Working (LAST – CHANGES 3-04-2025, 12:00 PM)

Main.py :-

import tkinter as tk  
import pandas as pd  
from tkinter import filedialog, messagebox, ttk  
import ttkbootstrap as tb  
import os  
import logging  
import darkdetect  
import sys  
from datetime import datetime  
from docx import Document  
from file\_reader import read\_excel\_csv  
from data\_mapper import scan\_template\_placeholders, prepare\_row\_data, replace\_all\_placeholders  
from docx2pdf import convert  
  
*# Configure logging*logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")  
  
  
class DocumentFillerApp:  
 def \_\_init\_\_(*self*, *root*):  
 *self*.root = *root  
 self*.load\_default\_templates() *# Load templates first  
 self*.setup\_ui()  
 *self*.setup\_menu()  
  
 *# Initialize variables  
 self*.input\_file = None  
 *self*.output\_folder = None  
 *self*.current\_data = None  
  
 def load\_default\_templates(*self*):  
 *"""Load default templates from the templates folder"""* try:  
 *# Get the directory where the executable or script is located* if *getattr*(sys, 'frozen', False):  
 *# Running as compiled executable* application\_path = os.path.dirname(sys.executable)  
 else:  
 *# Running as script* application\_path = os.path.dirname(os.path.abspath(\_\_file\_\_))  
  
 templates\_dir = os.path.join(application\_path, "templates")  
  
 *self*.eligible\_template = os.path.join(templates\_dir, "eligible\_template.docx")  
 *self*.ineligible\_template = os.path.join(templates\_dir, "ineligible\_template.docx")  
  
 if not os.path.exists(*self*.eligible\_template):  
 raise *FileNotFoundError*(f"Eligible template not found at {*self*.eligible\_template}")  
 if not os.path.exists(*self*.ineligible\_template):  
 raise *FileNotFoundError*(f"Ineligible template not found at {*self*.ineligible\_template}")  
  
 logging.info("Default templates loaded successfully")  
  
 except *Exception* as e:  
 logging.error(f"Failed to load default templates: {*str*(e)}")  
 messagebox.showerror("Error", f"Failed to load default templates: {*str*(e)}")  
 *self*.root.destroy()  
  
 def setup\_ui(*self*):  
 *"""Setup the main user interface"""  
 self*.root.title("Automated ISD Document Generator")  
 *self*.root.geometry("1920x1080")  
  
 *# Main container* main\_frame = tb.Frame(*self*.root)  
 main\_frame.pack(fill=tk.BOTH, expand=True, padx=20, pady=20)  
  
 *# Left panel - Controls (store as self.control\_frame)  
 self*.control\_frame = tb.Frame(main\_frame)  
 *self*.control\_frame.pack(side=tk.LEFT, fill=tk.Y, padx=10, pady=10)  
  
 *# Control buttons* btn\_data = tb.Button(*self*.control\_frame, text="📂 Upload Data File",  
 command=*self*.upload\_data\_file)  
 btn\_data.pack(fill=tk.X, padx=10, pady=5)  
  
 btn\_output = tb.Button(*self*.control\_frame, text="📁 Select Output Folder",  
 command=*self*.select\_output\_folder)  
 btn\_output.pack(fill=tk.X, padx=10, pady=5)  
  
 btn\_start = tb.Button(*self*.control\_frame, text="🚀 Generate ISD Invoices",  
 bootstyle="success",  
 command=*self*.start\_processing)  
 btn\_start.pack(fill=tk.X, padx=10, pady=20)  
  
 *# Add progress bar components (hidden initially)  
 self*.progress\_frame = tb.Frame(*self*.control\_frame)  
 *self*.progress\_label = tb.Label(*self*.progress\_frame, text="Ready", bootstyle="info")  
 *self*.progress\_label.pack(fill=tk.X)  
  
 *self*.progress\_bar = tb.Progressbar(  
 *self*.progress\_frame,  
 orient="horizontal",  
 length=200,  
 mode="determinate",  
 bootstyle="success-striped"  
 )  
 *self*.progress\_bar.pack(fill=tk.X, pady=5)  
 *self*.progress\_frame.pack\_forget() *# Hide initially  
  
 # Template status labels  
 self*.lbl\_eligible\_template = tb.Label(*self*.control\_frame,  
 text=f"✅ Eligible Template: {os.path.basename(*self*.eligible\_template)}",  
 bootstyle="success")  
 *self*.lbl\_eligible\_template.pack(fill=tk.X, padx=10, pady=5)  
  
 *self*.lbl\_ineligible\_template = tb.Label(*self*.control\_frame,  
 text=f"✅ Ineligible Template: {os.path.basename(*self*.ineligible\_template)}",  
 bootstyle="success")  
 *self*.lbl\_ineligible\_template.pack(fill=tk.X, padx=10, pady=5)  
  
