Web Scraping?

Web scraping is the process of programmatically extracting data from websites. Rather than manually copying and pasting information, you use scripts or tools to automate the collection of data from HTML pages. Some common use cases include:

- Gathering product prices from e-commerce sites for competitive analysis.
- Extracting articles or blog posts from online publications for text mining.
- Collecting social media or forum posts to analyze trends or sentiment.

2.1.2 Relevance to Data Wrangling

Although web scraping itself is not purely "data wrangling," it is often the very first step in collecting raw, unstructured data from websites. Once scraped, the data typically requires cleaning, restructuring, and integration—activities that fall squarely under the umbrella of data wrangling.

**2.2 Ethical and Legal Considerations

Before diving into technical details, it is crucial to understand the ethical and legal aspects of web scraping:

1. **Website Terms of Service (ToS)**: Many sites prohibit or restrict automated data collection. Always check the ToS to ensure compliance.
2. **Robots.txt**: This file (e.g., `example.com/robots.txt`) often outlines which parts of a website can be crawled by automated tools. Respect these directives.
3. **Rate Limiting and Politeness**: Sending too many requests in a short period can overwhelm a server. Throttling requests (adding delays) helps avoid being blocked or causing denial-of-service issues.
4. **Data Privacy**: Be mindful of scraping personal data or other sensitive information. Privacy laws may restrict what data can be collected and how it can be stored or shared.

**2.3 Common Tools and Libraries for Web Scraping

2.3.1 HTTP Libraries

- **Requests (Python)**: A popular Python library to handle HTTP requests. Provides methods like `requests.get(url)` or `requests.post(url, data=...)` for interacting with web pages and APIs.
- **Other Languages**: Node.js (Axios), Ruby (Net::HTTP), Java (Apache HttpClient), etc.

2.3.2 HTML Parsing Libraries

- **Beautiful Soup (Python)**: Simplifies HTML or XML parsing. Allows you to search for elements using tags, classes, or IDs.
- **lxml (Python)**: A faster, more low-level parser for HTML/XML.

- **Cheerio (JavaScript)**: Node.js library offering jQuery-like manipulation of HTML.

2.3.3 Browser Automation

- **Selenium**: Automates a real web browser (Chrome, Firefox, etc.) to handle dynamic, JavaScript-heavy pages.
- **Playwright / Puppeteer**: Headless browser automation tools by Microsoft (Playwright) and Google (Puppeteer) for complex JavaScript-driven sites.

When to choose browser automation?

- If the site relies heavily on JavaScript to generate content or requires login/session handling.
- If you need to interact with buttons, dropdown menus, or other dynamic elements.

2.4 Basic Workflow for Web Scraping

1. **Identify the Target Website**
   - Inspect the site's structure: which pages contain the data you need? Is it paginated?
   - Check `robots.txt` and Terms of Service.
2. **Make a Request**
   - Use an HTTP library like `requests.get(url)` to fetch the raw HTML of a page.
3. **Parse the Response**
   - Parse HTML using Beautiful Soup or a similar library:

     ```
     from bs4 import BeautifulSoup
     import requests

     url = 'https://example.com'
     ```

```
response = requests.get(url)
soup = BeautifulSoup(response.text, 'html.parser')
```

4. **Locate and Extract Data**
   - Use the website's HTML structure to locate tags (`<div>`, `<span>`, etc.), classes, or IDs.
   - Extract text, attributes (like `href` from a link), or other embedded data.
5. **Store or Export**
   - Convert scraped data into a structured format (CSV, JSON, etc.).
   - Save to a file or insert into a database.
6. **Repeat for Pagination (if needed)**
   - Many sites split content across multiple pages. Identify the pattern for the "Next" page URL or page parameters.
   - Loop through all pages to collect a complete dataset.

