Modify following code to get entire transaction data instead og only few IDs as mentioned in beginning of the code. Also, optimize code using python tools like list comprehension, vectorization, etc: @xw.func def process\_pandas(file\_path): IDS = [32018, 30284, 30355, 31991, 32229, 32162, 32029, 29681, 30234, 28354, 31331, 32179, 29515, 30759, 31313, 30362, 29072, 29708, 30457, 30338, 29026, 30026, 27356, 29213, 28697, 13306, 12152] IDS = list(map(str,IDS)) # Read the CSV file into a pandas DataFrame with headers and quotes set to true df = pd.read\_csv(file\_path, header=0, quotechar='"') df.columns = df.columns.str.strip() df['Project: ID'] = df['Project: ID'].fillna('') # Create the 'ID' column using vectorized regex operations id\_pattern = r'(^\d+(?=\s-)|^\d+\_\d+(?=\s-))' df['ID'] = df['Project: ID'].str.extract(id\_pattern, expand=False).fillna('') # Sanitize 'Amount' and 'Qty' columns df['Amount'] = df['Amount'].str.replace(r',', '', regex=True).str.replace(r'\$', '', regex=True).str.replace(r'\(', '-', regex=True).str.replace(r'\)', '', regex=True) df['Amount'] = pd.to\_numeric(df['Amount'], errors='coerce') df['Qty'] = df['Qty'].astype(str).str.replace(r',', '', regex=True).str.replace(r'\$', '', regex=True).str.replace(r'\(', '-', regex=True).str.replace(r'\)', '', regex=True) df['Qty'] = pd.to\_numeric(df['Qty'], errors='coerce') # Convert 'Date' column to datetime df['Date'] = pd.to\_datetime(df['Date'], format='%m/%d/%Y') final\_results = [] # Process each ID for ID1 in IDS: filtered\_df = df[(df['ID'] == ID1) & (df['Type'] != 'Purchase Order') & (df['Type'] != 'Sales Order') & (~df['Account'].str.contains(r'(?i)advances', na=False)) & (~df['Account'].str.contains(r'(?i)payable', na=False))] #& (df['Date'] < pd.Timestamp('2024-07-01'))] grouped\_df = filtered\_df.groupby(['Type', 'Project: ID', 'Account']).agg({'Amount': 'sum'}).reset\_index() grouped\_df['Amt'] = -grouped\_df['Amount'].where(grouped\_df['Amount'] > 0, grouped\_df['Amount']) grouped\_df.drop(columns=['Amount'], inplace=True) final\_results.append(grouped\_df) # Combine all results into a single DataFrame combined\_results = pd.concat(final\_results, ignore\_index=True) # Sort the combined results by 'Project: ID' in ascending order combined\_results.sort\_values(by=['Project: ID', 'Amt'], inplace=True) # Add headers to the output data output\_data = [combined\_results.columns.tolist()] + combined\_results.values.tolist() # Return the combined results with headers as a dynamic array return output\_data

Error creating bedrock response

Generate again

Error creating bedrock response

Generate again

Here's the modified and optimized version of the code that processes all transactions instead of specific IDs:

python

@xw.func

def process\_pandas(file\_path):

# Read the CSV file into a pandas DataFrame with headers and quotes set to true

df = pd.read\_csv(file\_path, header=0, quotechar='"')

# Clean and prepare the DataFrame using vectorized operations

df.columns = df.columns.str.strip()

df['Project: ID'] = df['Project: ID'].fillna('')

# Extract IDs using vectorized regex

df['ID'] = df['Project: ID'].str.extract(r'(^\d+(?=\s-)|^\d+\_\d+(?=\s-))', expand=False).fillna('')

# Clean numeric columns using vectorized operations

for col in ['Amount', 'Qty']:

df[col] = (df[col].astype(str)

.replace([r',', r'\$', r'\(', r'\)'], ['', '', '-', ''], regex=True)

.pipe(pd.to\_numeric, errors='coerce'))

# Convert date column

df['Date'] = pd.to\_datetime(df['Date'], format='%m/%d/%Y')

# Filter DataFrame using boolean indexing

filtered\_df = df[

(df['Type'].isin(['Purchase Order', 'Sales Order']) == False) &

(~df['Account'].str.contains('advances|payable', case=False, na=False))

# & (df['Date'] < pd.Timestamp('2024-07-01')) # Uncomment if date filtering is needed

]

# Group and aggregate data

result\_df = (filtered\_df.groupby(['Type', 'Project: ID', 'Account'])

.agg({'Amount': 'sum'})

.reset\_index())

# Calculate Amt column

result\_df['Amt'] = -result\_df['Amount'].where(result\_df['Amount'] > 0, result\_df['Amount'])

result\_df.drop(columns=['Amount'], inplace=True)

# Sort results

result\_df.sort\_values(by=['Project: ID', 'Amt'], inplace=True)

# Convert to output format

output\_data = [result\_df.columns.tolist()] + result\_df.values.tolist()

return output\_data

Key improvements made:

Removed the IDS list filtering - now processes all transactions

Used vectorized operations instead of loops

Simplified data cleaning using method chaining

Combined multiple string replacements into single operations

Used boolean indexing for filtering

Improved groupby operation efficiency

Removed unnecessary list comprehensions and loops

Used more efficient pandas methods like .pipe()

The code now:

Processes all transactions instead of specific IDs

Is more efficient due to vectorized operations

Is more readable with better structured operations

Maintains the same output format

Handles the same data cleaning and transformations

Performs the same filtering and grouping operations

The output will include all transactions that meet the filtering criteria, sorted by Project ID and Amount, rather than just the previously specified IDs.

Modify code - I have following columns: ProjectSOPO ,Class: Name ,Type ,Date ,Document Number ,Name ,Memo ,Account ,Clr ,Split ,Qty ,Amount Out of these Date should type date, Qty and Amount should be float.

Here's the modified code that handles the specific columns you mentioned:

python

@xw.func

def process\_pandas(file\_path):

# Define column dtypes for efficient reading

dtype\_dict = {

'ProjectSOPO': str,

'Class: Name': str,

'Type': str,

'Document Number': str,

'Name': str,

'Memo': str,

'Account': str,

'Clr': str,

'Split': str

}

# Read CSV with specified dtypes

df = pd.read\_csv(

file\_path,

header=0,

quotechar='"',

dtype=dtype\_dict,

parse\_dates=['Date'] # Automatically parse Date column

)

# Clean column names

df.columns = df.columns.str.strip()

# Clean numeric columns (Qty and Amount)

numeric\_cols = ['Qty', 'Amount']

for col in numeric\_cols:

df[col] = (df[col].astype(str)

.replace({

r',': '',

r'\$': '',

r'\(': '-',

r'\)': '',

r'[^\d.-]': '' # Remove any other non-numeric characters

}, regex=True)

.pipe(pd.to\_numeric, errors='coerce')

.astype(float))

# Filter DataFrame

filtered\_df = df[

(~df['Type'].isin(['Purchase Order', 'Sales Order'])) &

(~df['Account'].str.contains('advances|payable', case=False, na=False))

# & (df['Date'] < pd.Timestamp('2024-07-01')) # Uncomment if date filtering is needed

]

# Group and aggregate data

result\_df = (filtered\_df.groupby(['Type', 'ProjectSOPO', 'Account'])

.agg({'Amount': 'sum'})

.reset\_index())

# Calculate Amt column

result\_df['Amt'] = -result\_df['Amount'].where(result\_df['Amount'] > 0, result\_df['Amount'])

result\_df.drop(columns=['Amount'], inplace=True)

# Sort results

result\_df.sort\_values(by=['ProjectSOPO', 'Amt'], inplace=True)

# Convert to output format

output\_data = [result\_df.columns.tolist()] + result\_df.values.tolist()

return output\_data

Key changes made:

Specified the exact columns you're working with

Added proper data type handling:

Date column is parsed automatically as datetime using parse\_dates

Qty and Amount are converted to float after cleaning

Other columns are read as strings using dtype\_dict

Used 'ProjectSOPO' instead of 'Project: ID' in grouping and sorting

Improved numeric cleaning with a more comprehensive regex pattern

Maintained the same filtering and aggregation logic

The code now:

Properly handles all specified columns

Ensures correct data types:

Date as datetime

Qty and Amount as float

Other columns as strings

Maintains efficient vectorized operations

Handles missing values and invalid numeric entries

Produces the same grouped and sorted output format

You can further customize the code by:

Adding more specific data cleaning steps if needed

Modifying the grouping columns

Adding additional filters

Changing the sorting criteria

Adding more aggregations in the groupby operation

Let me know if you need any adjustments or have specific requirements for any of these columns!

Why this error: "ValueError: Missing column provided to 'parse\_dates': 'Date' ) f""Missing column provided to 'parse\_dates': '{missing\_cols}'"" raise ValueError( File ""D:\Programs\python\Lib\site-packages\pandas\io\parsers\base\_parser.py"", line 243, in \_validate\_parse\_dates\_presence ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^ self.\_validate\_parse\_dates\_presence(self.names) # type: ignore[has-type] File ""D:\Programs\python\Lib\site-packages\pandas\io\parsers\c\_parser\_wrapper.py"", line 161, in \_\_init\_\_ ~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^ return mapping[engine](f, \*\*self.options) File ""D:\Programs\python\Lib\site-packages\pandas\io\parsers\readers.py"", line 1898, in \_make\_engine ~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^ self.\_engine = self.\_make\_engine(f, self.engine) File ""D:\Programs\python\Lib\site-packages\pandas\io\parsers\readers.py"", line 1620, in \_\_init\_\_ parser = TextFileReader(filepath\_or\_buffer, \*\*kwds) File ""D:\Programs\python\Lib\site-packages\pandas\io\parsers\readers.py"", line 620, in \_read return \_read(filepath\_or\_buffer, kwds) File ""D:\Programs\python\Lib\site-packages\pandas\io\parsers\readers.py"", line 1026, in read\_csv ) parse\_dates=['Date'] # Automatically parse Date column ...<3 lines>... file\_path, df = pd.read\_csv( File ""d:\dev\xlwings\_sqlite\_python\_pandas.py"", line 632, in process\_pandas\_full ret = func(\*args) File ""D:\Programs\python\Lib\site-packages\xlwings\udfs.py"", line 539, in call\_udf res = call\_udf(script, fname, args, this\_workbook, FromVariant(caller)) File ""D:\Programs\python\Lib\site-packages\xlwings\com\_server.py"", line 235, in CallUDF return func(\*args) File ""D:\Programs\python\Lib\site-packages\win32com\server\policy.py"", line 633, in \_invokeex\_ ~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ return S\_OK, -1, self.\_invokeex\_(dispid, lcid, wFlags, args, None, None) File ""D:\Programs\python\Lib\site-packages\win32com\server\policy.py"", line 303, in \_invoke\_ ~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^ return self.\_invoke\_(dispid, lcid, wFlags, args) File ""D:\Programs\python\Lib\site-packages\win32com\server\policy.py"", line 298, in \_Invoke\_"

The error occurs because the 'Date' column name in the CSV file might be different or the column might not be present. Let's modify the code to handle this more robustly:

python

@xw.func

def process\_pandas(file\_path):

# Define column dtypes for efficient reading

dtype\_dict = {

'ProjectSOPO': str,

'Class: Name': str,

'Type': str,

'Document Number': str,

'Name': str,

'Memo': str,

'Account': str,

'Clr': str,

'Split': str,

'Qty': str, # Read as string first, will convert to float later

'Amount': str # Read as string first, will convert to float later

}

# First read the CSV to get the actual column names

df = pd.read\_csv(file\_path, header=0, quotechar='"', dtype=dtype\_dict)

# Clean column names

df.columns = df.columns.str.strip()

# Convert Date column to datetime (assuming the date column exists)

if 'Date' in df.columns:

df['Date'] = pd.to\_datetime(df['Date'], format='%m/%d/%Y', errors='coerce')

# Clean numeric columns (Qty and Amount)

numeric\_cols = ['Qty', 'Amount']

for col in numeric\_cols:

if col in df.columns:

df[col] = (df[col].astype(str)

.replace({

r',': '',

r'\$': '',

r'\(': '-',

r'\)': '',

r'[^\d.-]': '' # Remove any other non-numeric characters

}, regex=True)

.pipe(pd.to\_numeric, errors='coerce')

.astype(float))

# Filter DataFrame

filtered\_df = df[

(~df['Type'].isin(['Purchase Order', 'Sales Order'])) &

(~df['Account'].str.contains('advances|payable', case=False, na=False))

# & (df['Date'] < pd.Timestamp('2024-07-01')) # Uncomment if date filtering is needed

]

# Group and aggregate data

result\_df = (filtered\_df.groupby(['Type', 'ProjectSOPO', 'Account'])

.agg({'Amount': 'sum'})

.reset\_index())

# Calculate Amt column

result\_df['Amt'] = -result\_df['Amount'].where(result\_df['Amount'] > 0, result\_df['Amount'])

result\_df.drop(columns=['Amount'], inplace=True)

# Sort results

result\_df.sort\_values(by=['ProjectSOPO', 'Amt'], inplace=True)

# Convert to output format

output\_data = [result\_df.columns.tolist()] + result\_df.values.tolist()

return output\_data

Key changes made:

Removed parse\_dates parameter from read\_csv

Added date conversion after reading the CSV

Added column existence checks

Read numeric columns as strings first, then convert them

Added error handling for date parsing with errors='coerce'

This version should handle the data more reliably. If you're still getting errors, please check:

The actual column names in your CSV file

The date format in your CSV file

Whether all expected columns are present

You can also add a debug step to print the column names:

python

# Add this after reading the CSV

print("Available columns:", df.columns.tolist())

This will help verify the actual column names in your data.

Why I see only 4 columns in the output instead of all: Type ProjectSOPO Account Amt Invoice 10048 - Macy's Accounts receivable $(2,181,327.12) Bill 10048 - Macy's Total Costs : RG Materials $(865,962.97) Bill 10048 - Macy's Total Costs : RG Subcontractor $(577,887.09) Bill 10048 - Macy's Total Costs : Vendor Tax $(19,519.07) Bill 10048 - Macy's Total Costs : Shipping Cost $(7,690.78) Credit Memo 10048 - Macy's RG Glass - Total Sales : RG Glass Sales $(5,650.00) Bill 10048 - Macy's Total Costs : Engineering/Shop Drawing Services $(4,118.00) Bill 10048 - Macy's Total Costs : RG Energy Surcharge $(3,659.52) Bill 10048 - Macy's Total Costs : RG Project Mgmt Services $(1,000.00) Credit Card 10048 - Macy's Total Costs : RG Materials $(344.10) Bill 10048 - Macy's Total Costs : Hardware & Tools $(226.66)

The output shows only 4 columns because of the groupby operation in the code. Currently, the code is aggregating the data by 'Type', 'ProjectSOPO', and 'Account', and calculating the sum of 'Amount'. If you want to see more columns in the output, you'll need to modify the groupby and aggregation logic. Here's the modified code:

python

@xw.func

def process\_pandas(file\_path):

# Define column dtypes for efficient reading

dtype\_dict = {

'ProjectSOPO': str,

'Class: Name': str,

'Type': str,

'Document Number': str,

'Name': str,

'Memo': str,

'Account': str,

'Clr': str,

'Split': str,

'Qty': str,

'Amount': str

}

# Read the CSV

df = pd.read\_csv(file\_path, header=0, quotechar='"', dtype=dtype\_dict)

# Clean column names

df.columns = df.columns.str.strip()

# Convert Date column to datetime

if 'Date' in df.columns:

df['Date'] = pd.to\_datetime(df['Date'], format='%m/%d/%Y', errors='coerce')

# Clean numeric columns (Qty and Amount)

numeric\_cols = ['Qty', 'Amount']

for col in numeric\_cols:

if col in df.columns:

df[col] = (df[col].astype(str)

.replace({

r',': '',

r'\$': '',

r'\(': '-',

r'\)': '',

r'[^\d.-]': ''

}, regex=True)

.pipe(pd.to\_numeric, errors='coerce')

.astype(float))

# Filter DataFrame

filtered\_df = df[

(~df['Type'].isin(['Purchase Order', 'Sales Order'])) &

(~df['Account'].str.contains('advances|payable', case=False, na=False))

]

# Group and aggregate data with more columns

result\_df = (filtered\_df.groupby([

'Type', 'ProjectSOPO', 'Account', 'Date', 'Document Number',

'Name', 'Memo', 'Clr', 'Split'

]).agg({

'Qty': 'sum',

'Amount': 'sum'

}).reset\_index())

# Calculate Amt column

result\_df['Amt'] = -result\_df['Amount'].where(result\_df['Amount'] > 0, result\_df['Amount'])

# Reorder columns as needed

columns\_order = [

'Type', 'ProjectSOPO', 'Account', 'Date', 'Document Number',

'Name', 'Memo', 'Clr', 'Split', 'Qty', 'Amt'

]

result\_df = result\_df[columns\_order]

# Sort results

result\_df.sort\_values(by=['ProjectSOPO', 'Date', 'Amt'], inplace=True)

# Format date column for output

if 'Date' in result\_df.columns:

result\_df['Date'] = result\_df['Date'].dt.strftime('%m/%d/%Y')

# Format numeric columns

result\_df['Amt'] = result\_df['Amt'].apply(lambda x: f"${x:,.2f}")

result\_df['Qty'] = result\_df['Qty'].apply(lambda x: f"{x:,.2f}" if pd.notnull(x) else "")

# Convert to output format

output\_data = [result\_df.columns.tolist()] + result\_df.values.tolist()

return output\_data

Key changes made:

Added more columns to the groupby operation

Included additional columns in the final output

Added formatting for date and numeric columns

Reordered columns for better readability

Maintained the original data while showing more detail

The output will now include:

Type

ProjectSOPO

Account

Date

Document Number

Name

Memo

Clr

Split

Qty

Amt

The data will be:

Grouped by all these columns

Sorted by ProjectSOPO, Date, and Amt

Formatted for better readability

Show all transaction details instead of just the summaries

Let me know if you want to adjust the column order or formatting!

Don't group and aggregate. Produce data as per csv without any further processing - except change in types as mentioned above dates, qty and amount

Here's the simplified code that only handles data types without any grouping or aggregation:

python

@xw.func

def process\_pandas(file\_path):

# Define column dtypes for efficient reading

dtype\_dict = {

'ProjectSOPO': str,

'Class: Name': str,

'Type': str,

'Document Number': str,

'Name': str,

'Memo': str,

'Account': str,

'Clr': str,

'Split': str,

'Qty': str,

'Amount': str

}

# Read the CSV

df = pd.read\_csv(file\_path, header=0, quotechar='"', dtype=dtype\_dict)

# Clean column names

df.columns = df.columns.str.strip()

# Convert Date column to datetime

if 'Date' in df.columns:

df['Date'] = pd.to\_datetime(df['Date'], format='%m/%d/%Y', errors='coerce')

# Format date back to string in desired format

df['Date'] = df['Date'].dt.strftime('%m/%d/%Y')

# Clean numeric columns (Qty and Amount)

numeric\_cols = ['Qty', 'Amount']

for col in numeric\_cols:

if col in df.columns:

df[col] = (df[col].astype(str)

.replace({

r',': '',

r'\$': '',

r'\(': '-',

r'\)': '',

r'[^\d.-]': ''

}, regex=True)

.pipe(pd.to\_numeric, errors='coerce')

.astype(float))

# Format Amount with currency and commas

if col == 'Amount':

df[col] = df[col].apply(lambda x: f"${x:,.2f}" if pd.notnull(x) else "")

# Format Qty with commas

else:

df[col] = df[col].apply(lambda x: f"{x:,.2f}" if pd.notnull(x) else "")

# Convert to output format

output\_data = [df.columns.tolist()] + df.values.tolist()

return output\_data

This code now:

Reads the CSV file as is

Only converts data types:

Date to datetime and back to formatted string

Qty and Amount to float with proper formatting

Keeps all rows and columns from the original CSV

Does not perform any filtering, grouping, or aggregation

Returns the data in the same order as the input CSV

The output will maintain all columns from your CSV file with just the data type conversions and formatting:

Dates in MM/DD/YYYY format

Amounts formatted as currency with $ and commas

Quantities formatted with decimal places and commas

All other columns remain unchanged

Let me know if you need any adjustments to the formatting or if you want to handle any other columns differently!

Amounts are shown like text not vlues in excel: $151.55 $335.10 $123.63 $122.95 $470.65 $48.38 $470.65 $470.65 $185.95 $207.83 $52.30 $324.53 $100.75 $329.25 $57.78 $123.51 $47.48 $90.01 $329.88 $123.51 $235.45 $235.45 $83.56 $-12,000.00 $-2,000.00 $-17,790.00 $-1,150.00 $-1,349.79 $0.00 $0.00 $0.00 $0.00 $0.00 $0.00 All left aligned and with $sign.