 *# Status labels  
 self*.lbl\_data = tb.Label(*self*.control\_frame, text="No Data File Loaded", bootstyle="secondary")  
 *self*.lbl\_data.pack(fill=tk.X, padx=10, pady=5)  
  
 *self*.lbl\_output = tb.Label(*self*.control\_frame, text="No Output Folder Selected", bootstyle="secondary")  
 *self*.lbl\_output.pack(fill=tk.X, padx=10, pady=5)  
  
 *# Right panel - Data Preview* preview\_frame = tb.Frame(main\_frame)  
 preview\_frame.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True, padx=10, pady=10)  
  
 preview\_label = tb.Label(preview\_frame, text="Data Preview", bootstyle="primary")  
 preview\_label.pack(fill=tk.X, pady=5)  
  
 *# Create the treeview with proper scrollbars  
 self*.tree = *self*.create\_treeview(preview\_frame)  
  
 def has\_tax\_amounts(*self*, *row*, *is\_eligible*):  
 *"""Check if row has any tax amounts for the given type (eligible/ineligible)"""* prefix = "ELIGIBLE\_" if *is\_eligible* else "INELIGIBLE\_"  
 *# Use the specific column names from your Excel structure* tax\_fields = ['CGST\_AS\_IGST', 'SGST\_AS\_IGST', 'CGST\_AS\_CGST', 'SGST\_UTGST\_AS\_SGST\_UTGST']  
  
 for tax in tax\_fields:  
 col\_name = prefix + tax  
 if col\_name in *row* and pd.notna(*row*[col\_name]):  
 try:  
 if *float*(*row*[col\_name]) > 0:  
 return True  
 except (*ValueError*, *TypeError*):  
 continue  
 return False  
  
 def start\_processing(*self*):  
 *"""Start the document generation process with organized output folders"""* if not *all*([*self*.input\_file, *self*.output\_folder]):  
 messagebox.showerror("Error", "Please select data file and output folder!")  
 return  
  
 try:  
 *# Show and initialize progress bar  
 self*.progress\_frame.pack(fill=tk.X, padx=10, pady=(20, 5))  
 *self*.progress\_bar['value'] = 0  
 *self*.progress\_label.config(text="Preparing...")  
 *self*.root.update\_idletasks()  
  
 data = read\_excel\_csv(*self*.input\_file)  
 if data is None:  
 messagebox.showerror("Error", "Failed to read data file.")  
 *self*.progress\_frame.pack\_forget()  
 return  
  
 *# Create main output folders* eligible\_folder = os.path.join(*self*.output\_folder, "Eligible")  
 ineligible\_folder = os.path.join(*self*.output\_folder, "Ineligible")  
 temp\_docx\_folder = os.path.join(*self*.output\_folder, "TEMP\_DOCX")  
  
 os.makedirs(eligible\_folder, exist\_ok=True)  
 os.makedirs(ineligible\_folder, exist\_ok=True)  
 os.makedirs(temp\_docx\_folder, exist\_ok=True)  
  
 total\_rows = *len*(data)  
 success\_count = 0  
  
 for idx, row in data.iterrows():  
 try:  
 *# Update progress* progress = (idx + 1) / total\_rows \* 100  
 *self*.progress\_bar['value'] = progress  
 *self*.progress\_label.config(text=f"Processing row {idx + 1} of {total\_rows}")  
 *self*.root.update\_idletasks()  
  
 logging.info(f"\nProcessing row {idx}:")  
 logging.info(  
 f"Eligible amounts - CGST: {row['ELIGIBLE\_CGST\_AS\_IGST']}, "  
 f"SGST: {row['ELIGIBLE\_SGST\_AS\_IGST']}, "  
 f"IGST: {row['ELIGIBLE\_IGST\_AS\_IGST']}"  
 )  
 logging.info(  
 f"Ineligible amounts - CGST: {row['INELIGIBLE\_CGST\_AS\_IGST']}, "  
 f"SGST: {row['INELIGIBLE\_SGST\_AS\_IGST']}, "  
 f"IGST: {row['INELIGIBLE\_IGST\_AS\_IGST']}"  
 )  
  
 *# Process both eligible and ineligible documents* for is\_eligible in [True, False]:  
 if not *self*.has\_tax\_amounts(row, is\_eligible):  
 logging.info(f"No {'eligible' if is\_eligible else 'ineligible'} amounts found")  
 continue  
  
 *# Set paths based on eligibility* if is\_eligible:  
 output\_pdf\_folder = eligible\_folder  
 prefix = "Eligible"  
 template\_path = *self*.eligible\_template  
 else:  
 output\_pdf\_folder = ineligible\_folder  
 prefix = "Ineligible"  
 template\_path = *self*.ineligible\_template  
  