**2.5 Practical Example: Basic Web Scraping with Python

Imagine you want to scrape a simple website listing books with their titles, authors, and prices. The steps might look like this:

1. **Find the URL**: e.g., `https://examplebooks.com/catalogue?page=1`
2. **Analyze HTML**:
   - Each book is wrapped in a `<div class="book-item">`
   - Title is in a `<h2>` tag, author in a `<p class="author">`, price in a `<p class="price">`
3. **Write a Script** (simplified code):

```
import requests
from bs4 import BeautifulSoup
import csv

data_list = []
```

```python
for page_num in range(1, 6):  # Suppose there are 5 pages
    url = f"https://examplebooks.com/catalogue?page=
{page_num}"
    response = requests.get(url)
    soup = BeautifulSoup(response.text, 'html.parser')

    book_items = soup.find_all('div', class_='book-item')
    for item in book_items:
        title = item.find('h2').text.strip()
        author = item.find('p',
class_='author').text.strip()
        price = item.find('p',
class_='price').text.strip()
        data_list.append([title, author, price])

# Save to CSV
with open('books.csv', 'w', newline='', encoding='utf-8')
as f:
    writer = csv.writer(f)
    writer.writerow(['Title', 'Author', 'Price'])
    writer.writerows(data_list)
```

4. **Check Results**:
   - Verify that `books.csv` contains the columns `Title, Author, Price` for each book.

**2.6 Common Challenges in Web Scraping

1. **JavaScript-Rendered Content**
   - Some sites load data dynamically using JavaScript frameworks (React, Angular, Vue). The HTML you receive from `requests` might not contain all the data.
   - **Solution**: Use a headless browser like **Selenium**, **Playwright**, or **Puppeteer**.
2. **CAPTCHAs and Anti-Bot Measures**
   - Websites may deploy CAPTCHAs to block automated scraping.

    - **Solution**: In some cases, it may be illegal or unethical to bypass these measures. Consult legal guidelines; in other cases, you might use third-party CAPTCHA-solving services (with caution and compliance).
3. **Rate Limits and Throttling**
    - Frequent, rapid requests might get your IP blocked.
    - **Solution**: Introduce delays in your scraping script, use exponential backoff, or store cookies/session tokens responsibly.
4. **Session Handling**
    - Some sites require login or session cookies to view content.
    - **Solution**: Use requests' session features or browser automation.
5. **Changing Page Structure**
    - Websites update their layouts frequently, which can break your scrapers.
    - **Solution**: Maintain your code actively, check for layout changes, or implement robust error handling.

**2.7 Introduction to Binary Data Formats

After discussing how to collect data from the web, we shift our focus to **binary data formats**—an efficient way to store large, structured datasets. Unlike text files (CSV/TSV), binary formats are not directly human-readable but are often faster to read, write, and query.

2.7.1 Why Use Binary Formats?

1. **Speed**: Reading and writing can be significantly faster than CSV, especially for large datasets.
2. **Compression**: Many binary formats incorporate compression algorithms, reducing file size and I/O overhead.
3. **Schema**: Some formats store metadata about data types, column names, and other schema information.

2.7.2 Common Binary Formats

- **Parquet**: A columnar format widely used in the Apache Hadoop ecosystem (Spark, Hive). Great for analytical workloads because it allows selective reading of only required columns.
- **Feather**: Based on Apache Arrow; works well with pandas DataFrames in Python and R for quick reading/writing.
- **HDF5 (Hierarchical Data Format)**: Ideal for storing large numeric arrays, often used in scientific computing.

2.7.3 Working with Parquet (Example in Python)

```python
import pandas as pd

# Writing a pandas DataFrame to Parquet
df.to_parquet('data.parquet', index=False)

# Reading a Parquet file
df_parquet = pd.read_parquet('data.parquet')
```

Advantages:

- Columnar storage: faster analytical queries.
- Built-in compression and efficient encoding.

2.7.4 When to Choose Binary Over Text Formats

- **Large Datasets**: If your dataset is several GB or TB in size, CSV parsing can become a bottleneck.
- **Frequent Reads/Writes**: If you need to repeatedly read and write the same dataset, binary formats save both time and space.
- **Integration with Data Pipelines**: Tools like Apache Spark natively handle Parquet. This can significantly accelerate distributed processing.

Points to ponder:

1. **When to Use a Simple HTTP Request vs. a Headless Browser?**

    - Evaluate if the site relies on JavaScript to render crucial data or if plain HTML is sufficient.
2. **Handling Large-Scale Scraping**
    - Consider scheduling your scrapes, distributing tasks, or using scraping frameworks (Scrapy, etc.).
3. **Selecting a Binary Format**
    - Parquet vs. Feather vs. HDF5: which best fits your data's shape, size, and usage patterns?