To get the amounts as actual numbers in Excel (right-aligned, without text formatting), we need to modify how we handle the Amount column. Here's the corrected code:

python

@xw.func

def process\_pandas(file\_path):

# Define column dtypes for efficient reading

dtype\_dict = {

'ProjectSOPO': str,

'Class: Name': str,

'Type': str,

'Document Number': str,

'Name': str,

'Memo': str,

'Account': str,

'Clr': str,

'Split': str,

'Qty': str,

'Amount': str

}

# Read the CSV

df = pd.read\_csv(file\_path, header=0, quotechar='"', dtype=dtype\_dict)

# Clean column names

df.columns = df.columns.str.strip()

# Convert Date column to datetime

if 'Date' in df.columns:

df['Date'] = pd.to\_datetime(df['Date'], format='%m/%d/%Y', errors='coerce')

# Format date back to string in desired format

df['Date'] = df['Date'].dt.strftime('%m/%d/%Y')

# Clean numeric columns (Qty and Amount)

numeric\_cols = ['Qty', 'Amount']

for col in numeric\_cols:

if col in df.columns:

df[col] = (df[col].astype(str)

.replace({

r',': '',

r'\$': '',

r'\(': '-',

r'\)': '',

r'[^\d.-]': ''

}, regex=True)

.pipe(pd.to\_numeric, errors='coerce')

.astype(float))

# Don't format the numbers - let Excel handle the formatting

# This will keep them as actual numbers in Excel

# Convert to output format

output\_data = [df.columns.tolist()] + df.values.tolist()

return output\_data

Key changes:

Removed the string formatting for Amount column

Kept the numbers as actual float values

Excel will now display them as proper numbers (right-aligned)

You can then use Excel's number formatting to display the currency symbol if needed

In Excel, you can then:

Select the Amount column

Right-click → Format Cells

Choose "Currency" under Number format

This will properly format the numbers while maintaining them as actual values

This way, the numbers will:

Be right-aligned

Work properly in calculations

Sort correctly

Display as currency when formatted in Excel

Let me know if you need any adjustments!

Dates are output as string s not dates in excel. If they would be dates output will be date numbers in excel.

You're right. To get actual Excel dates, we need to convert the dates to Excel's numeric date format. Here's the corrected code:

python

@xw.func

def process\_pandas(file\_path):

# Define column dtypes for efficient reading

dtype\_dict = {

'ProjectSOPO': str,

'Class: Name': str,

'Type': str,

'Document Number': str,

'Name': str,

'Memo': str,

'Account': str,

'Clr': str,

'Split': str,

'Qty': str,

'Amount': str

}

# Read the CSV

df = pd.read\_csv(file\_path, header=0, quotechar='"', dtype=dtype\_dict)

# Clean column names

df.columns = df.columns.str.strip()

# Convert Date column to datetime and then to Excel date number

if 'Date' in df.columns:

df['Date'] = pd.to\_datetime(df['Date'], format='%m/%d/%Y', errors='coerce')

# Convert to Excel date number (number of days since 1900-01-01)

df['Date'] = (df['Date'] - pd.Timestamp('1899-12-30')).dt.days

# Clean numeric columns (Qty and Amount)

numeric\_cols = ['Qty', 'Amount']

for col in numeric\_cols:

if col in df.columns:

df[col] = (df[col].astype(str)

.replace({

r',': '',

r'\$': '',

r'\(': '-',

r'\)': '',

r'[^\d.-]': ''

}, regex=True)

.pipe(pd.to\_numeric, errors='coerce')

.astype(float))

# Convert to output format

output\_data = [df.columns.tolist()] + df.values.tolist()

return output\_data

Key changes:

Instead of converting dates back to strings, we now convert them to Excel date numbers

Excel uses days since December 30, 1899, as its date system

The dates will now appear as actual dates in Excel and can be formatted as needed

In Excel, you can then:

Select the Date column

Right-click → Format Cells

Choose your preferred date format

The dates will now:

Be actual Excel dates (not strings)

Sort correctly

Work in date calculations

Be formattable using Excel's date formats

Let me know if you need any adjustments!

Please sanitize memo column using regex so that if anywhere in description "\" is seen, it is replaced by " " (Space \s)

Here's the code with memo column sanitization to replace backslashes with spaces:

python

@xw.func

def process\_pandas(file\_path):

# Define column dtypes for efficient reading

dtype\_dict = {

'ProjectSOPO': str,

'Class: Name': str,

'Type': str,

'Document Number': str,

'Name': str,

'Memo': str,

'Account': str,

'Clr': str,

'Split': str,

'Qty': str,

'Amount': str

}

# Read the CSV

df = pd.read\_csv(file\_path, header=0, quotechar='"', dtype=dtype\_dict)

# Clean column names

df.columns = df.columns.str.strip()

# Sanitize Memo column - replace backslashes with spaces

if 'Memo' in df.columns:

df['Memo'] = df['Memo'].str.replace(r'\\', ' ', regex=True)

# Convert Date column to datetime and then to Excel date number

if 'Date' in df.columns:

df['Date'] = pd.to\_datetime(df['Date'], format='%m/%d/%Y', errors='coerce')

# Convert to Excel date number (number of days since 1900-01-01)

df['Date'] = (df['Date'] - pd.Timestamp('1899-12-30')).dt.days

# Clean numeric columns (Qty and Amount)

numeric\_cols = ['Qty', 'Amount']

for col in numeric\_cols:

if col in df.columns:

df[col] = (df[col].astype(str)

.replace({

r',': '',

r'\$': '',

r'\(': '-',

r'\)': '',

r'[^\d.-]': ''

}, regex=True)

.pipe(pd.to\_numeric, errors='coerce')

.astype(float))

# Convert to output format

output\_data = [df.columns.tolist()] + df.values.tolist()

return output\_data

Key changes:

Added sanitization for the Memo column using str.replace()

Used regex pattern r'\' to match backslashes

Replaced backslashes with single spaces

Added this processing before the date and numeric conversions

This will:

Find any backslashes in the Memo column

Replace them with spaces

Handle multiple backslashes if present

Preserve all other characters in the memo text

Let me know if you need any adjustments to the sanitization logic!

Can you optimize this code which runs too slow as there are about 4.5 million records.

Optimized Code for Processing 4.5 Million Records

Here's a significantly optimized version of the code to handle 4.5 million records efficiently:

python

@xw.func

def process\_pandas(file\_path):

# Only specify dtypes for columns we need to process

dtype\_dict = {

'Memo': str,

'Qty': str,

'Amount': str

}

# Use usecols to only read columns we need

needed\_columns = ['Date', 'Memo', 'Qty', 'Amount', 'ProjectSOPO', 'Class: Name',

'Type', 'Document Number', 'Name', 'Account', 'Clr', 'Split']

# Read the CSV with optimizations

df = pd.read\_csv(

file\_path,

header=0,

quotechar='"',

dtype=dtype\_dict,

usecols=needed\_columns,

parse\_dates=['Date'], # Parse dates during reading

infer\_datetime\_format=True,

engine='c', # Use C engine for faster parsing

low\_memory=True,

chunksize=500000 # Process in chunks

)

# Process in chunks

chunks = []

for chunk in df:

# Clean column names

chunk.columns = chunk.columns.str.strip()

# Sanitize Memo column - replace backslashes with spaces (vectorized)

if 'Memo' in chunk.columns:

chunk['Memo'] = chunk['Memo'].str.replace(r'\\', ' ', regex=True)

# Convert Date to Excel date number (vectorized)

if 'Date' in chunk.columns:

chunk['Date'] = (chunk['Date'] - pd.Timestamp('1899-12-30')).dt.days

# Process numeric columns (vectorized operations)

for col in ['Qty', 'Amount']:

if col in chunk.columns:

# Use numpy for faster numeric conversion

values = chunk[col].astype(str)

# Remove currency symbols and formatting

values = values.str.replace(r'[,$\(\)]', '', regex=True)

values = values.str.replace(r'\(', '-', regex=True)

# Convert to numeric

chunk[col] = pd.to\_numeric(values, errors='coerce')

chunks.append(chunk)

# Combine processed chunks

result\_df = pd.concat(chunks, ignore\_index=True)

# Convert to output format

output\_data = [result\_df.columns.tolist()] + result\_df.values.tolist()

return output\_data

Key Optimizations:

Chunked Processing: Processes data in chunks of 500,000 rows to reduce memory usage

Selective Column Reading: Only reads the columns we need

Parse Dates During Reading: Parses dates during CSV reading instead of after

C Engine: Uses the faster C engine for CSV parsing

Vectorized Operations: Uses pandas vectorized operations instead of loops

Simplified Regex: Reduced complexity of regex operations

Low Memory Mode: Enabled to reduce memory footprint

Optimized Data Types: Only specifies dtypes for columns that need special handling

Additional Optimization Options:

If this is still too slow, consider these additional approaches:

python

# Alternative approach using Dask for distributed processing

@xw.func

def process\_pandas\_dask(file\_path):

import dask.dataframe as dd

# Read with Dask

ddf = dd.read\_csv(file\_path)

# Process with Dask (similar operations as above)

if 'Memo' in ddf.columns:

ddf['Memo'] = ddf['Memo'].str.replace(r'\\', ' ', regex=True)

if 'Date' in ddf.columns:

ddf['Date'] = dd.to\_datetime(ddf['Date'])

ddf['Date'] = (ddf['Date'] - pd.Timestamp('1899-12-30')).dt.days

# Process numeric columns

for col in ['Qty', 'Amount']:

if col in ddf.columns:

ddf[col] = ddf[col].str.replace(r'[,$\(\)]', '', regex=True)

ddf[col] = ddf[col].str.replace(r'\(', '-', regex=True)

ddf[col] = dd.to\_numeric(ddf[col], errors='coerce')

# Compute result

result\_df = ddf.compute()

# Convert to output format

output\_data = [result\_df.columns.tolist()] + result\_df.values.tolist()

return output\_data

If you're dealing with extremely large files, you might also consider:

Processing the file in a separate script (not through Excel)

Using multiprocessing to parallelize the work

Using PyArrow or Polars for faster data processing

Saving intermediate results to disk

Let me know if you need further optimizations based on your specific hardware constraints!

Erors - 1st option chunks "ValueError: Usecols do not match columns, columns expected but not found: ['Name', 'Date', 'Memo', 'Qty', 'Split', 'ProjectSOPO', 'Clr', 'Class: Name', 'Account', 'Type', 'Amount', 'Document Number'] ) ...<2 lines>... raise ValueError( File ""D:\Programs\python\Lib\site-packages\pandas\io\parsers\base\_parser.py"", line 979, in \_validate\_usecols\_names ~~~~~~~~~~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^ self.\_validate\_usecols\_names(usecols, self.orig\_names) File ""D:\Programs\python\Lib\site-packages\pandas\io\parsers\c\_parser\_wrapper.py"", line 140, in \_\_init\_\_ ~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^ return mapping[engine](f, \*\*self.options) File ""D:\Programs\python\Lib\site-packages\pandas\io\parsers\readers.py"", line 1898, in \_make\_engine ~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^ self.\_engine = self.\_make\_engine(f, self.engine) File ""D:\Programs\python\Lib\site-packages\pandas\io\parsers\readers.py"", line 1620, in \_\_init\_\_ parser = TextFileReader(filepath\_or\_buffer, \*\*kwds) File ""D:\Programs\python\Lib\site-packages\pandas\io\parsers\readers.py"", line 620, in \_read return \_read(filepath\_or\_buffer, kwds) File ""D:\Programs\python\Lib\site-packages\pandas\io\parsers\readers.py"", line 1026, in read\_csv ) chunksize=500000 # Process in chunks ...<8 lines>... file\_path, df = pd.read\_csv( File ""d:\dev\xlwings\_sqlite\_python\_pandas.py"", line 682, in process\_pandas\_optimized ret = func(\*args) File ""D:\Programs\python\Lib\site-packages\xlwings\udfs.py"", line 539, in call\_udf res = call\_udf(script, fname, args, this\_workbook, FromVariant(caller)) File ""D:\Programs\python\Lib\site-packages\xlwings\com\_server.py"", line 235, in CallUDF return func(\*args) File ""D:\Programs\python\Lib\site-packages\win32com\server\policy.py"", line 633, in \_invokeex\_ ~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ return S\_OK, -1, self.\_invokeex\_(dispid, lcid, wFlags, args, None, None) File ""D:\Programs\python\Lib\site-packages\win32com\server\policy.py"", line 303, in \_invoke\_ ~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^ return self.\_invoke\_(dispid, lcid, wFlags, args) File ""D:\Programs\python\Lib\site-packages\win32com\server\policy.py"", line 298, in \_Invoke\_"

Let's fix the code to handle the column validation issue. The problem is that we're specifying columns that might not exist in the file. Here's a more robust version:

python

@xw.func

def process\_pandas(file\_path):

# First, peek at the file to get actual column names

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

header = f.readline().strip()

# Parse header to get actual column names

actual\_columns = [col.strip('"') for col in header.split(',')]

# Define dtypes for columns that exist

dtype\_dict = {}

for col in ['Memo', 'Qty', 'Amount']:

if col in actual\_columns:

dtype\_dict[col] = str

# Read the CSV without usecols first

df = pd.read\_csv(

file\_path,

header=0,

quotechar='"',

dtype=dtype\_dict,

engine='c', # Use C engine for faster parsing

low\_memory=True

)

# Clean column names

df.columns = df.columns.str.strip()

# Process in chunks to reduce memory usage

chunk\_size = 500000

total\_rows = len(df)

chunks = []

for start\_idx in range(0, total\_rows, chunk\_size):

end\_idx = min(start\_idx + chunk\_size, total\_rows)

chunk = df.iloc[start\_idx:end\_idx].copy()

# Sanitize Memo column - replace backslashes with spaces

if 'Memo' in chunk.columns:

chunk['Memo'] = chunk['Memo'].str.replace(r'\\', ' ', regex=True)

# Convert Date column to datetime and then to Excel date number

if 'Date' in chunk.columns:

chunk['Date'] = pd.to\_datetime(chunk['Date'], errors='coerce')

# Convert to Excel date number

chunk['Date'] = (chunk['Date'] - pd.Timestamp('1899-12-30')).dt.days

# Clean numeric columns

for col in ['Qty', 'Amount']:

if col in chunk.columns:

# Use vectorized string operations

chunk[col] = (chunk[col].astype(str)

.str.replace(r',', '', regex=True)

.str.replace(r'\$', '', regex=True)

.str.replace(r'\(', '-', regex=True)

.str.replace(r'\)', '', regex=True))

# Convert to numeric

chunk[col] = pd.to\_numeric(chunk[col], errors='coerce')

chunks.append(chunk)

# Free memory

del chunk

# Combine processed chunks

result\_df = pd.concat(chunks, ignore\_index=True)

# Convert to output format

output\_data = [result\_df.columns.tolist()] + result\_df.values.tolist()

return output\_data

If you're still experiencing memory issues, here's an even more memory-efficient version that processes the file in chunks from the beginning:

python

@xw.func

def process\_pandas(file\_path):

# First, peek at the file to get actual column names

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

header = f.readline().strip()

# Define dtypes for columns we expect

dtype\_dict = {

'Memo': str,

'Qty': str,

'Amount': str

}

# Read and process in chunks

chunk\_size = 100000

chunks = []

# Use chunked reading from the start

for chunk in pd.read\_csv(

file\_path,

header=0,

quotechar='"',

dtype=dtype\_dict,

chunksize=chunk\_size, # Process in chunks from the beginning

engine='c',

low\_memory=True

):

# Clean column names

chunk.columns = chunk.columns.str.strip()

# Sanitize Memo column - replace backslashes with spaces

if 'Memo' in chunk.columns:

chunk['Memo'] = chunk['Memo'].str.replace(r'\\', ' ', regex=True)

# Convert Date column to datetime and then to Excel date number

if 'Date' in chunk.columns:

chunk['Date'] = pd.to\_datetime(chunk['Date'], errors='coerce')

# Convert to Excel date number

chunk['Date'] = (chunk['Date'] - pd.Timestamp('1899-12-30')).dt.days

# Clean numeric columns

for col in ['Qty', 'Amount']:

if col in chunk.columns:

# Use vectorized string operations

chunk[col] = (chunk[col].astype(str)

.str.replace(r',', '', regex=True)

.str.replace(r'\$', '', regex=True)

.str.replace(r'\(', '-', regex=True)

.str.replace(r'\)', '', regex=True))

# Convert to numeric

chunk[col] = pd.to\_numeric(chunk[col], errors='coerce')

chunks.append(chunk)

# Free memory

del chunk

# Combine processed chunks

result\_df = pd.concat(chunks, ignore\_index=True)

# Convert to output format

output\_data = [result\_df.columns.tolist()] + result\_df.values.tolist()

return output\_data

If you're still having issues, we can try an even more memory-efficient approach that doesn't keep all chunks in memory:

python

@xw.func

def process\_pandas(file\_path):

import tempfile

import os

# Create a temporary file for processed data

temp\_file = tempfile.NamedTemporaryFile(delete=False, suffix='.csv')

temp\_file.close()

# Define dtypes for columns we expect

dtype\_dict = {

'Memo': str,

'Qty': str,

'Amount': str

}

# Process in chunks and write directly to temp file

chunk\_size = 100000

header\_saved = False

# Use chunked reading

for chunk in pd.read\_csv(

file\_path,

header=0,

quotechar='"',

dtype=dtype\_dict,

chunksize=chunk\_size,

engine='c',

low\_memory=True

):

# Clean column names

chunk.columns = chunk.columns.str.strip()

# Sanitize Memo column

if 'Memo' in chunk.columns:

chunk['Memo'] = chunk['Memo'].str.replace(r'\\', ' ', regex=True)

# Convert Date column

if 'Date' in chunk.columns:

chunk['Date'] = pd.to\_datetime(chunk['Date'], errors='coerce')

chunk['Date'] = (chunk['Date'] - pd.Timestamp('1899-12-30')).dt.days

# Clean numeric columns

for col in ['Qty', 'Amount']:

if col in chunk.columns:

chunk[col] = (chunk[col].astype(str)

.str.replace(r',', '', regex=True)

.str.replace(r'\$', '', regex=True)

.str.replace(r'\(', '-', regex=True)

.str.replace(r'\)', '', regex=True))

chunk[col] = pd.to\_numeric(chunk[col], errors='coerce')

# Write to temp file

chunk.to\_csv(temp\_file.name, mode='a', header=not header\_saved, index=False)

header\_saved = True

# Free memory

del chunk

# Read back the processed data

result\_df = pd.read\_csv(temp\_file.name)

# Clean up

try:

os.unlink(temp\_file.name)

except:

pass

# Convert to output format

output\_data = [result\_df.columns.tolist()] + result\_df.values.tolist()

return output\_data

Choose the approach that works best for your specific memory constraints and file size.

These are the columns: Project ,Class: Name ,Type ,Date ,Document Number ,Name ,Memo ,Account ,Clr ,Split ,Qty ,Amount

Thanks for providing the exact column names. Let's create an optimized version specifically for these columns:

python

@xw.func

def process\_pandas(file\_path):

# Define column dtypes for efficient reading

dtype\_dict = {

'Project': str,

'Class: Name': str,

'Type': str,

'Document Number': str,

'Name': str,

'Memo': str,

'Account': str,

'Clr': str,

'Split': str,

'Qty': str,

'Amount': str

}

# Process in chunks to handle 4.5 million records efficiently

chunk\_size = 100000

chunks = []

# Use chunked reading from the start

for chunk in pd.read\_csv(

file\_path,

header=0,

quotechar='"',

dtype=dtype\_dict,

chunksize=chunk\_size,

engine='c',

low\_memory=True

):

# Clean column names

chunk.columns = chunk.columns.str.strip()

# Sanitize Memo column - replace backslashes with spaces

if 'Memo' in chunk.columns:

chunk['Memo'] = chunk['Memo'].str.replace(r'\\', ' ', regex=True)

# Convert Date column to datetime and then to Excel date number

if 'Date' in chunk.columns:

chunk['Date'] = pd.to\_datetime(chunk['Date'], errors='coerce')

# Convert to Excel date number

chunk['Date'] = (chunk['Date'] - pd.Timestamp('1899-12-30')).dt.days

# Clean numeric columns

for col in ['Qty', 'Amount']:

if col in chunk.columns:

# Use vectorized string operations

chunk[col] = (chunk[col].astype(str)

.str.replace(r',', '', regex=True)

.str.replace(r'\$', '', regex=True)

.str.replace(r'\(', '-', regex=True)

.str.replace(r'\)', '', regex=True))

# Convert to numeric

chunk[col] = pd.to\_numeric(chunk[col], errors='coerce')

chunks.append(chunk)

# Combine processed chunks

result\_df = pd.concat(chunks, ignore\_index=True)

# Convert to output format

output\_data = [result\_df.columns.tolist()] + result\_df.values.tolist()

return output\_data

If memory is still an issue, here's an even more memory-efficient version that uses a temporary file to avoid keeping all chunks in memory:

python

@xw.func

def process\_pandas(file\_path):

import tempfile

import os

# Create a temporary file for processed data

temp\_file = tempfile.NamedTemporaryFile(delete=False, suffix='.csv')

temp\_file.close()

# Define column dtypes for efficient reading

dtype\_dict = {

'Project': str,

'Class: Name': str,

'Type': str,

'Document Number': str,

'Name': str,

'Memo': str,

'Account': str,

'Clr': str,

'Split': str,

'Qty': str,

'Amount': str

}

# Process in chunks and write directly to temp file

chunk\_size = 100000

header\_saved = False

try:

# Use chunked reading

for chunk in pd.read\_csv(

file\_path,

header=0,

quotechar='"',

dtype=dtype\_dict,

chunksize=chunk\_size,

engine='c',

low\_memory=True

):

# Clean column names

chunk.columns = chunk.columns.str.strip()

# Sanitize Memo column

if 'Memo' in chunk.columns:

chunk['Memo'] = chunk['Memo'].str.replace(r'\\', ' ', regex=True)

# Convert Date column

if 'Date' in chunk.columns:

chunk['Date'] = pd.to\_datetime(chunk['Date'], errors='coerce')

chunk['Date'] = (chunk['Date'] - pd.Timestamp('1899-12-30')).dt.days

# Clean numeric columns

for col in ['Qty', 'Amount']:

if col in chunk.columns:

chunk[col] = (chunk[col].astype(str)

.str.replace(r',', '', regex=True)

.str.replace(r'\$', '', regex=True)

.str.replace(r'\(', '-', regex=True)

.str.replace(r'\)', '', regex=True))

chunk[col] = pd.to\_numeric(chunk[col], errors='coerce')

# Write to temp file

chunk.to\_csv(temp\_file.name, mode='a', header=not header\_saved, index=False)

header\_saved = True

# Free memory

del chunk

# Read back the processed data

result\_df = pd.read\_csv(temp\_file.name)

# Convert to output format

output\_data = [result\_df.columns.tolist()] + result\_df.values.tolist()

return output\_data

finally:

# Clean up temp file

try:

os.unlink(temp\_file.name)

except:

pass

For even better performance, you could consider:

Using a database like SQLite for intermediate storage

Processing only the columns you actually need to transform

Using parallel processing with the multiprocessing module

Using PyArrow or Polars instead of pandas for faster processing

Let me know if you need any adjustments to these solutions!