 *# Generate document* doc = Document(template\_path)  
 placeholders = scan\_template\_placeholders(template\_path)  
 row\_data = prepare\_row\_data(row, placeholders, is\_eligible)  
  
 if not replace\_all\_placeholders(doc, row\_data):  
 logging.error(f"Skipping row {idx} due to replacement errors")  
 continue  
  
 *# Save temporary DOCX* invoice\_num = *str*(row.get('INVOICE\_NUMBER', idx + 1)).strip()  
 timestamp = datetime.now().strftime("%Y%m%d\_%H%M%S")  
 docx\_filename = f"{prefix}\_ISD\_{invoice\_num}\_{timestamp}.docx"  
 docx\_path = os.path.join(temp\_docx\_folder, docx\_filename)  
 doc.save(docx\_path)  
  
 *# Convert to PDF in appropriate folder* pdf\_filename = f"{prefix}\_ISD\_{invoice\_num}\_{timestamp}.pdf"  
 pdf\_path = os.path.join(output\_pdf\_folder, pdf\_filename)  
 convert(docx\_path, pdf\_path)  
  
 *# Delete temporary DOCX* os.remove(docx\_path)  
 success\_count += 1  
 logging.info(f"Generated {pdf\_filename}")  
  
 except *Exception* as e:  
 logging.error(f"Error processing row {idx}: {*str*(e)}", exc\_info=True)  
 continue  
  
 *# Clean up temporary folder* try:  
 os.rmdir(temp\_docx\_folder)  
 except *OSError*:  
 pass *# Folder not empty  
  
 # Final progress update  
 self*.progress\_bar['value'] = 100  
 *self*.progress\_label.config(text=f"Completed: {success\_count} documents generated")  
 *self*.root.update\_idletasks()  
  
 messagebox.showinfo("Success",  
 f"Processing complete!\n\n"  
 f"Eligible PDFs: {eligible\_folder}\n"  
 f"Ineligible PDFs: {ineligible\_folder}\n"  
 f"Total generated: {success\_count}")  
  
 except *Exception* as e:  
 if *hasattr*(*self*, 'progress\_label'):  
 *self*.progress\_label.config(text="Processing failed!", bootstyle="danger")  
 messagebox.showerror("Error", f"Processing failed: {*str*(e)}")  
 logging.error(f"Processing error: {*str*(e)}")  
  
 def is\_row\_eligible(*self*, *row*):  
 *"""Determine if row contains eligible or ineligible data"""* eligible\_cols = [  
 'ELIGIBLE\_IGST\_AS\_IGST', 'ELIGIBLE\_CGST\_AS\_IGST',  
 'ELIGIBLE\_SGST\_AS\_IGST', 'ELIGIBLE\_CGST\_AS\_CGST',  
 'ELIGIBLE\_SGST\_UTGST\_AS\_SGST\_UTGST'  
 ]  
  
 *# Check if any eligible tax amount is > 0* for col in eligible\_cols:  
 if col in *row*:  
 try:  
 val = *float*(*row*[col]) if pd.notna(*row*[col]) else 0  
 if val > 0:  
 return True  
 except (*ValueError*, *TypeError*):  
 continue  
 return False  
  
 def create\_treeview(*self*, *parent\_frame*):  
 *"""Create and configure the Treeview widget with proper scrollbars"""  
 # Container frame* container = tb.Frame(*parent\_frame*)  
 container.pack(fill=tk.BOTH, expand=True)  
  
 *# Treeview widget* tree = ttk.Treeview(container, selectmode="extended")  
  
 *# Vertical Scrollbar* yscroll = ttk.Scrollbar(container, orient="vertical", command=tree.yview)  
 yscroll.pack(side=tk.RIGHT, fill=tk.Y)  
 tree.configure(yscrollcommand=yscroll.set)  
  
 *# Horizontal Scrollbar* xscroll = ttk.Scrollbar(container, orient="horizontal", command=tree.xview)  
 xscroll.pack(side=tk.BOTTOM, fill=tk.X)  
 tree.configure(xscrollcommand=xscroll.set)  
  
 *# Pack treeview last* tree.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)  
  
 return tree  
  
 def on\_tree\_right\_click(*self*, *event*, *tree*):  
 *"""Right-click menu to auto-resize columns"""* menu = tk.Menu(*self*.root, tearoff=0)  
 menu.add\_command(label="Auto-size Columns", command=lambda: *self*.auto\_size\_columns(*tree*))  
 menu.post(*event*.x\_root, *event*.y\_root)  
  
 def auto\_resize\_columns(*self*):  
 *"""Automatically resize columns to fit content"""* for col in *self*.tree["columns"]:  
 *# Set minimum width based on header* min\_width = tk.font.Font().measure(col[:20]) + 20 *# Add padding  
  
