Great! Let’s move forward by adding more advanced features to your \*\*Text-Based Data Cleaning Assistant\*\*. We'll focus on:

1. \*\*Advanced Cleaning Functions\*\* (Pattern Matching and Validation)

2. \*\*Custom Cleaning Rules\*\* (Using a Configuration File)

3. \*\*Error Handling and Logging\*\*

---

### \*\*1. Advanced Cleaning Functions: Pattern Matching and Data Validation\*\*

We can add some functions that validate specific data formats, like:

- \*\*Emails\*\*: Ensure email addresses have a valid format.

- \*\*Phone Numbers\*\*: Standardize phone numbers.

- \*\*Dates\*\*: Format and validate date entries.

#### \*\*a. Email Validation\*\*

To validate email addresses, we can use a regular expression (regex).

```python

import re

def validate\_email\_format(rows):

"""Validates and corrects email format in the data."""

email\_regex = r'^[a-zA-Z0-9\_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$'

new\_rows = []

for row in rows:

new\_row = [cell if not isinstance(cell, str) or re.match(email\_regex, cell) else 'Invalid Email' for cell in row]

new\_rows.append(new\_row)

return new\_rows

```

\*\*Explanation\*\*:

- We use a regular expression to match valid email patterns.

- If an invalid email is found, it’s replaced with `"Invalid Email"` (or you can flag it however you wish).

#### \*\*b. Phone Number Standardization\*\*

We can standardize phone numbers to a specific format (e.g., `(XXX) XXX-XXXX`).

```python

def standardize\_phone\_numbers(rows):

"""Standardizes phone numbers to the format (XXX) XXX-XXXX."""

phone\_regex = r'(\d{3})[^\d]\*(\d{3})[^\d]\*(\d{4})'

new\_rows = []

for row in rows:

new\_row = [re.sub(phone\_regex, r'(\1) \2-\3', cell) if isinstance(cell, str) and re.search(phone\_regex, cell) else cell for cell in row]

new\_rows.append(new\_row)

return new\_rows

```

\*\*Explanation\*\*:

- We use regex to extract the digits of the phone number and reformat them to a standardized format.

- This assumes North American phone numbers as an example; you can customize it for different formats.

#### \*\*c. Date Validation\*\*

We’ll also check and reformat dates to a specific format (e.g., `YYYY-MM-DD`).

```python

from datetime import datetime

def validate\_and\_format\_dates(rows, date\_format="%Y-%m-%d"):

"""Validates and formats dates to the specified format."""

new\_rows = []

for row in rows:

new\_row = []

for cell in row:

if isinstance(cell, str):

try:

# Try parsing the date

parsed\_date = datetime.strptime(cell, "%m/%d/%Y")

new\_row.append(parsed\_date.strftime(date\_format))

except ValueError:

new\_row.append(cell) # Leave as is if not a valid date

else:

new\_row.append(cell)

new\_rows.append(new\_row)

return new\_rows

```

\*\*Explanation\*\*:

- This function converts dates from `MM/DD/YYYY` to `YYYY-MM-DD`.

- You can modify the `date\_format` to fit other formats as needed.

---

### \*\*2. Custom Cleaning Rules Using a Configuration File\*\*

We can allow users to define custom cleaning rules using a configuration file (e.g., JSON or YAML). This file would specify which cleaning functions to apply, and any specific patterns or formats.

#### \*\*Sample JSON Configuration\*\*

Create a file `cleaning\_rules.json` that looks like this:

```json

{

"trim\_whitespace": true,

"convert\_to\_lowercase": true,

"remove\_duplicates": true,

"validate\_email\_format": true,

"standardize\_phone\_numbers": false,

"date\_format": "YYYY-MM-DD"

}

```

#### \*\*Reading the Configuration File\*\*

We can read this configuration and apply cleaning rules accordingly.

```python

import json

def load\_config(file\_path):

"""Loads cleaning rules from a JSON config file."""

try:

with open(file\_path, 'r') as file:

config = json.load(file)

return config

except FileNotFoundError:

print(f"Config file '{file\_path}' not found.")

return {}

```

#### \*\*Updating the Main Block to Use Configurations\*\*

We’ll modify the script to apply functions based on the configuration.

```python

if \_\_name\_\_ == "\_\_main\_\_":

# Load the configuration

config\_path = input("Enter the path to the configuration file: ")

config = load\_config(config\_path)

# Step 1: Read the input CSV

input\_path = input("Enter the path to your input CSV file: ")

data = read\_csv(input\_path)

if data:

print("Cleaning data based on the configuration...")

# Apply functions based on config

if config.get('trim\_whitespace', False):

data = trim\_whitespace(data)

if config.get('convert\_to\_lowercase', False):

data = convert\_to\_lowercase(data)

if config.get('remove\_duplicates', False):

data = remove\_duplicates(data)

if config.get('validate\_email\_format', False):

data = validate\_email\_format(data)

if config.get('standardize\_phone\_numbers', False):

data = standardize\_phone\_numbers(data)

if config.get('date\_format'):

data = validate\_and\_format\_dates(data, config['date\_format'])

# Step 3: Write the cleaned data

output\_path = input("Enter the path to save your cleaned CSV file: ")

write\_csv(output\_path, data)

else:

print("No data to process.")

```

---

### \*\*3. Error Handling and Logging\*\*

We’ll add basic error handling and logging to make it easier to debug issues or track what’s happening.

#### \*\*a. Basic Error Handling\*\*

We’ve already added error handling to the file I/O functions. Let’s expand that to ensure no data corruption during cleaning:

```python

def handle\_exceptions(func):

"""A decorator to handle exceptions in cleaning functions."""

def wrapper(\*args, \*\*kwargs):

try:

return func(\*args, \*\*kwargs)

except Exception as e:

print(f"Error occurred in {func.\_\_name\_\_}: {e}")

return args[0] # Return original data on failure

return wrapper

```

You can then wrap each cleaning function with this decorator:

```python

@handle\_exceptions

def trim\_whitespace(rows):

# Function body remains the same...

```

#### \*\*b. Adding Logging\*\*

We’ll use Python’s `logging` module to log actions during the cleaning process:

```python

import logging

# Set up logging

logging.basicConfig(filename='cleaning.log', level=logging.INFO)

# Modify cleaning functions to log actions

def log\_action(action, detail=""):

logging.info(f"{action}: {detail}")

@handle\_exceptions

def trim\_whitespace(rows):

log\_action("Trimming whitespace")

# The function remains the same...

```

Now, every time a function is called, it logs the action to `cleaning.log`.

---

### \*\*4. Bringing it All Together\*\*

Here’s how the full project might look now:

1. \*\*Main Script\*\*: Handles file I/O, cleaning logic, and applying user-defined rules.

2. \*\*Configuration File\*\*: Specifies the cleaning operations to perform.

3. \*\*Logging and Error Handling\*\*: Provides feedback and debugging support.

---

### \*\*Next Steps\*\*

At this point, your \*\*Text-Based Data Cleaning Assistant\*\* is quite powerful. Here’s what you could explore next:

1. \*\*GUI\*\*: Create a simple GUI using libraries like `Tkinter` to make the tool more user-friendly.

2. \*\*Performance\*\*: If you’re dealing with very large files, consider using libraries like `pandas` for efficient data manipulation.

3. \*\*More Validation\*\*: Add other validation types, like checking for missing values or incorrect data types.

Would you like to implement one of these enhancements, or would you like me to help you test and refine what we’ve already built?