Can you transform code using pyarrow? (code with chunks - if pyarrow allows that. I used your following code (Note change in name of UDF - You can change it to pyarrow\_chunks) @xw.func def process\_pandas\_chunks(file\_path): # First, peek at the file to get actual column names with open(file\_path, 'r') as f: header = f.readline().strip() # Define dtypes for columns we expect dtype\_dict = { 'Memo': str, 'Qty': str, 'Amount': str } # Read and process in chunks chunk\_size = 100000 chunks = [] # Use chunked reading from the start for chunk in pd.read\_csv( file\_path, header=0, quotechar='"', dtype=dtype\_dict, chunksize=chunk\_size, # Process in chunks from the beginning engine='c', low\_memory=True ): # Clean column names chunk.columns = chunk.columns.str.strip() # Sanitize Memo column - replace backslashes with spaces if 'Memo' in chunk.columns: chunk['Memo'] = chunk['Memo'].str.replace(r'\\', ' ', regex=True) # Convert Date column to datetime and then to Excel date number if 'Date' in chunk.columns: chunk['Date'] = pd.to\_datetime(chunk['Date'], errors='coerce') # Convert to Excel date number chunk['Date'] = (chunk['Date'] - pd.Timestamp('1899-12-30')).dt.days # Clean numeric columns for col in ['Qty', 'Amount']: if col in chunk.columns: # Use vectorized string operations chunk[col] = (chunk[col].astype(str) .str.replace(r',', '', regex=True) .str.replace(r'\$', '', regex=True) .str.replace(r'\(', '-', regex=True) .str.replace(r'\)', '', regex=True)) # Convert to numeric chunk[col] = pd.to\_numeric(chunk[col], errors='coerce') chunks.append(chunk) # Free memory del chunk # Combine processed chunks result\_df = pd.concat(chunks, ignore\_index=True) # Convert to output format output\_data = [result\_df.columns.tolist()] + result\_df.values.tolist() return output\_data

PyArrow Implementation for Fast CSV Processing

Here's a highly optimized version using PyArrow for processing large CSV files:

python

@xw.func

def process\_pyarrow\_chunks(file\_path):

import pyarrow as pa

import pyarrow.csv as csv

import pyarrow.compute as pc

import pandas as pd

import numpy as np

# Define read options for PyArrow

read\_options = csv.ReadOptions(

block\_size=10 \* 1024 \* 1024, # 10MB chunks

use\_threads=True

)

parse\_options = csv.ParseOptions(

delimiter=',',

quote\_char='"'

)

convert\_options = csv.ConvertOptions(

strings\_can\_be\_null=True,

include\_columns=None # Read all columns

)

# Read the CSV file in batches

batches = []

batch\_size = 500000 # Process in large batches

batch\_count = 0

# Create a reader for the CSV file

with csv.open\_csv(

file\_path,

read\_options=read\_options,

parse\_options=parse\_options,

convert\_options=convert\_options

) as reader:

schema = reader.schema

# Process each batch

for batch in reader:

batch\_count += 1

print(f"Processing batch {batch\_count}")

# Get column indices for processing

memo\_idx = schema.get\_field\_index('Memo') if 'Memo' in schema.names else -1

date\_idx = schema.get\_field\_index('Date') if 'Date' in schema.names else -1

qty\_idx = schema.get\_field\_index('Qty') if 'Qty' in schema.names else -1

amount\_idx = schema.get\_field\_index('Amount') if 'Amount' in schema.names else -1

# Convert to dictionary for column-wise processing

batch\_dict = batch.to\_pydict()

# Process Memo column - replace backslashes with spaces

if memo\_idx >= 0:

memo\_array = batch.column(memo\_idx)

# Replace backslashes with spaces using PyArrow compute functions

if memo\_array.null\_count < len(memo\_array):

# Convert to string array if needed

if not pa.types.is\_string(memo\_array.type):

memo\_array = pc.cast(memo\_array, pa.string())

# Replace backslashes with spaces

memo\_array = pc.replace\_substring\_regex(memo\_array, r'\\', ' ')

batch\_dict['Memo'] = memo\_array.to\_pylist()

# Process Date column

if date\_idx >= 0:

date\_array = batch.column(date\_idx)

# Convert to pandas for date processing

dates = pd.Series(date\_array.to\_pandas())

dates = pd.to\_datetime(dates, errors='coerce')

# Convert to Excel date number

excel\_dates = (dates - pd.Timestamp('1899-12-30')).dt.days

batch\_dict['Date'] = excel\_dates.tolist()

# Process numeric columns

for col\_name, col\_idx in [('Qty', qty\_idx), ('Amount', amount\_idx)]:

if col\_idx >= 0:

# Get column as array

col\_array = batch.column(col\_idx)

# Convert to string for processing

if not pa.types.is\_string(col\_array.type):

col\_array = pc.cast(col\_array, pa.string())

# Clean numeric strings

col\_array = pc.replace\_substring\_regex(col\_array, r',', '')

col\_array = pc.replace\_substring\_regex(col\_array, r'\$', '')

col\_array = pc.replace\_substring\_regex(col\_array, r'\(', '-')

col\_array = pc.replace\_substring\_regex(col\_array, r'\)', '')

# Convert to pandas for numeric conversion

numeric\_values = pd.to\_numeric(

pd.Series(col\_array.to\_pandas()),

errors='coerce'

)

batch\_dict[col\_name] = numeric\_values.tolist()

# Create a new table from the processed dictionary

processed\_batch = pa.Table.from\_pydict(batch\_dict)

batches.append(processed\_batch)

# Free memory

del batch

del batch\_dict

# Combine all batches

if batches:

result\_table = pa.concat\_tables(batches)

# Convert to pandas for output

result\_df = result\_table.to\_pandas()

# Convert to output format

output\_data = [result\_df.columns.tolist()] + result\_df.values.tolist()

return output\_data

else:

return [["No data found"]]

Alternative Implementation with Simpler PyArrow Approach

If the above is too complex, here's a simpler PyArrow implementation that still leverages its speed advantages:

python

@xw.func

def process\_pyarrow\_simple(file\_path):

import pyarrow as pa

import pyarrow.csv as csv

import pandas as pd

# Read the entire CSV file with PyArrow (much faster than pandas)

table = csv.read\_csv(file\_path)

# Convert to pandas for easier manipulation

df = table.to\_pandas()

# Clean column names

df.columns = df.columns.str.strip()

# Process in chunks to reduce memory usage

chunk\_size = 500000

total\_rows = len(df)

chunks = []

for start\_idx in range(0, total\_rows, chunk\_size):

end\_idx = min(start\_idx + chunk\_size, total\_rows)

chunk = df.iloc[start\_idx:end\_idx].copy()

# Sanitize Memo column - replace backslashes with spaces

if 'Memo' in chunk.columns:

chunk['Memo'] = chunk['Memo'].str.replace(r'\\', ' ', regex=True)

# Convert Date column to datetime and then to Excel date number

if 'Date' in chunk.columns:

chunk['Date'] = pd.to\_datetime(chunk['Date'], errors='coerce')

# Convert to Excel date number

chunk['Date'] = (chunk['Date'] - pd.Timestamp('1899-12-30')).dt.days

# Clean numeric columns

for col in ['Qty', 'Amount']:

if col in chunk.columns:

# Use vectorized string operations

chunk[col] = (chunk[col].astype(str)

.str.replace(r',', '', regex=True)

.str.replace(r'\$', '', regex=True)

.str.replace(r'\(', '-', regex=True)

.str.replace(r'\)', '', regex=True))

# Convert to numeric

chunk[col] = pd.to\_numeric(chunk[col], errors='coerce')

chunks.append(chunk)

# Free memory

del chunk

# Combine processed chunks

result\_df = pd.concat(chunks, ignore\_index=True)

# Convert to output format

output\_data = [result\_df.columns.tolist()] + result\_df.values.tolist()

return output\_data

Memory-Efficient PyArrow Implementation

Here's a memory-efficient version that uses PyArrow for reading and processes in batches:

python

@xw.func

def process\_pyarrow\_efficient(file\_path):

import pyarrow as pa

import pyarrow.csv as csv

import pandas as pd

import tempfile

import os

# Create a temporary file for processed data

temp\_file = tempfile.NamedTemporaryFile(delete=False, suffix='.csv')

temp\_file.close()

try:

# Define read options for PyArrow

read\_options = csv.ReadOptions(

block\_size=10 \* 1024 \* 1024, # 10MB chunks

use\_threads=True

)

parse\_options = csv.ParseOptions(

delimiter=',',

quote\_char='"'

)

# Read and process in batches

header\_saved = False

with csv.open\_csv(file\_path, read\_options=read\_options, parse\_options=parse\_options) as reader:

for batch in reader:

# Convert batch to pandas for processing

chunk = batch.to\_pandas()

# Clean column names

chunk.columns = chunk.columns.str.strip()

# Sanitize Memo column - replace backslashes with spaces

if 'Memo' in chunk.columns:

chunk['Memo'] = chunk['Memo'].str.replace(r'\\', ' ', regex=True)

# Convert Date column to datetime and then to Excel date number

if 'Date' in chunk.columns:

chunk['Date'] = pd.to\_datetime(chunk['Date'], errors='coerce')

# Convert to Excel date number

chunk['Date'] = (chunk['Date'] - pd.Timestamp('1899-12-30')).dt.days

# Clean numeric columns

for col in ['Qty', 'Amount']:

if col in chunk.columns:

# Use vectorized string operations

chunk[col] = (chunk[col].astype(str)

.str.replace(r',', '', regex=True)

.str.replace(r'\$', '', regex=True)

.str.replace(r'\(', '-', regex=True)

.str.replace(r'\)', '', regex=True))

# Convert to numeric

chunk[col] = pd.to\_numeric(chunk[col], errors='coerce')

# Write to temp file

chunk.to\_csv(temp\_file.name, mode='a', header=not header\_saved, index=False)

header\_saved = True

# Free memory

del chunk

# Read back the processed data

result\_df = pd.read\_csv(temp\_file.name)

# Convert to output format

output\_data = [result\_df.columns.tolist()] + result\_df.values.tolist()

return output\_data

finally:

# Clean up temp file

try:

os.unlink(temp\_file.name)

except:

pass

Choose the implementation that best fits your needs. The first one is the most optimized but also the most complex, while the third one is the most memory-efficient for extremely large files.

You need to sanitise memo column to replace "\" with space \s " ": Only give me the cleaning command for the following code used @xw.func def process\_pyarrow\_chunks(file\_path): import pyarrow as pa import pyarrow.csv as csv import pyarrow.compute as pc import pandas as pd import numpy as np # Define read options for PyArrow read\_options = csv.ReadOptions( block\_size=10 \* 1024 \* 1024, # 10MB chunks use\_threads=True ) parse\_options = csv.ParseOptions( delimiter=',', quote\_char='"' ) convert\_options = csv.ConvertOptions( strings\_can\_be\_null=True, include\_columns=None # Read all columns ) # Read the CSV file in batches batches = [] batch\_size = 500000 # Process in large batches batch\_count = 0 # Create a reader for the CSV file with csv.open\_csv( file\_path, read\_options=read\_options, parse\_options=parse\_options, convert\_options=convert\_options ) as reader: schema = reader.schema # Process each batch for batch in reader: batch\_count += 1 print(f"Processing batch {batch\_count}") # Get column indices for processing memo\_idx = schema.get\_field\_index('Memo') if 'Memo' in schema.names else -1 date\_idx = schema.get\_field\_index('Date') if 'Date' in schema.names else -1 qty\_idx = schema.get\_field\_index('Qty') if 'Qty' in schema.names else -1 amount\_idx = schema.get\_field\_index('Amount') if 'Amount' in schema.names else -1 # Convert to dictionary for column-wise processing batch\_dict = batch.to\_pydict() # Process Memo column - replace backslashes with spaces if memo\_idx >= 0: memo\_array = batch.column(memo\_idx) # Replace backslashes with spaces using PyArrow compute functions if memo\_array.null\_count < len(memo\_array): # Convert to string array if needed if not pa.types.is\_string(memo\_array.type): memo\_array = pc.cast(memo\_array, pa.string()) # Replace backslashes with spaces memo\_array = pc.replace\_substring\_regex(memo\_array, r'\\', ' ') batch\_dict['Memo'] = memo\_array.to\_pylist() # Process Date column if date\_idx >= 0: date\_array = batch.column(date\_idx) # Convert to pandas for date processing dates = pd.Series(date\_array.to\_pandas()) dates = pd.to\_datetime(dates, errors='coerce') # Convert to Excel date number excel\_dates = (dates - pd.Timestamp('1899-12-30')).dt.days batch\_dict['Date'] = excel\_dates.tolist() # Process numeric columns for col\_name, col\_idx in [('Qty', qty\_idx), ('Amount', amount\_idx)]: if col\_idx >= 0: # Get column as array col\_array = batch.column(col\_idx) # Convert to string for processing if not pa.types.is\_string(col\_array.type): col\_array = pc.cast(col\_array, pa.string()) #Clean Memo Field # Clean numeric strings col\_array = pc.replace\_substring\_regex(col\_array, r',', '') col\_array = pc.replace\_substring\_regex(col\_array, r'\$', '') col\_array = pc.replace\_substring\_regex(col\_array, r'\(', '-') col\_array = pc.replace\_substring\_regex(col\_array, r'\)', '') # Convert to pandas for numeric conversion numeric\_values = pd.to\_numeric( pd.Series(col\_array.to\_pandas()), errors='coerce' ) batch\_dict[col\_name] = numeric\_values.tolist() # Create a new table from the processed dictionary processed\_batch = pa.Table.from\_pydict(batch\_dict) batches.append(processed\_batch) # Free memory del batch del batch\_dict # Combine all batches if batches: result\_table = pa.concat\_tables(batches) # Convert to pandas for output result\_df = result\_table.to\_pandas() # Convert to output format output\_data = [result\_df.columns.tolist()] + result\_df.values.tolist() return output\_data else: return [["No data found"]]

Error creating bedrock response

Generate

To properly sanitize the Memo column to replace backslashes with spaces in PyArrow, here's the corrected section of the code:

python

# Process Memo column - replace backslashes with spaces

if memo\_idx >= 0:

memo\_array = batch.column(memo\_idx)

# Replace backslashes with spaces using PyArrow compute functions

if memo\_array.null\_count < len(memo\_array):

# Convert to string array if needed

if not pa.types.is\_string(memo\_array.type):

memo\_array = pc.cast(memo\_array, pa.string())

# Replace backslashes with spaces - using correct regex pattern

memo\_array = pc.replace\_substring\_regex(memo\_array, pattern='\\\\', replacement=' ')

# Alternative if above doesn't work:

# memo\_array = pc.replace\_substring\_regex(memo\_array, pattern=r'\\', replacement=' ')

batch\_dict['Memo'] = memo\_array.to\_pylist()

If you need more aggressive cleaning, you could add multiple replacements:

python

# Process Memo column with multiple cleaning steps

if memo\_idx >= 0:

memo\_array = batch.column(memo\_idx)

if memo\_array.null\_count < len(memo\_array):

# Convert to string array if needed

if not pa.types.is\_string(memo\_array.type):

memo\_array = pc.cast(memo\_array, pa.string())

# Multiple cleaning steps

memo\_array = pc.replace\_substring\_regex(memo\_array, pattern='\\\\', replacement=' ') # Replace backslashes

memo\_array = pc.replace\_substring\_regex(memo\_array, pattern=r'\s+', replacement=' ') # Replace multiple spaces

memo\_array = pc.strip\_chars(memo\_array) # Trim whitespace

batch\_dict['Memo'] = memo\_array.to\_pylist()

These replacements use PyArrow's compute functions for better performance than pandas string operations.

Where in the code do I insert this?

Insert the Memo cleaning code right after the batch\_dict creation and before the Date processing. Here's the exact location in the code:

python

# Create a reader for the CSV file

with csv.open\_csv(

file\_path,

read\_options=read\_options,

parse\_options=parse\_options,

convert\_options=convert\_options

) as reader:

schema = reader.schema

# Process each batch

for batch in reader:

batch\_count += 1

print(f"Processing batch {batch\_count}")

# Get column indices for processing

memo\_idx = schema.get\_field\_index('Memo') if 'Memo' in schema.names else -1

date\_idx = schema.get\_field\_index('Date') if 'Date' in schema.names else -1

qty\_idx = schema.get\_field\_index('Qty') if 'Qty' in schema.names else -1

amount\_idx = schema.get\_field\_index('Amount') if 'Amount' in schema.names else -1

# Convert to dictionary for column-wise processing

batch\_dict = batch.to\_pydict()

# INSERT HERE - Process Memo column with multiple cleaning steps

if memo\_idx >= 0:

memo\_array = batch.column(memo\_idx)

if memo\_array.null\_count < len(memo\_array):

# Convert to string array if needed

if not pa.types.is\_string(memo\_array.type):

memo\_array = pc.cast(memo\_array, pa.string())

# Multiple cleaning steps

memo\_array = pc.replace\_substring\_regex(memo\_array, pattern='\\\\', replacement=' ') # Replace backslashes

memo\_array = pc.replace\_substring\_regex(memo\_array, pattern=r'\s+', replacement=' ') # Replace multiple spaces

memo\_array = pc.strip\_chars(memo\_array) # Trim whitespace

batch\_dict['Memo'] = memo\_array.to\_pylist()

# Process Date column

if date\_idx >= 0:

date\_array = batch.column(date\_idx)

# ... rest of the code continues

Replace the existing Memo processing section with this new one. It should go right after creating the batch\_dict and before processing the Date column.

Your code has not replaced "\",. For example I find content like this in memo: remake peice 1 10mm Opti-White+Temp. \ .060 Clear PVB \ 10mm 66 3/ 4 113 1/ 4 53.84 SF 23.01 1238.73 1238.73 16142(R) Opti-White+Temp.

Let's modify the Memo cleaning code to specifically handle this case. Here's the updated version:

python

# Process Memo column with enhanced cleaning steps

if memo\_idx >= 0:

memo\_array = batch.column(memo\_idx)

if memo\_array.null\_count < len(memo\_array):

# Convert to string array if needed

if not pa.types.is\_string(memo\_array.type):

memo\_array = pc.cast(memo\_array, pa.string())

# Try different approaches to replace backslashes

# Method 1: Using single backslash pattern

memo\_array = pc.replace\_substring\_regex(memo\_array, pattern='\\\\', replacement=' ')

# Method 2: Using literal backslash

memo\_array = pc.replace\_substring\_regex(memo\_array, pattern='[\\\\]', replacement=' ')

# Method 3: Additional cleanup for any remaining backslashes

memo\_array = pc.replace\_substring\_regex(memo\_array, pattern='\\\\(?=\\s|$)', replacement=' ')

# Clean up multiple spaces and trim

memo\_array = pc.replace\_substring\_regex(memo\_array, pattern=r'\s+', replacement=' ')

memo\_array = pc.strip\_chars(memo\_array)

batch\_dict['Memo'] = memo\_array.to\_pylist()

# For debugging - print a sample of processed memos

sample\_memos = memo\_array.take([0]).to\_pylist()

print("Sample processed memo:", sample\_memos[0] if sample\_memos else "No sample available")

If this still doesn't work, we can try an alternative approach using pandas string operations:

python

# Alternative approach using pandas for stubborn backslashes

if memo\_idx >= 0:

memo\_series = pd.Series(batch.column(memo\_idx).to\_pandas())

if not memo\_series.empty:

# Multiple replacement steps

memo\_series = (memo\_series

.str.replace(r'\\', ' ', regex=True) # Replace backslashes

.str.replace(r'\s+', ' ', regex=True) # Clean up multiple spaces

.str.strip()) # Trim whitespace

batch\_dict['Memo'] = memo\_series.tolist()

# For debugging

print("Sample processed memo:", memo\_series.iloc[0] if not memo\_series.empty else "No sample available")

Try both approaches and see which one works better for your specific case. The second approach using pandas might be more reliable for complex string replacements, though it might be slightly slower.

If you're still seeing backslashes, please let me know and we can try additional patterns or a different approach entirely.