 # Check all items for content width* for item in *self*.tree.get\_children():  
 cell\_value = *str*(*self*.tree.set(item, col))  
 cell\_width = tk.font.Font().measure(cell\_value[:30]) + 20 *# Limit check to 30 chars* if cell\_width > min\_width:  
 min\_width = cell\_width  
  
 *# Set final column width  
 self*.tree.column(col, width=min\_width)  
  
 def display\_data(*self*, *data*):  
 *"""Display data in Treeview using first row for column width reference"""  
 # Clear existing data  
 self*.tree.delete(\**self*.tree.get\_children())  
  
 *# Set up columns  
 self*.tree["columns"] = *list*(*data*.columns)  
 *self*.tree["show"] = "headings"  
  
 *# Add first row and use it for column width reference* if *len*(*data*) > 0:  
 first\_row = *data*.iloc[0]  
  
 *# Configure columns based on first row values* for col in *data*.columns:  
 *# Get header width* header\_width = tk.font.Font().measure(col) + 20 *# Add padding  
  
 # Get first row cell content width* cell\_value = *str*(first\_row[col])  
 cell\_width = tk.font.Font().measure(cell\_value) + 20 *# Add padding  
  
 # Use whichever is wider (header or first row content)* col\_width = *max*(header\_width, cell\_width)  
  
 *# Apply column configuration  
 self*.tree.heading(col, text=col)  
 *self*.tree.column(col, width=col\_width, stretch=False) *# Fixed width  
  
 # Insert all rows (first row will match our column widths)* for \_, row in *data*.iterrows():  
 *self*.tree.insert("", "end", values=*list*(row))  
 else:  
 *# Empty dataset - just set up columns* for col in *data*.columns:  
 *self*.tree.heading(col, text=col)  
 *self*.tree.column(col, width=tk.font.Font().measure(col) + 20, stretch=False)  
  
 *# Update the view  
 self*.tree.update\_idletasks()  
  
 def setup\_menu(*self*):  
 *"""Setup the menu bar"""* menu\_bar = tk.Menu(*self*.root)  
  
 *# File menu* file\_menu = tk.Menu(menu\_bar, tearoff=0)  
 file\_menu.add\_command(label="Upload Data File", command=*self*.upload\_data\_file)  
 file\_menu.add\_command(label="Exit", command=*self*.root.quit)  
 menu\_bar.add\_cascade(label="File", menu=file\_menu)  
  
 *# Theme menu* theme\_menu = tk.Menu(menu\_bar, tearoff=0)  
 theme\_options = {  
 "darkly": "🌙 Dark",  
 "journal": "📖 Light",  
 "flatly": "📄 Flat",  
 "cyborg": "🤖 Cyborg",  
 "superhero": "🦸 Superhero",  
 "minty": "🌿 Minty"  
 }  
  
 for theme, label in theme\_options.items():  
 theme\_menu.add\_command(label=label, command=lambda *t*=theme: *self*.change\_theme(t))  
  
 menu\_bar.add\_cascade(label="Theme", menu=theme\_menu)  
  
 *self*.root.config(menu=menu\_bar)  
  
 def change\_theme(*self*, *selected\_theme*):  
 *"""Change the application theme"""  
 self*.root.style.theme\_use(*selected\_theme*)  
  
 def upload\_data\_file(*self*):  
 *"""Handle data file upload"""* file\_path = filedialog.askopenfilename(filetypes=[("Excel/CSV files", "\*.xlsx;\*.xls;\*.csv")])  
 if file\_path:  
 *self*.input\_file = file\_path  
 *self*.lbl\_data.config(text=f"📂 {os.path.basename(file\_path)} Loaded")  
 logging.info(f"Data file loaded: {file\_path}")  
  
 try:  
 *self*.current\_data = read\_excel\_csv(file\_path)  
 if *self*.current\_data is not None:  
 *self*.display\_data(*self*.current\_data)  
 messagebox.showinfo("Success", "Data file loaded and displayed successfully!")  
 else:  
 messagebox.showerror("Error", "Failed to read data file.")  
 except *Exception* as e:  
 messagebox.showerror("Error", f"Failed to load data: {*str*(e)}")  
 logging.error(f"Data loading error: {*str*(e)}")  
  
 def select\_output\_folder(*self*):  
 *"""Handle output folder selection"""* folder = filedialog.askdirectory()  
 if folder:  
 *self*.output\_folder = folder  
 *self*.lbl\_output.config(text=f"📁 Output Folder: {folder}")  
 logging.info(f"Output folder selected: {folder}")  
  