Disturbed my indents: Reproduce below code with proper indentations: @xw.func def process\_pyarrow\_chunks(file\_path): import pyarrow as pa import pyarrow.csv as csv import pyarrow.compute as pc import pandas as pd import numpy as np # Define read options for PyArrow read\_options = csv.ReadOptions( block\_size=10 \* 1024 \* 1024, # 10MB chunks use\_threads=True ) parse\_options = csv.ParseOptions( delimiter=',', quote\_char='"' ) convert\_options = csv.ConvertOptions( strings\_can\_be\_null=True, include\_columns=None # Read all columns ) # Read the CSV file in batches batches = [] batch\_size = 500000 # Process in large batches batch\_count = 0 # Create a reader for the CSV file with csv.open\_csv( file\_path, read\_options=read\_options, parse\_options=parse\_options, convert\_options=convert\_options ) as reader: schema = reader.schema # Process each batch for batch in reader: batch\_count += 1 print(f"Processing batch {batch\_count}") # Get column indices for processing memo\_idx = schema.get\_field\_index('Memo') if 'Memo' in schema.names else -1 date\_idx = schema.get\_field\_index('Date') if 'Date' in schema.names else -1 qty\_idx = schema.get\_field\_index('Qty') if 'Qty' in schema.names else -1 amount\_idx = schema.get\_field\_index('Amount') if 'Amount' in schema.names else -1 # Process Memo column with enhanced cleaning steps if memo\_idx >= 0: memo\_array = batch.column(memo\_idx) if memo\_array.null\_count < len(memo\_array): # Convert to string array if needed if not pa.types.is\_string(memo\_array.type): memo\_array = pc.cast(memo\_array, pa.string()) # Try different approaches to replace backslashes # Method 1: Using single backslash pattern memo\_array = pc.replace\_substring\_regex(memo\_array, pattern='\\\\', replacement=' ') # Method 2: Using literal backslash memo\_array = pc.replace\_substring\_regex(memo\_array, pattern='[\\\\]', replacement=' ') # Method 3: Additional cleanup for any remaining backslashes memo\_array = pc.replace\_substring\_regex(memo\_array, pattern='\\\\(?=\\s|$)', replacement=' ') # Clean up multiple spaces and trim memo\_array = pc.replace\_substring\_regex(memo\_array, pattern=r'\s+', replacement=' ') memo\_array = pc.strip\_chars(memo\_array) batch\_dict['Memo'] = memo\_array.to\_pylist() # For debugging - print a sample of processed memos sample\_memos = memo\_array.take([0]).to\_pylist() print("Sample processed memo:", sample\_memos[0] if sample\_memos else "No sample available") # Process Date column if date\_idx >= 0: date\_array = batch.column(date\_idx) # Convert to pandas for date processing dates = pd.Series(date\_array.to\_pandas()) dates = pd.to\_datetime(dates, errors='coerce') # Convert to Excel date number excel\_dates = (dates - pd.Timestamp('1899-12-30')).dt.days batch\_dict['Date'] = excel\_dates.tolist() # Process numeric columns for col\_name, col\_idx in [('Qty', qty\_idx), ('Amount', amount\_idx)]: if col\_idx >= 0: # Get column as array col\_array = batch.column(col\_idx) # Convert to string for processing if not pa.types.is\_string(col\_array.type): col\_array = pc.cast(col\_array, pa.string()) #Clean Memo Field # Clean numeric strings col\_array = pc.replace\_substring\_regex(col\_array, r',', '') col\_array = pc.replace\_substring\_regex(col\_array, r'\$', '') col\_array = pc.replace\_substring\_regex(col\_array, r'\(', '-') col\_array = pc.replace\_substring\_regex(col\_array, r'\)', '') # Convert to pandas for numeric conversion numeric\_values = pd.to\_numeric( pd.Series(col\_array.to\_pandas()), errors='coerce' ) batch\_dict[col\_name] = numeric\_values.tolist() # Create a new table from the processed dictionary processed\_batch = pa.Table.from\_pydict(batch\_dict) batches.append(processed\_batch) # Free memory del batch del batch\_dict # Combine all batches if batches: result\_table = pa.concat\_tables(batches) # Convert to pandas for output result\_df = result\_table.to\_pandas() # Convert to output format output\_data = [result\_df.columns.tolist()] + result\_df.values.tolist() return output\_data else: return [["No data found"]]

Here's the properly indented code:

python

@xw.func

def process\_pyarrow\_chunks(file\_path):

import pyarrow as pa

import pyarrow.csv as csv

import pyarrow.compute as pc

import pandas as pd

import numpy as np

# Define read options for PyArrow

read\_options = csv.ReadOptions(

block\_size=10 \* 1024 \* 1024, # 10MB chunks

use\_threads=True

)

parse\_options = csv.ParseOptions(

delimiter=',',

quote\_char='"'

)

convert\_options = csv.ConvertOptions(

strings\_can\_be\_null=True,

include\_columns=None # Read all columns

)

# Read the CSV file in batches

batches = []

batch\_size = 500000 # Process in large batches

batch\_count = 0

# Create a reader for the CSV file

with csv.open\_csv(

file\_path,

read\_options=read\_options,

parse\_options=parse\_options,

convert\_options=convert\_options

) as reader:

schema = reader.schema

# Process each batch

for batch in reader:

batch\_count += 1

print(f"Processing batch {batch\_count}")

# Get column indices for processing

memo\_idx = schema.get\_field\_index('Memo') if 'Memo' in schema.names else -1

date\_idx = schema.get\_field\_index('Date') if 'Date' in schema.names else -1

qty\_idx = schema.get\_field\_index('Qty') if 'Qty' in schema.names else -1

amount\_idx = schema.get\_field\_index('Amount') if 'Amount' in schema.names else -1

# Convert to dictionary for column-wise processing

batch\_dict = batch.to\_pydict()

# Process Memo column with enhanced cleaning steps

if memo\_idx >= 0:

memo\_array = batch.column(memo\_idx)

if memo\_array.null\_count < len(memo\_array):

# Convert to string array if needed

if not pa.types.is\_string(memo\_array.type):

memo\_array = pc.cast(memo\_array, pa.string())

# Try different approaches to replace backslashes

# Method 1: Using single backslash pattern

memo\_array = pc.replace\_substring\_regex(memo\_array, pattern='\\\\', replacement=' ')

# Method 2: Using literal backslash

memo\_array = pc.replace\_substring\_regex(memo\_array, pattern='[\\\\]', replacement=' ')

# Method 3: Additional cleanup for any remaining backslashes

memo\_array = pc.replace\_substring\_regex(memo\_array, pattern='\\\\(?=\\s|$)', replacement=' ')

# Clean up multiple spaces and trim

memo\_array = pc.replace\_substring\_regex(memo\_array, pattern=r'\s+', replacement=' ')

memo\_array = pc.strip\_chars(memo\_array)

batch\_dict['Memo'] = memo\_array.to\_pylist()

# For debugging - print a sample of processed memos

sample\_memos = memo\_array.take([0]).to\_pylist()

print("Sample processed memo:", sample\_memos[0] if sample\_memos else "No sample available")

# Process Date column

if date\_idx >= 0:

date\_array = batch.column(date\_idx)

# Convert to pandas for date processing

dates = pd.Series(date\_array.to\_pandas())

dates = pd.to\_datetime(dates, errors='coerce')

# Convert to Excel date number

excel\_dates = (dates - pd.Timestamp('1899-12-30')).dt.days

batch\_dict['Date'] = excel\_dates.tolist()

# Process numeric columns

for col\_name, col\_idx in [('Qty', qty\_idx), ('Amount', amount\_idx)]:

if col\_idx >= 0:

# Get column as array

col\_array = batch.column(col\_idx)

# Convert to string for processing

if not pa.types.is\_string(col\_array.type):

col\_array = pc.cast(col\_array, pa.string())

# Clean numeric strings

col\_array = pc.replace\_substring\_regex(col\_array, r',', '')

col\_array = pc.replace\_substring\_regex(col\_array, r'\$', '')

col\_array = pc.replace\_substring\_regex(col\_array, r'\(', '-')

col\_array = pc.replace\_substring\_regex(col\_array, r'\)', '')

# Convert to pandas for numeric conversion

numeric\_values = pd.to\_numeric(

pd.Series(col\_array.to\_pandas()),

errors='coerce'

)

batch\_dict[col\_name] = numeric\_values.tolist()

# Create a new table from the processed dictionary

processed\_batch = pa.Table.from\_pydict(batch\_dict)

batches.append(processed\_batch)

# Free memory

del batch

del batch\_dict

# Combine all batches

if batches:

result\_table = pa.concat\_tables(batches)

# Convert to pandas for output

result\_df = result\_table.to\_pandas()

# Convert to output format

output\_data = [result\_df.columns.tolist()] + result\_df.values.tolist()

return output\_data

else:

return [["No data found"]]

The main changes were:

Added proper indentation for the Memo processing section

Aligned it with the rest of the batch processing loop

Maintained consistent 4-space indentation throughout the code

Removed duplicate comments and extra spaces

I think this part of replacement is wrong: pattern='\\\\', replacement=' ' You need to check only "\" not multiple "\\\\"

Let's simplify the backslash replacement to target single backslashes. Here's the corrected Memo processing section:

python

# Process Memo column with enhanced cleaning steps

if memo\_idx >= 0:

memo\_array = batch.column(memo\_idx)

if memo\_array.null\_count < len(memo\_array):

# Convert to string array if needed

if not pa.types.is\_string(memo\_array.type):

memo\_array = pc.cast(memo\_array, pa.string())

# Simple single backslash replacement

memo\_array = pc.replace\_substring\_regex(memo\_array, pattern='\\', replacement=' ')

# Clean up multiple spaces and trim

memo\_array = pc.replace\_substring\_regex(memo\_array, pattern=r'\s+', replacement=' ')

memo\_array = pc.strip\_chars(memo\_array)

batch\_dict['Memo'] = memo\_array.to\_pylist()

# For debugging - print a sample of processed memos

sample\_memos = memo\_array.take([0]).to\_pylist()

print("Sample processed memo:", sample\_memos[0] if sample\_memos else "No sample available")

If this still doesn't work, we could try an alternative approach using a single backslash:

python

# Alternative approach

memo\_array = pc.replace\_substring\_regex(memo\_array, pattern='[\\]', replacement=' ')

Or using pandas if PyArrow's regex is being troublesome:

python

# Pandas approach for stubborn backslashes

memo\_series = pd.Series(memo\_array.to\_pandas())

memo\_series = memo\_series.str.replace('\\', ' ', regex=True)

batch\_dict['Memo'] = memo\_series.tolist()

Let me know if any of these approaches works better for your data.

I think this entire cleaning up section date, memo, amount does not clean: for batch in reader: batch\_count += 1 print(f"Processing batch {batch\_count}") # Get column indices for processing memo\_idx = schema.get\_field\_index('Memo') if 'Memo' in schema.names else -1 date\_idx = schema.get\_field\_index('Date') if 'Date' in schema.names else -1 qty\_idx = schema.get\_field\_index('Qty') if 'Qty' in schema.names else -1 amount\_idx = schema.get\_field\_index('Amount') if 'Amount' in schema.names else -1 # Convert to dictionary for column-wise processing batch\_dict = batch.to\_pydict() # Process Memo column with enhanced cleaning steps if memo\_idx >= 0: memo\_array = batch.column(memo\_idx) if memo\_array.null\_count < len(memo\_array): # Convert to string array if needed if not pa.types.is\_string(memo\_array.type): memo\_array = pc.cast(memo\_array, pa.string()) # Try different approaches to replace backslashes # Method 1: Using single backslash pattern memo\_array = pc.replace\_substring\_regex(memo\_array, pattern='\\', replacement=' ') print("Replaced backslash") # Clean up multiple spaces and trim memo\_array = pc.replace\_substring\_regex(memo\_array, pattern=r'\s+', replacement=' ') memo\_array = pc.strip\_chars(memo\_array) batch\_dict['Memo'] = memo\_array.to\_pylist() # For debugging - print a sample of processed memos sample\_memos = memo\_array.take([0]).to\_pylist() print("Sample processed memo:", sample\_memos[0] if sample\_memos else "No sample available") # Process Date column if date\_idx >= 0: date\_array = batch.column(date\_idx) # Convert to pandas for date processing dates = pd.Series(date\_array.to\_pandas()) dates = pd.to\_datetime(dates, errors='coerce') # Convert to Excel date number excel\_dates = (dates - pd.Timestamp('1899-12-30')).dt.days batch\_dict['Date'] = excel\_dates.tolist() # Process numeric columns for col\_name, col\_idx in [('Qty', qty\_idx), ('Amount', amount\_idx)]: if col\_idx >= 0: # Get column as array col\_array = batch.column(col\_idx) # Convert to string for processing if not pa.types.is\_string(col\_array.type): col\_array = pc.cast(col\_array, pa.string()) # Clean numeric strings col\_array = pc.replace\_substring\_regex(col\_array, r',', '') col\_array = pc.replace\_substring\_regex(col\_array, r'\$', '') col\_array = pc.replace\_substring\_regex(col\_array, r'\(', '-') col\_array = pc.replace\_substring\_regex(col\_array, r'\)', '') # Convert to pandas for numeric conversion numeric\_values = pd.to\_numeric( pd.Series(col\_array.to\_pandas()), errors='coerce' ) batch\_dict[col\_name] = numeric\_values.tolist()

python

@xw.func

def process\_pyarrow\_chunks(file\_path):

import pyarrow as pa

import pyarrow.csv as csv

import pyarrow.compute as pc

import pandas as pd

import numpy as np

# Define read options for PyArrow

read\_options = csv.ReadOptions(

block\_size=10 \* 1024 \* 1024, # 10MB chunks

use\_threads=True

)

parse\_options = csv.ParseOptions(

delimiter=',',

quote\_char='"'

)

convert\_options = csv.ConvertOptions(

strings\_can\_be\_null=True,

include\_columns=None # Read all columns

)

# Read the CSV file in batches

batches = []

batch\_size = 500000 # Process in large batches

batch\_count = 0

# Create a reader for the CSV file

with csv.open\_csv(

file\_path,

read\_options=read\_options,

parse\_options=parse\_options,

convert\_options=convert\_options

) as reader:

schema = reader.schema

# Process each batch

for batch in reader:

batch\_count += 1

print(f"Processing batch {batch\_count}")

# Get column indices for processing

memo\_idx = schema.get\_field\_index('Memo') if 'Memo' in schema.names else -1

date\_idx = schema.get\_field\_index('Date') if 'Date' in schema.names else -1

qty\_idx = schema.get\_field\_index('Qty') if 'Qty' in schema.names else -1

amount\_idx = schema.get\_field\_index('Amount') if 'Amount' in schema.names else -1

# Convert to dictionary for column-wise processing

batch\_dict = batch.to\_pydict()

# Process Memo column with enhanced cleaning steps

if memo\_idx >= 0:

memo\_array = batch.column(memo\_idx)

if memo\_array.null\_count < len(memo\_array):

# Convert to string array if needed

if not pa.types.is\_string(memo\_array.type):

memo\_array = pc.cast(memo\_array, pa.string())

# Replace backslashes with spaces

memo\_array = pc.replace\_substring(memo\_array, '\\', ' ')

# Clean up multiple spaces and trim

memo\_array = pc.replace\_substring\_regex(memo\_array, pattern=r'\s+', replacement=' ')

memo\_array = pc.strip\_chars(memo\_array)

batch\_dict['Memo'] = memo\_array.to\_pylist()

# For debugging - print a sample of processed memos

sample\_memos = memo\_array.take([0]).to\_pylist()

print("Sample processed memo:", sample\_memos[0] if sample\_memos else "No sample available")

# Process Date column

if date\_idx >= 0:

date\_array = batch.column(date\_idx)

# Convert to pandas for date processing

dates = pd.Series(date\_array.to\_pandas())

dates = pd.to\_datetime(dates, errors='coerce')

# Convert to Excel date number

excel\_dates = (dates - pd.Timestamp('1899-12-30')).dt.days

batch\_dict['Date'] = excel\_dates.tolist()

# Process numeric columns

for col\_name, col\_idx in [('Qty', qty\_idx), ('Amount', amount\_idx)]:

if col\_idx >= 0:

# Get column as array

col\_array = batch.column(col\_idx)

# Convert to string for processing

if not pa.types.is\_string(col\_array.type):

col\_array = pc.cast(col\_array, pa.string())

# Clean numeric strings

col\_array = pc.replace\_substring(col\_array, ',', '')

col\_array = pc.replace\_substring(col\_array, '$', '')

col\_array = pc.replace\_substring(col\_array, '(', '-')

col\_array = pc.replace\_substring(col\_array, ')', '')

# Convert to pandas for numeric conversion

numeric\_values = pd.to\_numeric(

pd.Series(col\_array.to\_pandas()),

errors='coerce'

)

batch\_dict[col\_name] = numeric\_values.tolist()

# Create a new table from the processed dictionary

processed\_batch = pa.Table.from\_pydict(batch\_dict)

batches.append(processed\_batch)

# Free memory

del batch

del batch\_dict

# Combine all batches

if batches:

result\_table = pa.concat\_tables(batches)

# Convert to pandas for output

result\_df = result\_table.to\_pandas()

# Convert to output format

output\_data = [result\_df.columns.tolist()] + result\_df.values.tolist()

return output\_data

else:

return [["No data found"]]

I notice you've switched from pc.replace\_substring\_regex() to pc.replace\_substring() in your updated code. This is a good change for simple string replacements. However, if the cleaning isn't working, let's try using pandas string methods which are generally more reliable for string cleaning. Here's the modified cleaning section:

python

# Process each batch

for batch in reader:

batch\_count += 1

print(f"Processing batch {batch\_count}")

# Convert batch to pandas DataFrame for easier processing

df = batch.to\_pandas()

# Process Memo column

if 'Memo' in df.columns:

df['Memo'] = (df['Memo']

.astype(str)

.str.replace('\\', ' ', regex=False) # Replace backslashes

.str.replace(r'\s+', ' ', regex=True) # Clean multiple spaces

.str.strip()) # Trim whitespace

print("Sample Memo after cleaning:", df['Memo'].iloc[0] if len(df) > 0 else "No data")

# Process Date column

if 'Date' in df.columns:

df['Date'] = pd.to\_datetime(df['Date'], errors='coerce')

df['Date'] = (df['Date'] - pd.Timestamp('1899-12-30')).dt.days

# Process Qty column

if 'Qty' in df.columns:

df['Qty'] = (df['Qty']

.astype(str)

.str.replace(',', '')

.str.replace('$', '')

.str.replace('(', '-')

.str.replace(')', ''))

df['Qty'] = pd.to\_numeric(df['Qty'], errors='coerce')

# Process Amount column

if 'Amount' in df.columns:

df['Amount'] = (df['Amount']

.astype(str)

.str.replace(',', '')

.str.replace('$', '')

.str.replace('(', '-')

.str.replace(')', ''))

df['Amount'] = pd.to\_numeric(df['Amount'], errors='coerce')

# Convert back to PyArrow Table

processed\_batch = pa.Table.from\_pandas(df)

batches.append(processed\_batch)

# Free memory

del df

del batch

Key changes:

Converted the batch to pandas DataFrame first

Used pandas string methods for cleaning

Simplified the cleaning process

Added more explicit type conversions

Used regex=False for simple replacements

Added debug print for Memo cleaning

This approach should be more reliable because:

Pandas string methods are more forgiving with different data types

The cleaning operations are more straightforward

We can see the immediate results of the cleaning with the debug print

Type conversions are more explicit

Let me know if this works better for your data.

But why pyarrow fails and we want to switch to pandas?

PyArrow doesn't necessarily fail - it's just that its string manipulation functions can be less flexible and more strict compared to pandas. Here are the specific reasons:

PyArrow's String Operations:

pc.replace\_substring\_regex() and pc.replace\_substring() are lower-level operations that can be more sensitive to exact patterns

They require precise type matching and can be less forgiving with mixed data types

The regex engine in PyArrow is more strict in its pattern matching

Debugging Challenges:

python

# PyArrow approach

memo\_array = pc.replace\_substring\_regex(memo\_array, pattern='\\', replacement=' ')

# Hard to inspect intermediate results

We could make PyArrow work by being more explicit and adding better error handling:

python

# More robust PyArrow approach

if memo\_idx >= 0:

memo\_array = batch.column(memo\_idx)

try:

# Ensure we're working with strings

memo\_array = pc.cast(memo\_array, pa.string())

# Print original value for debugging

print("Before cleaning:", memo\_array.take([0]).to\_pylist())

# Simple replacement without regex

memo\_array = pc.replace\_substring(memo\_array, pattern='\\', replacement=' ')

print("After backslash replacement:", memo\_array.take([0]).to\_pylist())

# Clean spaces

memo\_array = pc.replace\_substring\_regex(memo\_array, pattern=r'\s+', replacement=' ')

memo\_array = pc.strip\_chars(memo\_array)

print("After space cleaning:", memo\_array.take([0]).to\_pylist())

batch\_dict['Memo'] = memo\_array.to\_pylist()

except Exception as e:

print(f"Error processing Memo: {e}")

# Fallback to pandas if needed

memo\_series = pd.Series(memo\_array.to\_pandas())

memo\_series = memo\_series.str.replace('\\', ' ').str.strip()

batch\_dict['Memo'] = memo\_series.tolist()

Benefits of staying with PyArrow:

Better memory efficiency

Faster processing for large datasets

Consistency with the rest of the PyArrow pipeline

Avoid unnecessary conversions between PyArrow and pandas

If you want to optimize the PyArrow approach, we could:

Add better error handling

Include debugging statements

Use simpler string operations

Maintain type consistency

Would you like to try an optimized PyArrow version instead of switching to pandas?

Give me complete code with pandas alternative of replacements in memo, date and amount columns

Here's the complete code using pandas for the cleaning operations while maintaining the PyArrow structure:

python

@xw.func

def process\_pyarrow\_chunks(file\_path):

import pyarrow as pa

import pyarrow.csv as csv

import pyarrow.compute as pc

import pandas as pd

import numpy as np

# Define read options for PyArrow

read\_options = csv.ReadOptions(

block\_size=10 \* 1024 \* 1024, # 10MB chunks

use\_threads=True

)

parse\_options = csv.ParseOptions(

delimiter=',',

quote\_char='"'

)

convert\_options = csv.ConvertOptions(

strings\_can\_be\_null=True,

include\_columns=None # Read all columns

)

# Read the CSV file in batches

batches = []

batch\_size = 500000 # Process in large batches

batch\_count = 0

# Create a reader for the CSV file

with csv.open\_csv(

file\_path,

read\_options=read\_options,

parse\_options=parse\_options,

convert\_options=convert\_options

) as reader:

schema = reader.schema

# Process each batch

for batch in reader:

batch\_count += 1

print(f"Processing batch {batch\_count}")

# Convert batch to pandas DataFrame for processing

df = batch.to\_pandas()

# Process Memo column

if 'Memo' in df.columns:

try:

# Convert to string and clean

df['Memo'] = (df['Memo']

.fillna('')

.astype(str)

.str.replace('\\', ' ', regex=False) # Replace backslashes

.str.replace(r'\s+', ' ', regex=True) # Clean multiple spaces

.str.strip()) # Remove leading/trailing spaces

print(f"Sample Memo after cleaning: {df['Memo'].iloc[0] if len(df) > 0 else 'No data'}")

except Exception as e:

print(f"Error processing Memo: {e}")

# Process Date column

if 'Date' in df.columns:

try:

df['Date'] = pd.to\_datetime(df['Date'], errors='coerce')

df['Date'] = (df['Date'] - pd.Timestamp('1899-12-30')).dt.days

print(f"Sample Date after conversion: {df['Date'].iloc[0] if len(df) > 0 else 'No data'}")

except Exception as e:

print(f"Error processing Date: {e}")

# Process Amount column

if 'Amount' in df.columns:

try:

df['Amount'] = (df['Amount']

.fillna('')

.astype(str)

.str.replace(',', '', regex=False)

.str.replace('$', '', regex=False)

.str.replace('(', '-', regex=False)

.str.replace(')', '', regex=False)

.str.strip())

df['Amount'] = pd.to\_numeric(df['Amount'], errors='coerce')

print(f"Sample Amount after cleaning: {df['Amount'].iloc[0] if len(df) > 0 else 'No data'}")

except Exception as e:

print(f"Error processing Amount: {e}")

# Process Qty column

if 'Qty' in df.columns:

try:

df['Qty'] = (df['Qty']

.fillna('')

.astype(str)

.str.replace(',', '', regex=False)

.str.replace('$', '', regex=False)

.str.replace('(', '-', regex=False)

.str.replace(')', '', regex=False)

.str.strip())

df['Qty'] = pd.to\_numeric(df['Qty'], errors='coerce')

print(f"Sample Qty after cleaning: {df['Qty'].iloc[0] if len(df) > 0 else 'No data'}")

except Exception as e:

print(f"Error processing Qty: {e}")

# Convert back to PyArrow Table and append to batches

try:

processed\_batch = pa.Table.from\_pandas(df)

batches.append(processed\_batch)

except Exception as e:

print(f"Error converting back to PyArrow: {e}")

# Free memory

del df

del batch

# Combine all batches

if batches:

try:

result\_table = pa.concat\_tables(batches)

result\_df = result\_table.to\_pandas()

output\_data = [result\_df.columns.tolist()] + result\_df.values.tolist()

return output\_data

except Exception as e:

print(f"Error in final processing: {e}")

return [["Error processing data"]]

else:

return [["No data found"]]

Key changes and improvements:

Converted to pandas DataFrame for each batch

Added proper error handling for each column processing

Used pandas string methods for cleaner string operations

Added debugging prints for each column

Used fillna('') to handle null values

Disabled regex for simple replacements to improve performance

Added proper type conversions

Maintained the overall PyArrow batch processing structure

The main advantages of this approach:

More reliable string operations

Better handling of null values

More intuitive data cleaning operations

Better error handling and debugging

Still maintains the memory efficiency of PyArrow's batch processing

Let me know if you need any clarification or adjustments to the code.

Even that failed to clean up date, memo, etc columns. Why?

The cleaning might be failing for several reasons. Let's debug step by step:

First, let's check what data we're actually receiving:

python

@xw.func

def process\_pyarrow\_chunks(file\_path):

import pyarrow as pa

import pyarrow.csv as csv

import pyarrow.compute as pc

import pandas as pd

import numpy as np

# ... (previous read options setup)

with csv.open\_csv(file\_path, read\_options=read\_options,

parse\_options=parse\_options,

convert\_options=convert\_options) as reader:

schema = reader.schema

print("Schema:", schema) # Debug schema

for batch in reader:

batch\_count += 1

print(f"\nProcessing batch {batch\_count}")

# Convert batch to pandas DataFrame

df = batch.to\_pandas()

# Debug: Print raw data samples

print("\nRaw data samples:")

for col in df.columns:

print(f"\n{col} (dtype: {df[col].dtype}):")

print(df[col].head())

# Process Memo

if 'Memo' in df.columns:

print("\nMemo cleaning:")

print("Before:", df['Memo'].head())

df['Memo'] = (df['Memo']

.fillna('')

.astype(str)

.replace('\\', ' ', regex=False)

.str.replace(r'\s+', ' ', regex=True)

.str.strip())

print("After:", df['Memo'].head())

# Process Date

if 'Date' in df.columns:

print("\nDate cleaning:")

print("Before:", df['Date'].head())

try:

# First check if dates are already in numeric format

if pd.api.types.is\_numeric\_dtype(df['Date']):

print("Dates are already numeric")

else:

df['Date'] = pd.to\_datetime(df['Date'], errors='coerce')

df['Date'] = (df['Date'] - pd.Timestamp('1899-12-30')).dt.days

print("After:", df['Date'].head())

except Exception as e:

print(f"Date conversion error: {e}")

# Process Amount

if 'Amount' in df.columns:

print("\nAmount cleaning:")

print("Before:", df['Amount'].head())

try:

# Check if already numeric

if not pd.api.types.is\_numeric\_dtype(df['Amount']):

df['Amount'] = (df['Amount']

.fillna('')

.astype(str)

.str.replace(',', '')

.str.replace('$', '')

.str.replace('(', '-')

.str.replace(')', '')

.str.strip())

df['Amount'] = pd.to\_numeric(df['Amount'], errors='coerce')

print("After:", df['Amount'].head())

except Exception as e:

print(f"Amount conversion error: {e}")

# Convert back to PyArrow and append

processed\_batch = pa.Table.from\_pandas(df)

batches.append(processed\_batch)

# Only process first batch for debugging

break

# Return first few rows for debugging

if batches:

result\_table = pa.concat\_tables(batches)

result\_df = result\_table.to\_pandas()

print("\nFinal data sample:")

print(result\_df.head())

return [result\_df.columns.tolist()] + result\_df.head().values.tolist()

return [["No data found"]]

This debugging version will help us understand:

What the input schema looks like

What data types we're receiving

What the data looks like before and after each cleaning step

Where exactly the cleaning is failing

Could you run this version and share the debug output? This will help us identify:

If the data is being read correctly

If the data types are what we expect

If the cleaning operations are actually being applied

If there are any specific errors in the conversion process

The most common issues could be:

Data is already in the correct format (no cleaning needed)

Data types are different than expected

The cleaning operations aren't matching the actual data format

The conversions are failing silently

Once we see the debug output, we can adjust the cleaning operations accordingly.

pythoncom error: Python error invoking COM method. Traceback (most recent call last): File "D:\Programs\python\Lib\site-packages\win32com\server\policy.py", line 298, in \_Invoke\_ return self.\_invoke\_(dispid, lcid, wFlags, args) ~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^ File "D:\Programs\python\Lib\site-packages\win32com\server\policy.py", line 303, in \_invoke\_ return S\_OK, -1, self.\_invokeex\_(dispid, lcid, wFlags, args, None, None) ~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ File "D:\Programs\python\Lib\site-packages\win32com\server\policy.py", line 633, in \_invokeex\_ return func(\*args) File "D:\Programs\python\Lib\site-packages\xlwings\com\_server.py", line 235, in CallUDF res = call\_udf(script, fname, args, this\_workbook, FromVariant(caller)) File "D:\Programs\python\Lib\site-packages\xlwings\udfs.py", line 539, in call\_udf ret = func(\*args) File "d:\dev\xlwings\_sqlite\_python\_pandas.py", line 679, in process\_pyarrow\_chunks with csv.open\_csv(file\_path, read\_options=read\_options, ^^^^^^^^^^^^ NameError: name 'read\_options' is not defined pythoncom error: Python error invoking COM method.

I apologize for the error. The code was missing the options initialization. Here's the complete, corrected version:

python

@xw.func

def process\_pyarrow\_chunks(file\_path):

import pyarrow as pa

import pyarrow.csv as csv

import pyarrow.compute as pc

import pandas as pd

import numpy as np

try:

# Define read options for PyArrow

read\_options = csv.ReadOptions(

block\_size=10 \* 1024 \* 1024, # 10MB chunks

use\_threads=True

)

parse\_options = csv.ParseOptions(

delimiter=',',

quote\_char='"'

)

convert\_options = csv.ConvertOptions(

strings\_can\_be\_null=True,

include\_columns=None # Read all columns

)

print("Opening file:", file\_path)

with csv.open\_csv(

file\_path,

read\_options=read\_options,

parse\_options=parse\_options,

convert\_options=convert\_options

) as reader:

schema = reader.schema

print("Schema:", schema) # Debug schema

# Initialize batches list

batches = []

batch\_count = 0

for batch in reader:

batch\_count += 1

print(f"\nProcessing batch {batch\_count}")

# Convert batch to pandas DataFrame

df = batch.to\_pandas()

# Debug: Print raw data samples

print("\nRaw data samples:")

for col in df.columns:

print(f"\n{col} (dtype: {df[col].dtype}):")

print(df[col].head())

# Process Memo

if 'Memo' in df.columns:

print("\nMemo cleaning:")

print("Before:", df['Memo'].head())

df['Memo'] = (df['Memo']

.fillna('')

.astype(str)

.replace('\\', ' ', regex=False)

.str.replace(r'\s+', ' ', regex=True)

.str.strip())

print("After:", df['Memo'].head())

# Process Date

if 'Date' in df.columns:

print("\nDate cleaning:")

print("Before:", df['Date'].head())

try:

if pd.api.types.is\_numeric\_dtype(df['Date']):

print("Dates are already numeric")

else:

df['Date'] = pd.to\_datetime(df['Date'], errors='coerce')

df['Date'] = (df['Date'] - pd.Timestamp('1899-12-30')).dt.days

print("After:", df['Date'].head())

except Exception as e:

print(f"Date conversion error: {e}")

# Process Amount

if 'Amount' in df.columns:

print("\nAmount cleaning:")

print("Before:", df['Amount'].head())

try:

if not pd.api.types.is\_numeric\_dtype(df['Amount']):

df['Amount'] = (df['Amount']

.fillna('')

.astype(str)

.str.replace(',', '')

.str.replace('$', '')

.str.replace('(', '-')

.str.replace(')', '')

.str.strip())

df['Amount'] = pd.to\_numeric(df['Amount'], errors='coerce')

print("After:", df['Amount'].head())

except Exception as e:

print(f"Amount conversion error: {e}")

# Convert back to PyArrow and append

processed\_batch = pa.Table.from\_pandas(df)

batches.append(processed\_batch)

# Only process first batch for debugging

break

# Return first few rows for debugging

if batches:

result\_table = pa.concat\_tables(batches)

result\_df = result\_table.to\_pandas()

print("\nFinal data sample:")

print(result\_df.head())

return [result\_df.columns.tolist()] + result\_df.head().values.tolist()

return [["No data found"]]

except Exception as e:

print(f"Error in processing: {e}")

return [["Error: " + str(e)]]

Key changes:

Added proper initialization of all PyArrow CSV options

Added more comprehensive error handling

Added more debug print statements

Wrapped everything in a try-except block

This version should:

Show us exactly what data we're receiving

Show the data types of each column

Show before and after states for each cleaning operation

Help identify where any failures might be occurring

Run this version and let me know what the debug output shows. This will help us identify exactly where and why the cleaning operations might be failing.

Output of sample data: Opening file: Z:\\Private\\dhawal\\tr\_data.csv Schema: Project : string Class: Name : string Type : string Date : string Document Number : string Name : string Memo : string Account : string Clr : string Split : string Qty : string Amount : string Processing batch 1 Raw data samples: Project (dtype: object): 0 None 1 None 2 None 3 None 4 None Name: Project , dtype: object Class: Name (dtype: object): 0 Carvart Solutions DBA RG Glass 1 Carvart Solutions DBA RG Glass 2 Carvart Solutions DBA RG Glass 3 Carvart Solutions DBA RG Glass 4 Carvart Solutions DBA RG Glass Name: Class: Name , dtype: object Type (dtype: object): 0 Sales Order 1 Sales Order 2 Sales Order 3 Sales Order 4 Sales Order Name: Type , dtype: object Date (dtype: object): 0 1/3/2017 1 1/3/2017 2 1/3/2017 3 1/4/2017 4 1/4/2017 Name: Date , dtype: object Document Number (dtype: object): 0 4308 1 20803 2 4307 3 15515 4 20804 Name: Document Number , dtype: object Name (dtype: object): 0 Sample Order Customer 1 Sample Order Customer 2 Kohn Pedersen Fox Associates PC 3 Gensler - Chicago 4 Sample Order Customer Name: Name , dtype: object Memo (dtype: object): 0 None 1 None 2 None 3 None 4 None Name: Memo , dtype: object Account (dtype: object): 0 Architectural Glass - Total Sales : AG Samples 1 Architectural Glass - Total Sales : AG Samples 2 Architectural Glass - Total Sales : AG Samples 3 Architectural Glass - Total Sales : AG Samples 4 Architectural Glass - Total Sales : AG Samples Name: Account , dtype: object Clr (dtype: object): 0 F 1 F 2 F 3 F 4 F Name: Clr , dtype: object Split (dtype: object): 0 - Sales Orders 1 - Sales Orders 2 - Sales Orders 3 - Sales Orders 4 - Sales Orders Name: Split , dtype: object Qty (dtype: object): 0 -1 1 -1 2 -1 3 -3 4 -3 Name: Qty , dtype: object Amount (dtype: object): 0 $0.00 1 $0.00 2 $0.00 3 $0.00 4 $0.00 Name: Amount , dtype: object Final data sample: Project Class: Name Type Date ... Clr Split Qty Amount 0 None Carvart Solutions DBA RG Glass Sales Order 1/3/2017 ... F - Sales Orders -1 $0.00 1 None Carvart Solutions DBA RG Glass Sales Order 1/3/2017 ... F - Sales Orders -1 $0.00 2 None Carvart Solutions DBA RG Glass Sales Order 1/3/2017 ... F - Sales Orders -1 $0.00 3 None Carvart Solutions DBA RG Glass Sales Order 1/4/2017 ... F - Sales Orders -3 $0.00 4 None Carvart Solutions DBA RG Glass Sales Order 1/4/2017 ... F - Sales Orders -3 $0.00 [5 rows x 12 columns]

Now I can see the exact data structure and why some cleanings might not be working. Let's modify the code to properly handle this data:

python

@xw.func

def process\_pyarrow\_chunks(file\_path):

import pyarrow as pa

import pyarrow.csv as csv

import pyarrow.compute as pc

import pandas as pd

import numpy as np

try:

# Define read options for PyArrow

read\_options = csv.ReadOptions(

block\_size=10 \* 1024 \* 1024,

use\_threads=True

)

parse\_options = csv.ParseOptions(

delimiter=',',

quote\_char='"'

)

convert\_options = csv.ConvertOptions(

strings\_can\_be\_null=True,

include\_columns=None

)

batches = []

batch\_count = 0

with csv.open\_csv(

file\_path,

read\_options=read\_options,

parse\_options=parse\_options,

convert\_options=convert\_options

) as reader:

for batch in reader:

batch\_count += 1

df = batch.to\_pandas()

# Clean column names - remove extra spaces

df.columns = df.columns.str.strip()

# Process Memo (appears to be all None in sample)

if 'Memo' in df.columns:

df['Memo'] = df['Memo'].fillna('').astype(str)

df.loc[df['Memo'] == 'None', 'Memo'] = ''

df['Memo'] = (df['Memo']

.str.replace('\\', ' ', regex=False)

.str.replace(r'\s+', ' ', regex=True)

.str.strip())

# Process Date (format: M/D/YYYY)

if 'Date' in df.columns:

try:

df['Date'] = pd.to\_datetime(df['Date'], format='%m/%d/%Y', errors='coerce')

df['Date'] = (df['Date'] - pd.Timestamp('1899-12-30')).dt.days

except Exception as e:

print(f"Date conversion error: {e}")

# Process Amount (format: $0.00)

if 'Amount' in df.columns:

df['Amount'] = (df['Amount']

.fillna('$0.00')

.astype(str)

.str.replace('$', '', regex=False)

.str.replace(',', '', regex=False)

.str.replace('(', '-', regex=False)

.str.replace(')', '', regex=False)

.str.strip())

df['Amount'] = pd.to\_numeric(df['Amount'], errors='coerce')

# Process Qty (already numeric but as string)

if 'Qty' in df.columns:

df['Qty'] = pd.to\_numeric(df['Qty'], errors='coerce')

# Handle None values in other columns

for col in df.columns:

if col not in ['Date', 'Amount', 'Qty']:

df[col] = df[col].fillna('').astype(str)

df.loc[df[col] == 'None', col] = ''

# Convert back to PyArrow and append

processed\_batch = pa.Table.from\_pandas(df)

batches.append(processed\_batch)

if batches:

result\_table = pa.concat\_tables(batches)

result\_df = result\_table.to\_pandas()

# Debug: Print sample of processed data

print("\nProcessed data sample:")

print(result\_df.head())

print("\nColumn dtypes:")

print(result\_df.dtypes)

return [result\_df.columns.tolist()] + result\_df.values.tolist()

return [["No data found"]]

except Exception as e:

print(f"Error in processing: {e}")

return [["Error: " + str(e)]]

Key changes based on the sample data:

Memo Column:

All values were 'None' in sample

Added specific handling for 'None' strings

Converting to empty string when None

Date Column:

Format is 'M/D/YYYY'

Added specific format in pd.to\_datetime

Converting to Excel dates

Amount Column:

Format is '$0.00'

Added handling for dollar sign

Converting to numeric

Qty Column:

Already numeric but as string

Simple numeric conversion

Other Columns:

Added general cleaning for 'None' values

Converting to empty strings where appropriate

Column Names:

Added cleaning of column names to remove extra spaces

This version should:

Properly handle the 'None' values

Correctly convert dates from 'M/D/YYYY' format

Clean up amount values from '$0.00' format

Convert quantities to numeric

Clean up all other string columns

Try this version and let me know if it properly cleans the data now.

Okay - That worked finally! Now can we try replacing pandas replacement part with pyarrow for date, memo, etc. replacements?

Here's the version using PyArrow computations instead of pandas for better performance:

python

@xw.func

def process\_pyarrow\_chunks(file\_path):

import pyarrow as pa

import pyarrow.csv as csv

import pyarrow.compute as pc

import pandas as pd

from datetime import datetime

try:

# Define read options for PyArrow

read\_options = csv.ReadOptions(

block\_size=10 \* 1024 \* 1024,

use\_threads=True

)

parse\_options = csv.ParseOptions(

delimiter=',',

quote\_char='"'

)

convert\_options = csv.ConvertOptions(

strings\_can\_be\_null=True,

include\_columns=None

)

batches = []

batch\_count = 0

with csv.open\_csv(

file\_path,

read\_options=read\_options,

parse\_options=parse\_options,

convert\_options=convert\_options

) as reader:

for batch in reader:

batch\_count += 1

# Create a dictionary to store the processed columns

processed\_columns = {}

for column\_name in batch.schema.names:

column = batch.column(column\_name)

# Process Memo

if column\_name == 'Memo':

# Fill null values with empty string

column = pc.fill\_null(column, '')

# Convert to string

column = pc.cast(column, pa.string())

# Replace 'None' with empty string

column = pc.replace\_substring(column, 'None', '')

# Replace backslashes

column = pc.replace\_substring(column, '\\', ' ')

# Trim whitespace

column = pc.strip(column)

processed\_columns[column\_name] = column

# Process Date

elif column\_name == 'Date':

try:

# Convert to timestamp

column = pc.strptime(column, format='%m/%d/%Y', unit='s')

# Convert to days since Excel epoch (1899-12-30)

excel\_epoch = datetime(1899, 12, 30).timestamp()

column = pc.divide(pc.subtract(column, excel\_epoch), 86400) # seconds in a day

processed\_columns[column\_name] = column

except Exception as e:

print(f"Date conversion error: {e}")

processed\_columns[column\_name] = column

# Process Amount

elif column\_name == 'Amount':

# Fill null values

column = pc.fill\_null(column, '$0.00')

# Convert to string

column = pc.cast(column, pa.string())

# Remove $ and commas

column = pc.replace\_substring(column, '$', '')

column = pc.replace\_substring(column, ',', '')

# Handle parentheses for negative numbers

column = pc.replace\_substring(column, '(', '-')

column = pc.replace\_substring(column, ')', '')

# Trim whitespace

column = pc.strip(column)

# Convert to float

column = pc.cast(column, pa.float64())

processed\_columns[column\_name] = column

# Process Qty

elif column\_name == 'Qty':

# Convert to float directly if possible

column = pc.cast(column, pa.float64())

processed\_columns[column\_name] = column

# Process other string columns

else:

# Fill null values with empty string

column = pc.fill\_null(column, '')

# Convert to string

column = pc.cast(column, pa.string())

# Replace 'None' with empty string

column = pc.replace\_substring(column, 'None', '')

# Trim whitespace

column = pc.strip(column)

processed\_columns[column\_name] = column

# Create new table with processed columns

processed\_batch = pa.Table.from\_arrays(

[processed\_columns[name] for name in batch.schema.names],

names=batch.schema.names

)

batches.append(processed\_batch)

if batches:

result\_table = pa.concat\_tables(batches)

result\_df = result\_table.to\_pandas()

# Debug: Print sample of processed data

print("\nProcessed data sample:")

print(result\_df.head())

print("\nColumn dtypes:")

print(result\_df.dtypes)

return [result\_df.columns.tolist()] + result\_df.values.tolist()

return [["No data found"]]

except Exception as e:

print(f"Error in processing: {e}")

return [["Error: " + str(e)]]

Key changes:

Replaced pandas operations with PyArrow compute functions:

Used pc.fill\_null() instead of fillna()

Used pc.cast() for type conversions

Used pc.replace\_substring() instead of pandas string replacements

Used pc.strip() for trimming whitespace

Used pc.strptime() for date parsing

Processing is now done column by column using PyArrow's compute functions

Date conversion uses PyArrow's timestamp functions

String operations use PyArrow's built-in string functions

Benefits of this approach:

Better memory efficiency

Faster processing due to native C++ implementations

Maintains data in Arrow format throughout processing

Reduced conversion overhead between pandas and Arrow

The code should now be more efficient while producing the same results as the pandas version. Let me know if you need any adjustments or if you see any performance differences.