  
*# Initialize and run the application*if \_\_name\_\_ == "\_\_main\_\_":  
 theme = "darkly" if darkdetect.isDark() else "journal"  
 root = tb.Window(themename=theme)  
 app = DocumentFillerApp(root)  
 root.mainloop()

Data\_Mapper.py :-  
  
import os  
import re  
import logging  
from docx import Document  
import pandas as pd  
from typing import List, Optional, Set, Dict  
from datetime import datetime  
from copy import deepcopy  
from num2words import num2words  
from docx.shared import Pt  
from typing import Dict  
  
logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")  
  
*# Enhanced column mapping with both eligible and ineligible tax fields*COLUMN\_MAPPING = {  
 *# Invoice fields* 'invoicenumber': 'INVOICE\_NUMBER',  
 'invoicedate': 'INVOICE\_DATE',  
  
 *# ISD Distributor fields* 'isddistributorgstin': 'ISD\_DISTRIBUTOR\_GSTIN',  
 'isddistributorname': 'ISD\_DISTRIBUTOR\_NAME',  
 'isddistributoraddress': 'ISD\_DISTRIBUTOR\_ADDRESS',  
 'isddistributorstate': 'ISD\_DISTRIBUTOR\_STATE',  
 'isddistributorpincode': 'ISD\_DISTRIBUTOR\_PINCODE',  
 'isddistributorstatecode': 'ISD\_DISTRIBUTOR\_STATE\_CODE',  
  
 *# Credit Recipient fields* 'creditrecipientgstin': 'CREDIT\_RECIPIENT\_GSTIN',  
 'creditrecipientname': 'CREDIT\_RECIPIENT\_NAME',  
 'creditrecipientaddress': 'CREDIT\_RECIPIENT\_ADDRESS',  
 'creditrecipientstate': 'CREDIT\_RECIPIENT\_STATE',  
 'creditrecipientpincode': 'CREDIT\_RECIPIENT\_PINCODE',  
 'creditrecipientstatecode': 'CREDIT\_RECIPIENT\_STATE\_CODE',  
  
 *# Tax fields - Handle both eligible and ineligible  
 # Eligible Tax fields* 'eligibleigstasigst': 'ELIGIBLE\_IGST\_AS\_IGST',  
 'eligiblecgstasigst': 'ELIGIBLE\_CGST\_AS\_IGST',  
 'eligiblesgstasigst': 'ELIGIBLE\_SGST\_AS\_IGST',  
 'eligibleigstsum': 'ELIGIBLE\_IGST\_SUM',  
 'eligiblecgstascgst': 'ELIGIBLE\_CGST\_AS\_CGST',  
 'eligiblecgstsum': 'ELIGIBLE\_CGST\_SUM',  
 'eligiblesgstutgstassgstutgst': 'ELIGIBLE\_SGST\_UTGST\_AS\_SGST\_UTGST',  
 'eligiblesgstutgstassgstutgstsum': 'ELIGIBLE\_SGST\_UTGST\_SUM',  
 'eligibleamount': 'ELIGIBLE\_AMOUNT',  
 *# Ineligible Tax fields* 'ineligibleigstasigst': 'INELIGIBLE\_IGST\_AS\_IGST',  
 'ineligiblecgstasigst': 'INELIGIBLE\_CGST\_AS\_IGST',  
 'ineligiblesgstasigst': 'INELIGIBLE\_SGST\_AS\_IGST',  
 'ineligibleigstsum': 'INELIGIBLE\_IGST\_SUM',  
 'ineligiblecgstascgst': 'INELIGIBLE\_CGST\_AS\_CGST',  
 'ineligiblecgstsum': 'INELIGIBLE\_CGST\_SUM',  
 'ineligiblesgstutgstassgstutgst': 'INELIGIBLE\_SGST\_UTGST\_AS\_SGST\_UTGST',  
 'ineligiblesgstutgstassgstutgstsum': 'INELIGIBLE\_SGST\_UTGST\_SUM',  
 'ineligibleamount': 'INELIGIBLE\_AMOUNT',  
 'cgst': 'CGST', *# Fallback* 'sgst': 'SGST', *# Fallback* 'utgst': 'UTGST', *# Fallback* 'igst': 'IGST', *# Fallback  
  
 # Amount fields* 'amount': 'AMOUNT',  
 'total': 'AMOUNT',  
  
 *# Contact fields* 'regoffice': 'REG\_OFFICE',  
 'cin': 'CIN',  
 'e-mail': 'E\_MAIL',  
 'website': 'WEBSITE',  
  