Error: module 'pyarrow.compute' has no attribute 'strip'

Apologies for the error. You're right - let's correct it. For string operations in PyArrow, we need to use utf8\_trim() instead of strip(). Here's the corrected version:

python

@xw.func

def process\_pyarrow\_chunks(file\_path):

import pyarrow as pa

import pyarrow.csv as csv

import pyarrow.compute as pc

import pandas as pd

from datetime import datetime

try:

# Define read options for PyArrow

read\_options = csv.ReadOptions(

block\_size=10 \* 1024 \* 1024,

use\_threads=True

)

parse\_options = csv.ParseOptions(

delimiter=',',

quote\_char='"'

)

convert\_options = csv.ConvertOptions(

strings\_can\_be\_null=True,

include\_columns=None

)

batches = []

batch\_count = 0

with csv.open\_csv(

file\_path,

read\_options=read\_options,

parse\_options=parse\_options,

convert\_options=convert\_options

) as reader:

for batch in reader:

batch\_count += 1

# Create a dictionary to store the processed columns

processed\_columns = {}

for column\_name in batch.schema.names:

column = batch.column(column\_name)

# Process Memo

if column\_name == 'Memo':

# Fill null values with empty string

column = pc.fill\_null(column, '')

# Convert to string

column = pc.cast(column, pa.string())

# Replace 'None' with empty string

column = pc.replace\_substring(column, 'None', '')

# Replace backslashes

column = pc.replace\_substring(column, '\\', ' ')

# Trim whitespace

column = pc.utf8\_trim(column)

processed\_columns[column\_name] = column

# Process Date

elif column\_name == 'Date':

try:

# Convert to timestamp

column = pc.strptime(column, format='%m/%d/%Y', unit='s')

# Convert to days since Excel epoch (1899-12-30)

excel\_epoch = datetime(1899, 12, 30).timestamp()

column = pc.divide(pc.subtract(column, excel\_epoch), 86400) # seconds in a day

processed\_columns[column\_name] = column

except Exception as e:

print(f"Date conversion error: {e}")

processed\_columns[column\_name] = column

# Process Amount

elif column\_name == 'Amount':

# Fill null values

column = pc.fill\_null(column, '$0.00')

# Convert to string

column = pc.cast(column, pa.string())

# Remove $ and commas

column = pc.replace\_substring(column, '$', '')

column = pc.replace\_substring(column, ',', '')

# Handle parentheses for negative numbers

column = pc.replace\_substring(column, '(', '-')

column = pc.replace\_substring(column, ')', '')

# Trim whitespace

column = pc.utf8\_trim(column)

# Convert to float

column = pc.cast(column, pa.float64())

processed\_columns[column\_name] = column

# Process Qty

elif column\_name == 'Qty':

# Convert to float directly if possible

column = pc.cast(column, pa.float64())

processed\_columns[column\_name] = column

# Process other string columns

else:

# Fill null values with empty string

column = pc.fill\_null(column, '')

# Convert to string

column = pc.cast(column, pa.string())

# Replace 'None' with empty string

column = pc.replace\_substring(column, 'None', '')

# Trim whitespace

column = pc.utf8\_trim(column)

processed\_columns[column\_name] = column

# Create new table with processed columns

processed\_batch = pa.Table.from\_arrays(

[processed\_columns[name] for name in batch.schema.names],

names=batch.schema.names

)

batches.append(processed\_batch)

if batches:

result\_table = pa.concat\_tables(batches)

result\_df = result\_table.to\_pandas()

# Debug: Print sample of processed data

print("\nProcessed data sample:")

print(result\_df.head())

print("\nColumn dtypes:")

print(result\_df.dtypes)

return [result\_df.columns.tolist()] + result\_df.values.tolist()

return [["No data found"]]

except Exception as e:

print(f"Error in processing: {e}")

return [["Error: " + str(e)]]

Key changes:

Replaced pc.strip() with pc.utf8\_trim()

All other PyArrow operations remain the same

This should now work correctly with PyArrow's string processing functions. Let me know if you encounter any other issues!

Error: Function 'utf8\_trim' cannot be called without options

Ah yes, we need to specify the whitespace trimming options for utf8\_trim(). Here's the corrected version:

python

@xw.func

def process\_pyarrow\_chunks(file\_path):

import pyarrow as pa

import pyarrow.csv as csv

import pyarrow.compute as pc

import pandas as pd

from datetime import datetime

try:

# Define read options for PyArrow

read\_options = csv.ReadOptions(

block\_size=10 \* 1024 \* 1024,

use\_threads=True

)

parse\_options = csv.ParseOptions(

delimiter=',',

quote\_char='"'

)

convert\_options = csv.ConvertOptions(

strings\_can\_be\_null=True,

include\_columns=None

)

batches = []

batch\_count = 0

with csv.open\_csv(

file\_path,

read\_options=read\_options,

parse\_options=parse\_options,

convert\_options=convert\_options

) as reader:

for batch in reader:

batch\_count += 1

# Create a dictionary to store the processed columns

processed\_columns = {}

for column\_name in batch.schema.names:

column = batch.column(column\_name)

# Process Memo

if column\_name == 'Memo':

# Fill null values with empty string

column = pc.fill\_null(column, '')

# Convert to string

column = pc.cast(column, pa.string())

# Replace 'None' with empty string

column = pc.replace\_substring(column, 'None', '')

# Replace backslashes

column = pc.replace\_substring(column, '\\', ' ')

# Trim whitespace (both sides)

column = pc.utf8\_trim\_whitespace(column)

processed\_columns[column\_name] = column

# Process Date

elif column\_name == 'Date':

try:

# Convert to timestamp

column = pc.strptime(column, format='%m/%d/%Y', unit='s')

# Convert to days since Excel epoch (1899-12-30)

excel\_epoch = datetime(1899, 12, 30).timestamp()

column = pc.divide(pc.subtract(column, excel\_epoch), 86400) # seconds in a day

processed\_columns[column\_name] = column

except Exception as e:

print(f"Date conversion error: {e}")

processed\_columns[column\_name] = column

# Process Amount

elif column\_name == 'Amount':

# Fill null values

column = pc.fill\_null(column, '$0.00')

# Convert to string

column = pc.cast(column, pa.string())

# Remove $ and commas

column = pc.replace\_substring(column, '$', '')

column = pc.replace\_substring(column, ',', '')

# Handle parentheses for negative numbers

column = pc.replace\_substring(column, '(', '-')

column = pc.replace\_substring(column, ')', '')

# Trim whitespace

column = pc.utf8\_trim\_whitespace(column)

# Convert to float

column = pc.cast(column, pa.float64())

processed\_columns[column\_name] = column

# Process Qty

elif column\_name == 'Qty':

# Convert to float directly if possible

column = pc.cast(column, pa.float64())

processed\_columns[column\_name] = column

# Process other string columns

else:

# Fill null values with empty string

column = pc.fill\_null(column, '')

# Convert to string

column = pc.cast(column, pa.string())

# Replace 'None' with empty string

column = pc.replace\_substring(column, 'None', '')

# Trim whitespace

column = pc.utf8\_trim\_whitespace(column)

processed\_columns[column\_name] = column

# Create new table with processed columns

processed\_batch = pa.Table.from\_arrays(

[processed\_columns[name] for name in batch.schema.names],

names=batch.schema.names

)

batches.append(processed\_batch)

if batches:

result\_table = pa.concat\_tables(batches)

result\_df = result\_table.to\_pandas()

# Debug: Print sample of processed data

print("\nProcessed data sample:")

print(result\_df.head())

print("\nColumn dtypes:")

print(result\_df.dtypes)

return [result\_df.columns.tolist()] + result\_df.values.tolist()

return [["No data found"]]

except Exception as e:

print(f"Error in processing: {e}")

return [["Error: " + str(e)]]

Key changes:

Replaced pc.utf8\_trim() with pc.utf8\_trim\_whitespace()

This function trims all whitespace characters from both ends of the string

No additional options are needed

This should now work correctly. The utf8\_trim\_whitespace() function is a more convenient version that handles all types of whitespace characters automatically. Let me know if you encounter any other issues!

It seems we need to stick to pandas for string conversion part as pyarrow is weak there somehow. I don't know why.

You're right. PyArrow's string manipulation capabilities are more limited compared to pandas, and some operations can be tricky to implement. Let's create a hybrid approach where we use PyArrow for the heavy lifting and pandas for string manipulations:

python

@xw.func

def process\_pyarrow\_chunks(file\_path):

import pyarrow as pa

import pyarrow.csv as csv

import pyarrow.compute as pc

import pandas as pd

from datetime import datetime

try:

# Define read options for PyArrow

read\_options = csv.ReadOptions(

block\_size=10 \* 1024 \* 1024,

use\_threads=True

)

parse\_options = csv.ParseOptions(

delimiter=',',

quote\_char='"'

)

convert\_options = csv.ConvertOptions(

strings\_can\_be\_null=True,

include\_columns=None

)

batches = []

batch\_count = 0

with csv.open\_csv(

file\_path,

read\_options=read\_options,

parse\_options=parse\_options,

convert\_options=convert\_options

) as reader:

for batch in reader:

batch\_count += 1

df = batch.to\_pandas()

# Use pandas for string operations

# Clean column names

df.columns = df.columns.str.strip()

# Process Memo

if 'Memo' in df.columns:

df['Memo'] = df['Memo'].fillna('').astype(str)

df.loc[df['Memo'] == 'None', 'Memo'] = ''

df['Memo'] = (df['Memo']

.str.replace('\\', ' ', regex=False)

.str.replace(r'\s+', ' ', regex=True)

.str.strip())

# Use PyArrow for numeric and date operations

if 'Date' in df.columns:

try:

date\_column = pc.strptime(pa.array(df['Date']), format='%m/%d/%Y', unit='s')

excel\_epoch = datetime(1899, 12, 30).timestamp()

df['Date'] = pc.divide(pc.subtract(date\_column, excel\_epoch), 86400).to\_pandas()

except Exception as e:

print(f"Date conversion error: {e}")

# Process Amount

if 'Amount' in df.columns:

df['Amount'] = (df['Amount']

.fillna('$0.00')

.astype(str)

.str.replace('$', '', regex=False)

.str.replace(',', '', regex=False)

.str.replace('(', '-', regex=False)

.str.replace(')', '', regex=False)

.str.strip())

df['Amount'] = pd.to\_numeric(df['Amount'], errors='coerce')

# Process Qty using PyArrow

if 'Qty' in df.columns:

df['Qty'] = pc.cast(pa.array(df['Qty']), pa.float64()).to\_pandas()

# Process other string columns with pandas

for col in df.columns:

if col not in ['Date', 'Amount', 'Qty']:

df[col] = df[col].fillna('').astype(str)

df.loc[df[col] == 'None', col] = ''

df[col] = df[col].str.strip()

# Convert back to PyArrow and append

processed\_batch = pa.Table.from\_pandas(df)

batches.append(processed\_batch)

if batches:

result\_table = pa.concat\_tables(batches)

result\_df = result\_table.to\_pandas()

# Debug: Print sample of processed data

print("\nProcessed data sample:")

print(result\_df.head())

print("\nColumn dtypes:")

print(result\_df.dtypes)

return [result\_df.columns.tolist()] + result\_df.values.tolist()

return [["No data found"]]

except Exception as e:

print(f"Error in processing: {e}")

return [["Error: " + str(e)]]

Key changes:

Using pandas for string operations:

Column name cleaning

Memo field processing

General string column cleaning

Amount string preprocessing

Using PyArrow for:

Date conversions

Numeric casting

Batch processing

File reading

This hybrid approach gives us:

Better string manipulation capabilities from pandas

Fast numeric operations from PyArrow

Efficient file reading and batch processing from PyArrow

More reliable string handling

This should give you the best of both worlds - pandas' flexible string operations with PyArrow's performance benefits for everything else. Let me know if this works better!

Can we create a sample data frame using pyyarrow and check why replacement of dates, memo, etc. is failing?

Yes, let's create a sample DataFrame with PyArrow to debug the string operations. Here's a test script:

python

import pyarrow as pa

import pyarrow.compute as pc

import pandas as pd

from datetime import datetime

# Create sample data

data = {

'Date': ['1/1/2023', '2/15/2023', '3/30/2023', None, '12/31/2023'],

'Memo': ['Test memo\\with\\backslash', None, 'None', ' Extra spaces ', 'Normal memo'],

'Amount': ['$1,234.56', '($789.01)', '$0.00', None, '$5,432.10'],

'Qty': ['1', '2', None, '4', '5'],

'Other': ['Test\\string', None, 'None', ' Spaces ', 'Normal']

}

# Convert to PyArrow Table

table = pa.Table.from\_pydict(data)

print("\nOriginal PyArrow Table:")

print(table)

# Let's try different operations and print results at each step

# 1. Test Memo processing

print("\nTesting Memo processing:")

memo\_col = table.column('Memo')

print("Original Memo:", memo\_col)

# Step by step memo processing

memo\_filled = pc.fill\_null(memo\_col, '')

print("\nAfter fill\_null:", memo\_filled)

memo\_string = pc.cast(memo\_filled, pa.string())

print("\nAfter cast to string:", memo\_string)

memo\_no\_none = pc.replace\_substring(memo\_string, 'None', '')

print("\nAfter replacing 'None':", memo\_no\_none)

memo\_no\_backslash = pc.replace\_substring(memo\_no\_none, '\\', ' ')

print("\nAfter replacing backslash:", memo\_no\_backslash)

# 2. Test Date processing

print("\nTesting Date processing:")

date\_col = table.column('Date')

print("Original Date:", date\_col)

try:

date\_timestamp = pc.strptime(date\_col, format='%m/%d/%Y', unit='s')

print("\nAfter strptime:", date\_timestamp)

excel\_epoch = datetime(1899, 12, 30).timestamp()

date\_excel = pc.divide(pc.subtract(date\_timestamp, excel\_epoch), 86400)

print("\nAfter Excel conversion:", date\_excel)

except Exception as e:

print(f"Date conversion error: {e}")

# 3. Test Amount processing

print("\nTesting Amount processing:")

amount\_col = table.column('Amount')

print("Original Amount:", amount\_col)

amount\_filled = pc.fill\_null(amount\_col, '$0.00')

print("\nAfter fill\_null:", amount\_filled)

amount\_string = pc.cast(amount\_filled, pa.string())

print("\nAfter cast to string:", amount\_string)

amount\_no\_dollar = pc.replace\_substring(amount\_string, '$', '')

print("\nAfter removing $:", amount\_no\_dollar)

amount\_no\_comma = pc.replace\_substring(amount\_no\_dollar, ',', '')

print("\nAfter removing commas:", amount\_no\_comma)

# Compare with pandas approach

print("\nComparing with pandas approach:")

df = pd.DataFrame(data)

print("\nPandas processing:")

df['Memo'] = (df['Memo']

.fillna('')

.astype(str)

.replace('None', '')

.str.replace('\\', ' ', regex=False)

.str.strip())

print("Pandas Memo result:", df['Memo'].tolist())

# Print final comparison

print("\nFinal comparison:")

print("PyArrow Memo:", memo\_no\_backslash.to\_pylist())

print("Pandas Memo:", df['Memo'].tolist())

This script will help us:

See exactly what each PyArrow operation does

Compare PyArrow and pandas results

Identify where the processing might be failing

Understand the differences in how each library handles the data

Let's run this and analyze the output to see where the issues are occurring. Could you run this and share what output you get? This will help us pinpoint exactly where PyArrow's string processing might be falling short.

Make this sample run with xlwings in excel with printing in xlwings UDF server console to debug.

Here's the modified version that works with xlwings as a UDF to debug the processing:

python

import xlwings as xw

import pyarrow as pa

import pyarrow.compute as pc

import pandas as pd

from datetime import datetime

@xw.func

def test\_data\_processing():

"""Test function to debug data processing with sample data"""

try:

# Create sample data

data = {

'Date': ['1/1/2023', '2/15/2023', '3/30/2023', None, '12/31/2023'],

'Memo': ['Test memo\\with\\backslash', None, 'None', ' Extra spaces ', 'Normal memo'],

'Amount': ['$1,234.56', '($789.01)', '$0.00', None, '$5,432.10'],

'Qty': ['1', '2', None, '4', '5'],

'Other': ['Test\\string', None, 'None', ' Spaces ', 'Normal']

}

# Convert to PyArrow Table

table = pa.Table.from\_pydict(data)

print("\nOriginal PyArrow Table:")

print(table)

# Debug Memo processing

print("\n=== Memo Processing ===")

memo\_col = table.column('Memo')

print("Original Memo:", memo\_col.to\_pylist())

memo\_filled = pc.fill\_null(memo\_col, '')

print("After fill\_null:", memo\_filled.to\_pylist())

memo\_string = pc.cast(memo\_filled, pa.string())

print("After cast to string:", memo\_string.to\_pylist())

memo\_no\_none = pc.replace\_substring(memo\_string, 'None', '')

print("After replacing 'None':", memo\_no\_none.to\_pylist())

memo\_no\_backslash = pc.replace\_substring(memo\_no\_none, '\\', ' ')

print("After replacing backslash:", memo\_no\_backslash.to\_pylist())

# Debug Date processing

print("\n=== Date Processing ===")

date\_col = table.column('Date')

print("Original Date:", date\_col.to\_pylist())

try:

date\_timestamp = pc.strptime(date\_col, format='%m/%d/%Y', unit='s')

print("After strptime:", date\_timestamp.to\_pylist())

excel\_epoch = datetime(1899, 12, 30).timestamp()

date\_excel = pc.divide(pc.subtract(date\_timestamp, excel\_epoch), 86400)

print("After Excel conversion:", date\_excel.to\_pylist())

except Exception as e:

print(f"Date conversion error: {e}")

# Debug Amount processing

print("\n=== Amount Processing ===")

amount\_col = table.column('Amount')

print("Original Amount:", amount\_col.to\_pylist())

amount\_filled = pc.fill\_null(amount\_col, '$0.00')

print("After fill\_null:", amount\_filled.to\_pylist())

amount\_string = pc.cast(amount\_filled, pa.string())

print("After cast to string:", amount\_string.to\_pylist())

amount\_no\_dollar = pc.replace\_substring(amount\_string, '$', '')

print("After removing $:", amount\_no\_dollar.to\_pylist())

amount\_no\_comma = pc.replace\_substring(amount\_no\_dollar, ',', '')

print("After removing commas:", amount\_no\_comma.to\_pylist())

# Compare with pandas approach

print("\n=== Pandas Comparison ===")

df = pd.DataFrame(data)

# Pandas Memo processing

df['Memo'] = (df['Memo']

.fillna('')

.astype(str)

.replace('None', '')

.str.replace('\\', ' ', regex=False)

.str.strip())

print("Pandas Memo result:", df['Memo'].tolist())

# Pandas Amount processing

df['Amount'] = (df['Amount']

.fillna('$0.00')

.astype(str)

.str.replace('$', '', regex=False)

.str.replace(',', '', regex=False)

.str.replace('(', '-', regex=False)

.str.replace(')', '', regex=False)

.str.strip())

print("Pandas Amount result:", df['Amount'].tolist())

# Final comparison

print("\n=== Final Comparison ===")

print("PyArrow Memo:", memo\_no\_backslash.to\_pylist())

print("Pandas Memo:", df['Memo'].tolist())

# Return something to Excel

return [["Debug output in UDF Server console"]]

except Exception as e:

print(f"Error in test\_data\_processing: {e}")

return [["Error: " + str(e)]]

@xw.func

def test\_specific\_column(column\_name: str):

"""Test processing of a specific column"""

try:

# Create sample data with just the specified column

data = {

'Date': ['1/1/2023', '2/15/2023', '3/30/2023', None, '12/31/2023'],

'Memo': ['Test memo\\with\\backslash', None, 'None', ' Extra spaces ', 'Normal memo'],

'Amount': ['$1,234.56', '($789.01)', '$0.00', None, '$5,432.10'],

'Qty': ['1', '2', None, '4', '5'],

'Other': ['Test\\string', None, 'None', ' Spaces ', 'Normal']

}

if column\_name not in data:

return [["Column not found in sample data"]]

# Convert to PyArrow Table

table = pa.Table.from\_pydict({column\_name: data[column\_name]})

print(f"\n=== Testing {column\_name} Processing ===")

print(f"Original {column\_name}:", table.column(column\_name).to\_pylist())

# Process column based on its type

if column\_name == 'Memo':

# Test Memo processing

col = table.column(column\_name)

result = (pd.Series(col.to\_pylist())

.fillna('')

.astype(str)

.replace('None', '')

.str.replace('\\', ' ', regex=False)

.str.strip())

print(f"Processed {column\_name}:", result.tolist())

elif column\_name == 'Amount':

# Test Amount processing

col = table.column(column\_name)

result = (pd.Series(col.to\_pylist())

.fillna('$0.00')

.astype(str)

.str.replace('$', '', regex=False)

.str.replace(',', '', regex=False)

.str.replace('(', '-', regex=False)

.str.replace(')', '', regex=False)

.str.strip())

print(f"Processed {column\_name}:", result.tolist())

elif column\_name == 'Date':

# Test Date processing

col = table.column(column\_name)

try:

date\_timestamp = pc.strptime(col, format='%m/%d/%Y', unit='s')

excel\_epoch = datetime(1899, 12, 30).timestamp()

result = pc.divide(pc.subtract(date\_timestamp, excel\_epoch), 86400)

print(f"Processed {column\_name}:", result.to\_pylist())

except Exception as e:

print(f"Date conversion error: {e}")

return [["Check UDF Server console for debug output"]]

except Exception as e:

print(f"Error in test\_specific\_column: {e}")

return [["Error: " + str(e)]]

To use this in Excel:

First function test\_data\_processing():

excel

=test\_data\_processing()

This will run all the tests and print detailed debug information to the UDF server console.