 *# Special fields* 'amount\_in\_words': 'AMOUNT\_IN\_WORDS'  
}  
  
  
def normalize\_column\_names(*df*: pd.DataFrame) -> pd.DataFrame:  
 *"""Enhanced column name normalization"""  
 df*.columns = [  
 col.strip().upper()  
 .replace(' ', '\_')  
 .replace('-', '\_')  
 .replace('.', '')  
 .replace('ELIGABLE', 'ELIGIBLE') *# Fix common typo* for col in *df*.columns  
 ]  
 return *df*def map\_data\_to\_docx(*template\_path*: *str*, *data*: pd.DataFrame, *output\_folder*: *str*,  
 *is\_eligible*: *bool* = True) -> Optional[List[*str*]]:  
 *"""  
 Main function to generate DOCX files with template selection  
 Args:  
 template\_path: Path to the template file  
 data: DataFrame containing the data  
 output\_folder: Output directory for generated files  
 is\_eligible: Boolean indicating whether to use eligible template  
 """* try:  
 if not validate\_inputs(*template\_path*, *data*, *output\_folder*):  
 return None  
  
 os.makedirs(*output\_folder*, exist\_ok=True)  
 generated\_files = []  
 template\_placeholders = scan\_template\_placeholders(*template\_path*)  
  
 logging.info(f"Processing {*len*(*data*)} rows with {'eligible' if *is\_eligible* else 'ineligible'} template")  
  
 for idx, row in *data*.iterrows():  
 try:  
 doc = Document(*template\_path*)  
 row\_data = prepare\_row\_data(row, template\_placeholders, *is\_eligible*)  
  
 if idx == 0: *# Debug info for first row* log\_debug\_info(row, template\_placeholders, row\_data)  
  
 if not replace\_all\_placeholders(doc, row\_data):  
 logging.error(f"Skipping row {idx} due to replacement errors")  
 continue  
  
 output\_path = generate\_output\_path(*output\_folder*, row\_data, idx, *is\_eligible*)  
 doc.save(output\_path)  
 generated\_files.append(output\_path)  
 logging.info(f"Generated: {os.path.basename(output\_path)}")  
  
 except *Exception* as e:  
 logging.error(f"Error processing row {idx}: {*str*(e)}", exc\_info=True)  
 continue  
  
 return generated\_files if generated\_files else None  
  
 except *Exception* as e:  
 logging.error(f"Fatal error in document generation: {*str*(e)}", exc\_info=True)  
 return None  
  
  
def validate\_inputs(*template\_path*: *str*, *data*: pd.DataFrame, *output\_folder*: *str*) -> *bool*:  
 *"""Validate all input parameters"""* if not os.path.exists(*template\_path*):  
 logging.error(f"Template file not found: {*template\_path*}")  
 return False  
  
 if *data*.empty:  
 logging.error("No data provided in DataFrame")  
 return False  
  
 try:  
 os.makedirs(*output\_folder*, exist\_ok=True)  
 return True  
 except *Exception* as e:  
 logging.error(f"Output folder not writable: {*str*(e)}")  
 return False  
  
  
def prepare\_row\_data(*row*, *template\_placeholders*=None, *is\_eligible*=True):  
 prefix = "ELIGIBLE\_" if *is\_eligible* else "INELIGIBLE\_"  
 row\_data = {}  
  
 *# Tax components mapping* tax\_mapping = {  
 'IGST\_AS\_IGST': f'{prefix}IGST\_AS\_IGST',  
 'CGST\_AS\_IGST': f'{prefix}CGST\_AS\_IGST',  
 'SGST\_AS\_IGST': f'{prefix}SGST\_AS\_IGST',  
 'CGST\_AS\_CGST': f'{prefix}CGST\_AS\_CGST',  
 'SGST\_UTGST\_AS\_SGST\_UTGST': f'{prefix}SGST\_UTGST\_AS\_SGST\_UTGST',  
 'IGST\_SUM': f'{prefix}IGST\_SUM',  
 'CGST\_SUM': f'{prefix}CGST\_SUM',  
 'SGST\_UTGST\_SUM': f'{prefix}SGST\_UTGST\_SUM',  
 'AMOUNT': f'{prefix}AMOUNT'  
 }  
  
 *# First process all tax components* for placeholder, col in tax\_mapping.items():  
 if col in *row*:  
 row\_data[placeholder] = format\_value(*row*[col], placeholder)  
 else:  
 row\_data[placeholder] = "0.00" *# Default value if missing  
  
 # Calculate total amount if not provided or zero* if 'AMOUNT' not in row\_data or *float*(row\_data['AMOUNT'].replace(',', '')) == 0:  
 try:  
 total = *sum*([  
 *float*(*row*.get(f'{prefix}IGST\_SUM', 0)),  
 *float*(*row*.get(f'{prefix}CGST\_SUM', 0)),  
 *float*(*row*.get(f'{prefix}SGST\_UTGST\_SUM', 0))  
 ])  
 row\_data['AMOUNT'] = format\_value(total, 'AMOUNT')  
 except *Exception* as e:  
 logging.error(f"Error calculating total amount: {*str*(e)}")  
 row\_data['AMOUNT'] = "0.00"  
  
 *# Common fields mapping* common\_fields = {  
 *# Invoice fields* 'Invoice Number': 'INVOICE\_NUMBER',  
 'Invoice Date': 'INVOICE\_DATE',  
 *# ISD Distributor fields* 'ISD Distributor Name': 'ISD\_DISTRIBUTOR\_NAME',  
 'ISD Distributor Address': 'ISD\_DISTRIBUTOR\_ADDRESS',  
 'ISD Distributor State': 'ISD\_DISTRIBUTOR\_STATE',  
 'ISD Distributor Pincode': 'ISD\_DISTRIBUTOR\_PINCODE',  
 'ISD Distributor State Code': 'ISD\_DISTRIBUTOR\_STATE\_CODE',  
 'ISD Distributor GSTIN': 'ISD\_DISTRIBUTOR\_GSTIN',  
 *# Credit Recipient fields* 'Credit Recipient Name': 'CREDIT\_RECIPIENT\_NAME',  
 'Credit Recipient Address': 'CREDIT\_RECIPIENT\_ADDRESS',  
 'Credit Recipient State': 'CREDIT\_RECIPIENT\_STATE',  
 'Credit Recipient Pincode': 'CREDIT\_RECIPIENT\_PINCODE',  
 'Credit Recipient State Code': 'CREDIT\_RECIPIENT\_STATE\_CODE',  
 'Credit Recipient GSTIN': 'CREDIT\_RECIPIENT\_GSTIN',  
 *# Contact fields* 'Reg. Office': 'REG\_OFFICE',  
 'CIN': 'CIN',  
 'E-Mail': 'E\_MAIL',  
 'Website': 'WEBSITE',  
 *# Special fields* 'Amount\_In\_Words': 'AMOUNT\_IN\_WORDS'  
 }  
  
 *# Process common fields* for placeholder, col in common\_fields.items():  
 if col in *row*:  
 row\_data[placeholder] = format\_value(*row*[col], placeholder)  
  
 *# Generate amount in words if needed* if *any*('amount\_in\_words' in ph.lower() for ph in (*template\_placeholders* or [])):  
 try:  
 amount\_str = row\_data.get('AMOUNT', '0').replace(',', '')  
 amount = *float*(amount\_str)  
  
 if amount % 1 == 0:  
 words = num2words(*int*(amount), lang='en\_IN').title()  
 row\_data['amount\_in\_words'] = f"{words} Rupees Only"  
 else:  
 rupees = *int*(amount)  
 paise = *round*((amount - rupees) \* 100)  
 rupee\_words = num2words(rupees, lang='en\_IN').title()  
 paise\_words = num2words(paise, lang='en\_IN').title()  
 row\_data['amount\_in\_words'] = f"{rupee\_words} Rupees and {paise\_words} Paise Only"  
 except *Exception* as e:  
 logging.error(f"Error converting amount to words: {*str*(e)}")  
 row\_data['amount\_in\_words'] = "Rupees Only"  
  
 return row\_data  
  
def safe\_float\_conversion(*value*):  
 *"""Safely convert values to float, handling various edge cases"""* if pd.isna(*value*) or *value* in ['', None]:  
 return 0.0  
 try:  
 return *float*(*value*)  
 except (*ValueError*, *TypeError*):  
 return 0.0  
  
def replace\_all\_placeholders(*doc*: Document, *row\_data*: Dict[*str*, *str*]) -> *bool*:  
 *"""Replace placeholders throughout document with formatting preservation"""* try:  
 logging.info(f"Available placeholders in row\_data: {*list*(*row\_data*.keys())}")  
 logging.info("\n=== Placeholder Replacement ===")  
 logging.info(f"Amount value: {*row\_data*.get('AMOUNT', 'MISSING')}")  
 logging.info(f"Amount in words: {*row\_data*.get('amount\_in\_words', 'MISSING')}")  
  
 *# Rest of the function remains the same* for paragraph in *doc*.paragraphs:  
 replace\_in\_paragraph(paragraph, *row\_data*)  
  
 *# Process all tables* for table in *doc*.tables:  
 for row in table.rows:  
 for cell in row.cells:  
 for paragraph in cell.paragraphs:  
 replace\_in\_paragraph(paragraph, *row\_data*)  
  