Second function test\_specific\_column():

excel

=test\_specific\_column("Memo")

=test\_specific\_column("Date")

=test\_specific\_column("Amount")

This allows you to test processing of individual columns.

The debug output will appear in the UDF server console, showing:

Original values

Step-by-step processing results

Comparison between PyArrow and pandas approaches

Any errors that occur

This should help identify exactly where the processing is failing and what the differences are between PyArrow and pandas processing methods.

Changing like this: data = { 'Date': ['1/1/2023', '2/15/2023', '3/30/2023', None, '12/31/2023'], 'Memo': ['Test memo\with\backslash', None, 'None', ' Extra spaces ', 'Normal memo'], 'Amount': ['$1,234.56', '($789.01)', '$0.00', None, '$5,432.10'], 'Qty': ['1', '2', None, '4', '5'], 'Other': ['Test\string', None, 'None', ' Spaces ', 'Normal'] } Single backslash does not seem to work: === Testing Memo Processing === Original Memo: ['Test memo\\with\x08ackslash', None, 'None', ' Extra spaces ', 'Normal memo'] Processed Memo: ['Test memo with\x08ackslash', '', '', 'Extra spaces', 'Normal memo'] === Testing Other Processing === Original Other: ['Test\\string', None, 'None', ' Spaces ', 'Normal']

Ah, I see the issue. When using single backslashes in Python strings, they're interpreted as escape characters. Let's modify the test data to use raw strings (r-strings) to handle the backslashes correctly:

python

@xw.func

def test\_specific\_column(column\_name: str):

"""Test processing of a specific column"""

try:

# Create sample data with raw strings for backslashes

data = {

'Date': ['1/1/2023', '2/15/2023', '3/30/2023', None, '12/31/2023'],

'Memo': [r'Test memo\with\backslash', None, 'None', ' Extra spaces ', 'Normal memo'],

'Amount': ['$1,234.56', '($789.01)', '$0.00', None, '$5,432.10'],

'Qty': ['1', '2', None, '4', '5'],

'Other': [r'Test\string', None, 'None', ' Spaces ', 'Normal']

}

if column\_name not in data:

return [["Column not found in sample data"]]

# Convert to PyArrow Table

table = pa.Table.from\_pydict({column\_name: data[column\_name]})

print(f"\n=== Testing {column\_name} Processing ===")

print(f"Original {column\_name}:", table.column(column\_name).to\_pylist())

# Process column based on its type

if column\_name in ['Memo', 'Other']:

# Test string processing

col = table.column(column\_name)

result = (pd.Series(col.to\_pylist())

.fillna('')

.astype(str)

.replace('None', '')

.str.replace(r'\\', ' ', regex=True) # Changed to regex=True for proper backslash handling

.str.replace(r'\s+', ' ', regex=True) # Normalize spaces

.str.strip())

print(f"Processed {column\_name}:", result.tolist())

elif column\_name == 'Amount':

# Test Amount processing

col = table.column(column\_name)

result = (pd.Series(col.to\_pylist())

.fillna('$0.00')

.astype(str)

.str.replace('$', '', regex=False)

.str.replace(',', '', regex=False)

.str.replace('(', '-', regex=False)

.str.replace(')', '', regex=False)

.str.strip())

print(f"Processed {column\_name}:", result.tolist())

elif column\_name == 'Date':

# Test Date processing

col = table.column(column\_name)

try:

date\_timestamp = pc.strptime(col, format='%m/%d/%Y', unit='s')

excel\_epoch = datetime(1899, 12, 30).timestamp()

result = pc.divide(pc.subtract(date\_timestamp, excel\_epoch), 86400)

print(f"Processed {column\_name}:", result.to\_pylist())

except Exception as e:

print(f"Date conversion error: {e}")

return [["Check UDF Server console for debug output"]]

except Exception as e:

print(f"Error in test\_specific\_column: {e}")

return [["Error: " + str(e)]]

Key changes:

Used r-strings (raw strings) for the test data with backslashes

Changed the string replacement to use regex=True for proper backslash handling

Added an additional step to normalize spaces using r'\s+'

This should now correctly handle the backslashes in the test data. When you run:

excel

=test\_specific\_column("Memo")

You should see proper handling of the backslashes without the \x08 escape sequence appearing in the output. The backslashes should be properly replaced with spaces, and multiple spaces should be normalized to single spaces.

The output should look something like:

=== Testing Memo Processing ===

Original Memo: ['Test memo\with\backslash', None, 'None', ' Extra spaces ', 'Normal memo']

Processed Memo: ['Test memo with backslash', '', '', 'Extra spaces', 'Normal memo']

Test is ok with that r prefix as above but the real data has memo field like this: remake peice 1 10mm Opti-White+Temp. \ .060 Clear PVB \ 10mm 66 3/ 4 113 1/ 4 53.84 SF 23.01 1238.73 1238.73 16142(R) Opti-White+Temp. What do you do to replace \ in that string - panads is success but pyarrow is failure why?

Let's test with the actual data pattern to see why PyArrow struggles with it. Here's a modified test:

python

@xw.func

def test\_backslash\_handling():

"""Test backslash handling with real data example"""

try:

# Real data example

test\_memo = "remake peice 1 10mm Opti-White+Temp. \\ .060 Clear PVB \\ 10mm 66 3/ 4 113 1/ 4 53.84 SF 23.01 1238.73 1238.73 16142(R) Opti-White+Temp."

# Create test data

data = {

'Memo': [test\_memo, None, 'Normal text', ' Spaces ', 'Test\\string']

}

print("\n=== Original String ===")

print(test\_memo)

# PyArrow approach

print("\n=== PyArrow Processing ===")

table = pa.Table.from\_pydict(data)

memo\_col = table.column('Memo')

print("Original:", memo\_col.to\_pylist())

try:

# Try different PyArrow approaches

print("\nTrying different PyArrow replacements:")

# Approach 1: Basic replacement

result1 = pc.replace\_substring(memo\_col, '\\', ' ')

print("Basic replace\_substring:", result1.to\_pylist())

# Approach 2: Multiple steps

result2 = pc.replace\_substring\_regex(memo\_col, r'\\', ' ')

print("replace\_substring\_regex:", result2.to\_pylist())

except Exception as e:

print(f"PyArrow error: {e}")

# Pandas approach

print("\n=== Pandas Processing ===")

df = pd.DataFrame(data)

# Show intermediate steps

print("1. Original:", df['Memo'].tolist())

df['Memo'] = df['Memo'].fillna('')

print("2. After fillna:", df['Memo'].tolist())

df['Memo'] = df['Memo'].astype(str)

print("3. After astype(str):", df['Memo'].tolist())

df['Memo'] = df['Memo'].replace('None', '')

print("4. After replace None:", df['Memo'].tolist())

df['Memo'] = df['Memo'].str.replace('\\', ' ', regex=False)

print("5. After replace backslash:", df['Memo'].tolist())

df['Memo'] = df['Memo'].str.replace(r'\s+', ' ', regex=True)

print("6. After normalize spaces:", df['Memo'].tolist())

df['Memo'] = df['Memo'].str.strip()

print("7. Final result:", df['Memo'].tolist())

return [["Check UDF Server console for debug output"]]

except Exception as e:

print(f"Error in test: {e}")

return [["Error: " + str(e)]]

This test will help us understand:

Exactly how PyArrow and pandas handle the backslashes differently

What happens at each step of the processing

Why pandas succeeds where PyArrow fails

The key differences are:

Pandas has more flexible string handling with str.replace()

Pandas allows you to specify regex=False which treats the backslash as a literal character

PyArrow's replace\_substring() might be interpreting the backslash as an escape character

Let's run this test and see the detailed output. This will help us determine the best way to handle the backslashes in the real data. Could you run this and share what output you get? We can then modify the main processing function accordingly.

The solution might be to:

Convert to pandas for string operations

Do the string replacements

Convert back to PyArrow for the rest of the processing

Or find a specific PyArrow pattern that works with these literal backslashes.

But you have included \\ instead of single \ as I see in excel cell.

Ah, you're right. Let's modify the test to use the exact string as it appears in Excel (with single backslash):

python

@xw.func

def test\_backslash\_handling():

"""Test backslash handling with real data example"""

try:

# Real data example - exactly as it appears in Excel cell

test\_memo = "remake peice 1 10mm Opti-White+Temp. \ .060 Clear PVB \ 10mm 66 3/ 4 113 1/ 4 53.84 SF 23.01 1238.73 1238.73 16142(R) Opti-White+Temp."

# Create test data

data = {

'Memo': [test\_memo, None, 'Normal text', ' Spaces ', 'Test\string']

}

print("\n=== Original String ===")

print(test\_memo)

# PyArrow approach

print("\n=== PyArrow Processing ===")

table = pa.Table.from\_pydict(data)

memo\_col = table.column('Memo')

print("Original:", memo\_col.to\_pylist())

try:

# Try different PyArrow approaches

print("\nTrying different PyArrow replacements:")

# Approach 1: Basic replacement

result1 = pc.replace\_substring(memo\_col, '\\', ' ')

print("Basic replace\_substring:", result1.to\_pylist())

# Approach 2: Using regex

result2 = pc.replace\_substring\_regex(memo\_col, r'\\', ' ')

print("replace\_substring\_regex:", result2.to\_pylist())

except Exception as e:

print(f"PyArrow error: {e}")

# Pandas approach

print("\n=== Pandas Processing ===")

df = pd.DataFrame(data)

# Show intermediate steps

print("1. Original:", df['Memo'].tolist())

df['Memo'] = df['Memo'].fillna('')

print("2. After fillna:", df['Memo'].tolist())

df['Memo'] = df['Memo'].astype(str)

print("3. After astype(str):", df['Memo'].tolist())

df['Memo'] = df['Memo'].replace('None', '')

print("4. After replace None:", df['Memo'].tolist())

# Try both regex=True and regex=False for comparison

df\_copy = df.copy()

# With regex=False

df['Memo'] = df['Memo'].str.replace('\\', ' ', regex=False)

print("5a. After replace backslash (regex=False):", df['Memo'].tolist())

# With regex=True

df\_copy['Memo'] = df\_copy['Memo'].str.replace(r'\\', ' ', regex=True)

print("5b. After replace backslash (regex=True):", df\_copy['Memo'].tolist())

df['Memo'] = df['Memo'].str.replace(r'\s+', ' ', regex=True)

print("6. After normalize spaces:", df['Memo'].tolist())

df['Memo'] = df['Memo'].str.strip()

print("7. Final result:", df['Memo'].tolist())

return [["Check UDF Server console for debug output"]]

except Exception as e:

print(f"Error in test: {e}")

return [["Error: " + str(e)]]

This test should now exactly match what you're seeing in Excel. The key differences are:

Using single backslash in the test string

Testing both regex=True and regex=False in pandas

Showing the exact string content at each step

This should help us understand:

How PyArrow handles the single backslash

Why pandas might be handling it better

Which pandas approach (regex=True or regex=False) works correctly

Let's run this and see the output. Based on the results, we can determine the best approach for handling these single backslashes in your real data.

If we do that, I get warning in console: d:\dev\pyarrowtest.py:115: SyntaxWarning: invalid escape sequence '\ ' test\_memo = "remake peice 1 10mm Opti-White+Temp. \ .060 Clear PVB \ 10mm 66 3/ 4 113 1/ 4 53.84 SF 23.01 1238.73 1238.73 16142(R) Opti-White+Temp." d:\dev\pyarrowtest.py:119: SyntaxWarning: invalid escape sequence '\s' 'Memo': [test\_memo, None, 'Normal text', ' Spaces ', 'Test\string'] Imported functions from the following modules: xlwings\_sqlite\_python\_pandas, xlwings\_sqlite\_python, pyarrowtest

Ah yes, you're right. We need to use raw strings (r-prefix) to handle the literal backslashes correctly in Python. Here's the corrected version:

python

@xw.func

def test\_backslash\_handling():

"""Test backslash handling with real data example"""

try:

# Real data example - using raw string to handle backslashes correctly

test\_memo = r"remake peice 1 10mm Opti-White+Temp. \ .060 Clear PVB \ 10mm 66 3/ 4 113 1/ 4 53.84 SF 23.01 1238.73 1238.73 16142(R) Opti-White+Temp."

# Create test data with raw string

data = {

'Memo': [test\_memo, None, 'Normal text', ' Spaces ', r'Test\string']

}

print("\n=== Original String ===")

print(test\_memo)

# PyArrow approach

print("\n=== PyArrow Processing ===")

table = pa.Table.from\_pydict(data)

memo\_col = table.column('Memo')

print("Original:", memo\_col.to\_pylist())

try:

# Try different PyArrow approaches

print("\nTrying different PyArrow replacements:")

# Approach 1: Basic replacement

result1 = pc.replace\_substring(memo\_col, '\\', ' ')

print("Basic replace\_substring:", result1.to\_pylist())

# Approach 2: Using regex

result2 = pc.replace\_substring\_regex(memo\_col, r'\\', ' ')

print("replace\_substring\_regex:", result2.to\_pylist())

except Exception as e:

print(f"PyArrow error: {e}")

# Pandas approach

print("\n=== Pandas Processing ===")

df = pd.DataFrame(data)

# Show intermediate steps

print("1. Original:", df['Memo'].tolist())

df['Memo'] = df['Memo'].fillna('')

print("2. After fillna:", df['Memo'].tolist())

df['Memo'] = df['Memo'].astype(str)

print("3. After astype(str):", df['Memo'].tolist())

df['Memo'] = df['Memo'].replace('None', '')

print("4. After replace None:", df['Memo'].tolist())

# Try both regex=True and regex=False for comparison

df\_copy = df.copy()

# With regex=False

df['Memo'] = df['Memo'].str.replace('\\', ' ', regex=False)

print("5a. After replace backslash (regex=False):", df['Memo'].tolist())

# With regex=True

df\_copy['Memo'] = df\_copy['Memo'].str.replace(r'\\', ' ', regex=True)

print("5b. After replace backslash (regex=True):", df\_copy['Memo'].tolist())

df['Memo'] = df['Memo'].str.replace(r'\s+', ' ', regex=True)

print("6. After normalize spaces:", df['Memo'].tolist())

df['Memo'] = df['Memo'].str.strip()

print("7. Final result:", df['Memo'].tolist())

return [["Check UDF Server console for debug output"]]

except Exception as e:

print(f"Error in test: {e}")

return [["Error: " + str(e)]]

The key changes are:

Added 'r' prefix to the test\_memo string

Added 'r' prefix to the test string in the data dictionary

This should now:

Remove the syntax warnings

Properly handle the backslashes as literal characters

Show us exactly how PyArrow and pandas handle the backslash replacements

The r-prefix tells Python to treat backslashes as literal characters rather than escape sequences. This should give us a clearer picture of how to handle the backslashes in your real data.

xlwings server running, clsid={926C91E9-8456-4B13-BD6C-E4EACC54C8D2}, event loop on 20484 d:\dev\pyarrowtest.py:119: SyntaxWarning: invalid escape sequence '\s' 'Memo': [test\_memo, None, 'Normal text', ' Spaces ', 'Test\string'] Imported functions from the following modules: xlwings\_sqlite\_python\_pandas, xlwings\_sqlite\_python, pyarrowtest

Let's fix that remaining warning by using the r-prefix for the 'Test\string' as well:

python

@xw.func

def test\_backslash\_handling():

"""Test backslash handling with real data example"""

try:

# Real data example - using raw string to handle backslashes correctly

test\_memo = r"remake peice 1 10mm Opti-White+Temp. \ .060 Clear PVB \ 10mm 66 3/ 4 113 1/ 4 53.84 SF 23.01 1238.73 1238.73 16142(R) Opti-White+Temp."

# Create test data - using raw string for all backslash-containing strings

data = {

'Memo': [test\_memo, None, 'Normal text', ' Spaces ', r'Test\string']

}

# Rest of the code remains the same...

Now there should be no syntax warnings at all. Could you try running it and let me know what output you get? This will help us determine:

If PyArrow can handle the backslash replacement correctly

Which pandas approach works better (regex=True or regex=False)

The best way to implement this in your main processing function

The key is to understand how the data flows from Excel through PyArrow and pandas, and where the backslash handling needs to be adjusted.

See my code below: @xw.func def process\_pyarrow\_chunks\_new(file\_path): import pyarrow as pa import pyarrow.csv as csv import pyarrow.compute as pc import pandas as pd from datetime import datetime try: # Define read options for PyArrow read\_options = csv.ReadOptions( block\_size=10 \* 1024 \* 1024, use\_threads=True ) parse\_options = csv.ParseOptions( delimiter=',', quote\_char='"' ) convert\_options = csv.ConvertOptions( strings\_can\_be\_null=True, include\_columns=None ) batches = [] batch\_count = 0 with csv.open\_csv( file\_path, read\_options=read\_options, parse\_options=parse\_options, convert\_options=convert\_options ) as reader: for batch in reader: batch\_count += 1 # Create a dictionary to store the processed columns processed\_columns = {} for column\_name in batch.schema.names: column = batch.column(column\_name) # Process Memo if column\_name == 'Memo': print("Memo Found In Batch") # Fill null values with empty string column = pc.fill\_null(column, 'abc') # Convert to string column = pc.cast(column, pa.string()) # Replace 'None' with empty string column = pc.replace\_substring(column, 'None', '') # Replace backslashes column = pc.replace\_substring(column, '\\', ' ') # Trim whitespace (both sides) column = pc.utf8\_trim\_whitespace(column) processed\_columns[column\_name] = column # Process Date elif column\_name == 'Date': try: # Convert to timestamp column = pc.strptime(column, format='%m/%d/%Y', unit='s') # Convert to days since Excel epoch (1899-12-30) excel\_epoch = datetime(1899, 12, 30).timestamp() column = pc.divide(pc.subtract(column, excel\_epoch), 86400) # seconds in a day processed\_columns[column\_name] = column except Exception as e: print(f"Date conversion error: {e}") processed\_columns[column\_name] = column # Process Amount elif column\_name == 'Amount': # Fill null values column = pc.fill\_null(column, '$0.00') # Convert to string column = pc.cast(column, pa.string()) # Remove $ and commas column = pc.replace\_substring(column, '$', '') column = pc.replace\_substring(column, ',', '') # Handle parentheses for negative numbers column = pc.replace\_substring(column, '(', '-') column = pc.replace\_substring(column, ')', '') # Trim whitespace column = pc.utf8\_trim\_whitespace(column) # Convert to float column = pc.cast(column, pa.float64()) processed\_columns[column\_name] = column # Process Qty elif column\_name == 'Qty': # Convert to float directly if possible column = pc.cast(column, pa.float64()) processed\_columns[column\_name] = column # Process other string columns else: # Fill null values with empty string column = pc.fill\_null(column, '') # Convert to string column = pc.cast(column, pa.string()) # Replace 'None' with empty string column = pc.replace\_substring(column, 'None', '') # Trim whitespace column = pc.utf8\_trim\_whitespace(column) processed\_columns[column\_name] = column # Create new table with processed columns processed\_batch = pa.Table.from\_arrays( [processed\_columns[name] for name in batch.schema.names], names=batch.schema.names ) batches.append(processed\_batch) if batches: result\_table = pa.concat\_tables(batches) result\_df = result\_table.to\_pandas() # Debug: Print sample of processed data print("\nProcessed data sample:") print(result\_df.head()) print("\nColumn dtypes:") print(result\_df.dtypes) return [result\_df.columns.tolist()] + result\_df.values.tolist() return [["No data found"]] except Exception as e: print(f"Error in processing: {e}") return [["Error: " + str(e)]] I used simple debugging trick: print("Memo Found In Batch") But there seems to be a fundamental problem with your code. It never finds Memo column and prints this message. How then it will replace the "\". Same for date and other sanitized columns.

The issue might be with how the column names are being read or compared. Let's add more debugging to see what column names are actually present in the data:

python

@xw.func

def process\_pyarrow\_chunks\_new(file\_path):

import pyarrow as pa

import pyarrow.csv as csv

import pyarrow.compute as pc

import pandas as pd

from datetime import datetime

try:

# Define read options for PyArrow

read\_options = csv.ReadOptions(

block\_size=10 \* 1024 \* 1024,

use\_threads=True

)

parse\_options = csv.ParseOptions(

delimiter=',',

quote\_char='"'

)

convert\_options = csv.ConvertOptions(

strings\_can\_be\_null=True,

include\_columns=None

)

batches = []

batch\_count = 0

with csv.open\_csv(

file\_path,

read\_options=read\_options,

parse\_options=parse\_options,

convert\_options=convert\_options

) as reader:

for batch in reader:

batch\_count += 1

# Debug: Print schema and column names

print("\nBatch Schema:")

print(batch.schema)

print("\nColumn Names:")

print(batch.schema.names)

# Create a dictionary to store the processed columns

processed\_columns = {}

for column\_name in batch.schema.names:

# Debug: Print each column name being processed

print(f"\nProcessing column: '{column\_name}'")

column = batch.column(column\_name)

# Process Memo

if column\_name.strip().lower() == 'memo': # Make case-insensitive

print("Memo Found In Batch")

# Fill null values with empty string

column = pc.fill\_null(column, 'abc')

# Convert to string

column = pc.cast(column, pa.string())

# Replace 'None' with empty string

column = pc.replace\_substring(column, 'None', '')

# Replace backslashes

column = pc.replace\_substring(column, '\\', ' ')

# Trim whitespace (both sides)

column = pc.utf8\_trim\_whitespace(column)

processed\_columns[column\_name] = column

# Process Date

elif column\_name.strip().lower() == 'date':

print("Date Found In Batch")

try:

# Convert to timestamp

column = pc.strptime(column, format='%m/%d/%Y', unit='s')

# Convert to days since Excel epoch (1899-12-30)

excel\_epoch = datetime(1899, 12, 30).timestamp()

column = pc.divide(pc.subtract(column, excel\_epoch), 86400)

processed\_columns[column\_name] = column

except Exception as e:

print(f"Date conversion error: {e}")

processed\_columns[column\_name] = column

# Process Amount

elif column\_name.strip().lower() == 'amount':

print("Amount Found In Batch")

# Fill null values

column = pc.fill\_null(column, '$0.00')

# Convert to string

column = pc.cast(column, pa.string())

# Remove $ and commas

column = pc.replace\_substring(column, '$', '')

column = pc.replace\_substring(column, ',', '')

# Handle parentheses for negative numbers

column = pc.replace\_substring(column, '(', '-')

column = pc.replace\_substring(column, ')', '')

# Trim whitespace

column = pc.utf8\_trim\_whitespace(column)

# Convert to float

column = pc.cast(column, pa.float64())

processed\_columns[column\_name] = column

# Process Qty

elif column\_name.strip().lower() == 'qty':

print("Qty Found In Batch")

# Convert to float directly if possible

column = pc.cast(column, pa.float64())

processed\_columns[column\_name] = column

# Process other string columns

else:

# Fill null values with empty string

column = pc.fill\_null(column, '')

# Convert to string

column = pc.cast(column, pa.string())

# Replace 'None' with empty string

column = pc.replace\_substring(column, 'None', '')

# Trim whitespace

column = pc.utf8\_trim\_whitespace(column)

processed\_columns[column\_name] = column

# Create new table with processed columns

processed\_batch = pa.Table.from\_arrays(

[processed\_columns[name] for name in batch.schema.names],

names=batch.schema.names

)

batches.append(processed\_batch)

if batches:

result\_table = pa.concat\_tables(batches)

result\_df = result\_table.to\_pandas()

# Debug: Print sample of processed data

print("\nProcessed data sample:")

print(result\_df.head())

print("\nColumn dtypes:")

print(result\_df.dtypes)

return [result\_df.columns.tolist()] + result\_df.values.tolist()

return [["No data found"]]

except Exception as e:

print(f"Error in processing: {e}")

return [["Error: " + str(e)]]

Key changes:

Added debug printing of schema and column names

Added case-insensitive comparison for column names

Added debug printing for each column being processed

Added more "Found In Batch" messages for other columns

This will help us understand:

What column names are actually in your CSV file

If there are any case mismatches or hidden characters

If the column names are being read correctly

Could you run this version and share what appears in the debug output? This will help us identify why the column matching isn't working.

Finally discovered the root cause: Column Names: ['Project ', 'Class: Name ', 'Type ', 'Date ', 'Document Number ', 'Name ', 'Memo ', 'Account ', 'Clr ', 'Split ', 'Qty ', 'Amount '] As you can see each column name has extra space after the naem. So you need to trim the column names before we apply sanitization.

Ah, excellent catch! Yes, those trailing spaces in the column names are causing the exact matches to fail. Let's modify the code to handle this by trimming the column names when they're first read. Here's the corrected version:

python

@xw.func

def process\_pyarrow\_chunks\_new(file\_path):

import pyarrow as pa

import pyarrow.csv as csv

import pyarrow.compute as pc

import pandas as pd

from datetime import datetime

try:

# Define read options for PyArrow

read\_options = csv.ReadOptions(

block\_size=10 \* 1024 \* 1024,

use\_threads=True

)

parse\_options = csv.ParseOptions(

delimiter=',',

quote\_char='"'

)

convert\_options = csv.ConvertOptions(

strings\_can\_be\_null=True,

include\_columns=None

)

batches = []

batch\_count = 0

with csv.open\_csv(

file\_path,

read\_options=read\_options,

parse\_options=parse\_options,

convert\_options=convert\_options

) as reader:

for batch in reader:

batch\_count += 1

# Create a dictionary to store the processed columns

processed\_columns = {}

# Get cleaned column names (trimmed)

cleaned\_names = [name.strip() for name in batch.schema.names]

for i, column\_name in enumerate(batch.schema.names):

cleaned\_name = cleaned\_names[i]

column = batch.column(column\_name)

# Process Memo

if cleaned\_name == 'Memo':

print("Memo Found In Batch")

# Fill null values with empty string

column = pc.fill\_null(column, '')

# Convert to string

column = pc.cast(column, pa.string())

# Replace 'None' with empty string

column = pc.replace\_substring(column, 'None', '')

# Replace backslashes

column = pc.replace\_substring(column, '\\', ' ')

# Trim whitespace (both sides)

column = pc.utf8\_trim\_whitespace(column)

processed\_columns[column\_name] = column

# Process Date

elif cleaned\_name == 'Date':

print("Date Found In Batch")

try:

# Convert to timestamp

column = pc.strptime(column, format='%m/%d/%Y', unit='s')

# Convert to days since Excel epoch (1899-12-30)

excel\_epoch = datetime(1899, 12, 30).timestamp()

column = pc.divide(pc.subtract(column, excel\_epoch), 86400)

processed\_columns[column\_name] = column

except Exception as e:

print(f"Date conversion error: {e}")

processed\_columns[column\_name] = column

# Process Amount

elif cleaned\_name == 'Amount':

print("Amount Found In Batch")

# Fill null values

column = pc.fill\_null(column, '$0.00')

# Convert to string

column = pc.cast(column, pa.string())

# Remove $ and commas

column = pc.replace\_substring(column, '$', '')

column = pc.replace\_substring(column, ',', '')

# Handle parentheses for negative numbers

column = pc.replace\_substring(column, '(', '-')

column = pc.replace\_substring(column, ')', '')

# Trim whitespace

column = pc.utf8\_trim\_whitespace(column)

# Convert to float

column = pc.cast(column, pa.float64())

processed\_columns[column\_name] = column

# Process Qty

elif cleaned\_name == 'Qty':

print("Qty Found In Batch")

# Convert to float directly if possible

column = pc.cast(column, pa.float64())

processed\_columns[column\_name] = column

# Process other string columns

else:

# Fill null values with empty string

column = pc.fill\_null(column, '')

# Convert to string

column = pc.cast(column, pa.string())

# Replace 'None' with empty string

column = pc.replace\_substring(column, 'None', '')

# Trim whitespace

column = pc.utf8\_trim\_whitespace(column)

processed\_columns[column\_name] = column

# Create new table with processed columns

processed\_batch = pa.Table.from\_arrays(

[processed\_columns[name] for name in batch.schema.names],

names=batch.schema.names

)

batches.append(processed\_batch)

if batches:

result\_table = pa.concat\_tables(batches)

result\_df = result\_table.to\_pandas()

# Debug: Print sample of processed data

print("\nProcessed data sample:")

print(result\_df.head())

print("\nColumn dtypes:")

print(result\_df.dtypes)

return [result\_df.columns.tolist()] + result\_df.values.tolist()

return [["No data found"]]

except Exception as e:

print(f"Error in processing: {e}")

return [["Error: " + str(e)]]

Key changes:

Added a step to create cleaned column names: cleaned\_names = [name.strip() for name in batch.schema.names]

Use the cleaned names for comparison but keep original names for the dictionary keys

Changed the column name comparisons to use the cleaned names

Removed the .lower() since we know the exact case now

This should now correctly identify and process all columns, including 'Memo', 'Date', 'Amount', and 'Qty'. The sanitization of backslashes and other special characters should now work as expected.

But I found another issue also with amount in console: Error in processing: Failed to parse string: '1,000' as a scalar of type double

Ah yes, we need to handle the amount conversion more carefully. The issue is that we're trying to convert to float64 before removing the commas. Let's fix the Amount processing order:

python

# Process Amount

elif cleaned\_name == 'Amount':

print("Amount Found In Batch")

try:

# First convert to string to ensure we can do string operations

column = pc.cast(column, pa.string())

# Fill null values

column = pc.fill\_null(column, '$0.00')

# Remove $ and commas - do this before converting to float

column = pc.replace\_substring(column, '$', '')

column = pc.replace\_substring(column, ',', '')

# Handle parentheses for negative numbers

column = pc.replace\_substring(column, '(', '-')

column = pc.replace\_substring(column, ')', '')

# Trim whitespace

column = pc.utf8\_trim\_whitespace(column)

# Now try to convert to float

try:

column = pc.cast(column, pa.float64())

except:

# If conversion fails, print the problematic values

print("Problem values in Amount column:")

print(column.to\_pylist()[:10]) # Print first 10 values for debugging

raise

processed\_columns[column\_name] = column

except Exception as e:

print(f"Amount processing error: {e}")

# Fallback: keep original values if processing fails

processed\_columns[column\_name] = batch.column(column\_name)

Key changes:

Added explicit string conversion first

Added error handling to show problematic values

Added fallback to keep original values if processing fails

Added more detailed error reporting

This should handle amounts with commas properly now. If you still see any issues, the debug output will help us identify exactly which values are causing problems.

Let me know if you see any other issues or if you need additional adjustments to the amount processing logic.

I am going back to one of your earlier code suggestion with just one change that you have to make. Trim column names before processing. I think that code needs to work: @xw.func def process\_pyarrow\_chunks\_new(file\_path): import pyarrow as pa import pyarrow.csv as csv import pyarrow.compute as pc import pandas as pd import numpy as np # Define read options for PyArrow read\_options = csv.ReadOptions( block\_size=10 \* 1024 \* 1024, # 10MB chunks use\_threads=True ) parse\_options = csv.ParseOptions( delimiter=',', quote\_char='"' ) convert\_options = csv.ConvertOptions( strings\_can\_be\_null=True, include\_columns=None # Read all columns ) # Read the CSV file in batches batches = [] batch\_size = 500000 # Process in large batches batch\_count = 0 # Create a reader for the CSV file with csv.open\_csv( file\_path, read\_options=read\_options, parse\_options=parse\_options, convert\_options=convert\_options ) as reader: schema = reader.schema # Process each batch for batch in reader: batch\_count += 1 print(f"Processing batch {batch\_count}") # Get column indices for processing memo\_idx = schema.get\_field\_index('Memo') if 'Memo' in schema.names else -1 date\_idx = schema.get\_field\_index('Date') if 'Date' in schema.names else -1 qty\_idx = schema.get\_field\_index('Qty') if 'Qty' in schema.names else -1 amount\_idx = schema.get\_field\_index('Amount') if 'Amount' in schema.names else -1 # Convert to dictionary for column-wise processing batch\_dict = batch.to\_pydict() # Process Memo column - replace backslashes with spaces if memo\_idx >= 0: memo\_array = batch.column(memo\_idx) # Replace backslashes with spaces using PyArrow compute functions if memo\_array.null\_count < len(memo\_array): # Convert to string array if needed if not pa.types.is\_string(memo\_array.type): memo\_array = pc.cast(memo\_array, pa.string()) # Replace backslashes with spaces memo\_array = pc.replace\_substring\_regex(memo\_array, r'\\', ' ') batch\_dict['Memo'] = memo\_array.to\_pylist() # Process Date column if date\_idx >= 0: date\_array = batch.column(date\_idx) # Convert to pandas for date processing dates = pd.Series(date\_array.to\_pandas()) dates = pd.to\_datetime(dates, errors='coerce') # Convert to Excel date number excel\_dates = (dates - pd.Timestamp('1899-12-30')).dt.days batch\_dict['Date'] = excel\_dates.tolist() # Process numeric columns for col\_name, col\_idx in [('Qty', qty\_idx), ('Amount', amount\_idx)]: if col\_idx >= 0: # Get column as array col\_array = batch.column(col\_idx) # Convert to string for processing if not pa.types.is\_string(col\_array.type): col\_array = pc.cast(col\_array, pa.string()) # Clean numeric strings col\_array = pc.replace\_substring\_regex(col\_array, r',', '') col\_array = pc.replace\_substring\_regex(col\_array, r'\$', '') col\_array = pc.replace\_substring\_regex(col\_array, r'\(', '-') col\_array = pc.replace\_substring\_regex(col\_array, r'\)', '') # Convert to pandas for numeric conversion numeric\_values = pd.to\_numeric( pd.Series(col\_array.to\_pandas()), errors='coerce' ) batch\_dict[col\_name] = numeric\_values.tolist() # Create a new table from the processed dictionary processed\_batch = pa.Table.from\_pydict(batch\_dict) batches.append(processed\_batch) # Free memory del batch del batch\_dict # Combine all batches if batches: result\_table = pa.concat\_tables(batches) # Convert to pandas for output result\_df = result\_table.to\_pandas() # Convert to output format output\_data = [result\_df.columns.tolist()] + result\_df.values.tolist() return output\_data else: return [["No data found"]]

Yes, we can modify this code to handle the column names with trailing spaces. Here's the adjusted version focusing on the key parts that need to change:

python

@xw.func

def process\_pyarrow\_chunks\_new(file\_path):

import pyarrow as pa

import pyarrow.csv as csv

import pyarrow.compute as pc

import pandas as pd

import numpy as np

# [Previous options setup remains the same...]

# Create a reader for the CSV file

with csv.open\_csv(

file\_path,

read\_options=read\_options,

parse\_options=parse\_options,

convert\_options=convert\_options

) as reader:

schema = reader.schema

# Clean column names (remove trailing spaces)

cleaned\_names = {name.strip(): name for name in schema.names}

# Get column indices using cleaned names

memo\_idx = schema.get\_field\_index(cleaned\_names.get('Memo')) if 'Memo' in cleaned\_names else -1

date\_idx = schema.get\_field\_index(cleaned\_names.get('Date')) if 'Date' in cleaned\_names else -1

qty\_idx = schema.get\_field\_index(cleaned\_names.get('Qty')) if 'Qty' in cleaned\_names else -1

amount\_idx = schema.get\_field\_index(cleaned\_names.get('Amount')) if 'Amount' in cleaned\_names else -1

# Process each batch

for batch in reader:

batch\_count += 1

print(f"Processing batch {batch\_count}")

# Convert to dictionary for column-wise processing

batch\_dict = batch.to\_pydict()

# Process Memo column

if memo\_idx >= 0:

original\_memo\_name = cleaned\_names.get('Memo')

memo\_array = batch.column(memo\_idx)

if memo\_array.null\_count < len(memo\_array):

if not pa.types.is\_string(memo\_array.type):

memo\_array = pc.cast(memo\_array, pa.string())

memo\_array = pc.replace\_substring\_regex(memo\_array, r'\\', ' ')

batch\_dict[original\_memo\_name] = memo\_array.to\_pylist()

# Process Date column

if date\_idx >= 0:

original\_date\_name = cleaned\_names.get('Date')

date\_array = batch.column(date\_idx)

dates = pd.Series(date\_array.to\_pandas())

dates = pd.to\_datetime(dates, errors='coerce')

excel\_dates = (dates - pd.Timestamp('1899-12-30')).dt.days

batch\_dict[original\_date\_name] = excel\_dates.tolist()

# Process numeric columns

for col\_name, col\_idx in [('Qty', qty\_idx), ('Amount', amount\_idx)]:

if col\_idx >= 0:

original\_col\_name = cleaned\_names.get(col\_name)

col\_array = batch.column(col\_idx)

if not pa.types.is\_string(col\_array.type):

col\_array = pc.cast(col\_array, pa.string())

col\_array = pc.replace\_substring\_regex(col\_array, r',', '')

col\_array = pc.replace\_substring\_regex(col\_array, r'\$', '')

col\_array = pc.replace\_substring\_regex(col\_array, r'\(', '-')

col\_array = pc.replace\_substring\_regex(col\_array, r'\)', '')

numeric\_values = pd.to\_numeric(

pd.Series(col\_array.to\_pandas()),

errors='coerce'

)

batch\_dict[original\_col\_name] = numeric\_values.tolist()

# Create a new table from the processed dictionary

processed\_batch = pa.Table.from\_pydict(batch\_dict)

batches.append(processed\_batch)

# Free memory

del batch

del batch\_dict

# [Rest of the code remains the same...]

Key changes made:

Added a cleaned\_names dictionary that maps stripped column names to their original names

Modified the column index lookups to use the cleaned names for matching

Used the original column names when storing processed data back in the batch dictionary

This should now handle the column names with trailing spaces correctly while maintaining the original column names in the output. The processing logic remains the same, but the column name handling is more robust.

I want you to reproduce full code. You have ended like: # [Rest of the code remains the same...]

Here's the complete code:

python

@xw.func

def process\_pyarrow\_chunks\_new(file\_path):

import pyarrow as pa

import pyarrow.csv as csv

import pyarrow.compute as pc

import pandas as pd

import numpy as np

# Define read options for PyArrow

read\_options = csv.ReadOptions(

block\_size=10 \* 1024 \* 1024, # 10MB chunks

use\_threads=True

)

parse\_options = csv.ParseOptions(

delimiter=',',

quote\_char='"'

)

convert\_options = csv.ConvertOptions(

strings\_can\_be\_null=True,

include\_columns=None # Read all columns

)

# Read the CSV file in batches

batches = []

batch\_size = 500000 # Process in large batches

batch\_count = 0

# Create a reader for the CSV file

with csv.open\_csv(

file\_path,

read\_options=read\_options,

parse\_options=parse\_options,

convert\_options=convert\_options

) as reader:

schema = reader.schema

# Clean column names (remove trailing spaces)

cleaned\_names = {name.strip(): name for name in schema.names}

# Get column indices using cleaned names

memo\_idx = schema.get\_field\_index(cleaned\_names.get('Memo')) if 'Memo' in cleaned\_names else -1

date\_idx = schema.get\_field\_index(cleaned\_names.get('Date')) if 'Date' in cleaned\_names else -1

qty\_idx = schema.get\_field\_index(cleaned\_names.get('Qty')) if 'Qty' in cleaned\_names else -1

amount\_idx = schema.get\_field\_index(cleaned\_names.get('Amount')) if 'Amount' in cleaned\_names else -1

# Process each batch

for batch in reader:

batch\_count += 1

print(f"Processing batch {batch\_count}")

# Convert to dictionary for column-wise processing

batch\_dict = batch.to\_pydict()

# Process Memo column

if memo\_idx >= 0:

original\_memo\_name = cleaned\_names.get('Memo')

memo\_array = batch.column(memo\_idx)

if memo\_array.null\_count < len(memo\_array):

if not pa.types.is\_string(memo\_array.type):

memo\_array = pc.cast(memo\_array, pa.string())

memo\_array = pc.replace\_substring\_regex(memo\_array, r'\\', ' ')

batch\_dict[original\_memo\_name] = memo\_array.to\_pylist()

# Process Date column

if date\_idx >= 0:

original\_date\_name = cleaned\_names.get('Date')

date\_array = batch.column(date\_idx)

dates = pd.Series(date\_array.to\_pandas())

dates = pd.to\_datetime(dates, errors='coerce')

excel\_dates = (dates - pd.Timestamp('1899-12-30')).dt.days

batch\_dict[original\_date\_name] = excel\_dates.tolist()

# Process numeric columns

for col\_name, col\_idx in [('Qty', qty\_idx), ('Amount', amount\_idx)]:

if col\_idx >= 0:

original\_col\_name = cleaned\_names.get(col\_name)

col\_array = batch.column(col\_idx)

if not pa.types.is\_string(col\_array.type):

col\_array = pc.cast(col\_array, pa.string())

col\_array = pc.replace\_substring\_regex(col\_array, r',', '')

col\_array = pc.replace\_substring\_regex(col\_array, r'\$', '')

col\_array = pc.replace\_substring\_regex(col\_array, r'\(', '-')

col\_array = pc.replace\_substring\_regex(col\_array, r'\)', '')

numeric\_values = pd.to\_numeric(

pd.Series(col\_array.to\_pandas()),

errors='coerce'

)

batch\_dict[original\_col\_name] = numeric\_values.tolist()

# Create a new table from the processed dictionary

processed\_batch = pa.Table.from\_pydict(batch\_dict)

batches.append(processed\_batch)

# Free memory

del batch

del batch\_dict

# Combine all batches

if batches:

result\_table = pa.concat\_tables(batches)

# Convert to pandas for output

result\_df = result\_table.to\_pandas()

# Convert to output format

output\_data = [result\_df.columns.tolist()] + result\_df.values.tolist()

return output\_data

else:

return [["No data found"]]

This is the complete code with all the changes implemented, including:

The column name cleaning mechanism

Proper handling of trailing spaces

All the processing logic for Memo, Date, Amount, and Qty columns

The batch processing and memory management

The final conversion to the required output format

The code should now handle the column names with trailing spaces correctly while maintaining all the original functionality.