 *# Process headers and footers* for section in *doc*.sections:  
 for header in [section.header, section.first\_page\_header]:  
 if header:  
 for paragraph in header.paragraphs:  
 replace\_in\_paragraph(paragraph, *row\_data*)  
  
 for footer in [section.footer, section.first\_page\_footer]:  
 if footer:  
 for paragraph in footer.paragraphs:  
 replace\_in\_paragraph(paragraph, *row\_data*)  
  
 return True  
  
 except *Exception* as e:  
 logging.error(f"Error replacing placeholders: {*str*(e)}", exc\_info=True)  
 return False  
  
  
def replace\_in\_paragraph(*paragraph*, *row\_data*):  
 *# First combine all runs* full\_text = ''.join(run.text for run in *paragraph*.runs)  
  
 *# Skip if no replacements needed* if not *any*(ph in full\_text for ph in *row\_data*):  
 return  
  
 *# Clear existing content  
 paragraph*.clear()  
  
 *# Split text by placeholders* parts = re.split(r'(\{\{.+?\}\})', full\_text)  
  
 for part in parts:  
 if part.startswith('{{') and part.endswith('}}'):  
 *# This is a placeholder* ph = part[2:-2].strip() *# Remove braces* value = *str*(*row\_data*.get(ph, part)) *# Get value or keep original if not found* run = *paragraph*.add\_run(value)  
 else:  
 *# Regular text* run = *paragraph*.add\_run(part)  
  
 *# Preserve original formatting* if *paragraph*.runs and *paragraph*.runs[0].font.name:  
 run.font.name = *paragraph*.runs[0].font.name  
 run.font.size = Pt(10)  
  
  
def format\_value(*value*, *key*=None) -> *str*:  
 *"""Enhanced value formatting with special cases"""* if pd.isna(*value*) or *value* in ['', None]:  
 return ""  
  
 *# Handle numpy types* if *hasattr*(*value*, 'item'):  
 value = *value*.item()  
  
 *# Format amounts with 2 decimal places* if *key* and *any*(x in *str*(*key*).lower() for x in ['amount', 'igst', 'cgst', 'sgst']):  
 try:  
 return "{:,.2f}".format(*float*(*value*))  
 except:  
 return *str*(*value*)  
  
 *# Special formatting for amounts* if *key* and 'amount' in *key*.lower() and *isinstance*(*value*, (*int*, *float*)):  
 return "{:,.2f}".format(*value*)  
  
 *# Special handling for GSTIN (format with spaces)* if *key* and 'gstin' in *key*.lower() and *isinstance*(*value*, *str*) and *len*(*value*) == 15:  
 return f"{*value*[:2]} {*value*[2:5]} {*value*[5:7]} {*value*[7:12]} {*value*[12:15]}"  
  
 return *str*(*value*).strip()  
  
  
def scan\_template\_placeholders(*template\_path*: *str*) -> Set[*str*]:  
 *"""  
 Scan a DOCX template and extract all unique placeholder variables  
 Args:  
 template\_path: Path to the template DOCX file  
 Returns:  
 Set of all unique placeholder names found in the template  
 """* placeholders = *set*()  
 try:  
 doc = Document(*template\_path*)  
  
 *# Pattern to match {{placeholder}} but ignore \*\*bold\*\* markers* placeholder\_pattern = re.compile(r'\{\{\s\*([^{}]+?)\s\*\}\}(?!\\*)')  
  
 def extract\_placeholders(*text*: *str*):  
 return {match.group(1).strip()  
 for match in placeholder\_pattern.finditer(*text*)}  
  
 *# Check all paragraphs in main document* for paragraph in doc.paragraphs:  
 placeholders.update(extract\_placeholders(paragraph.text))  
  
 *# Check tables* for table in doc.tables:  
 for row in table.rows:  
 for cell in row.cells:  
 for paragraph in cell.paragraphs:  
 placeholders.update(extract\_placeholders(paragraph.text))  
  
 *# Check headers and footers* for section in doc.sections:  
 for header in [section.header, section.first\_page\_header]:  
 if header:  
 for paragraph in header.paragraphs:  
 placeholders.update(extract\_placeholders(paragraph.text))  
  
 for footer in [section.footer, section.first\_page\_footer]:  
 if footer:  
 for paragraph in footer.paragraphs:  
 placeholders.update(extract\_placeholders(paragraph.text))  
  
 *# Check for placeholders in runs (in case they're split across runs)* for paragraph in doc.paragraphs:  
 full\_text = ''.join(run.text for run in paragraph.runs)  
 placeholders.update(extract\_placeholders(full\_text))  
  
 logging.info(f"Found placeholders in template: {placeholders}")  
 return placeholders  
  
 except *Exception* as e:  
 logging.error(f"Error scanning template placeholders: {*str*(e)}")  
 return *set*()  
  
  
def generate\_output\_path(*output\_folder*: *str*, *row\_data*: *dict*, *idx*: *int*,  
 *is\_eligible*: *bool*) -> *str*:  
 *"""Generate output path with type prefix and invoice number"""* invoice\_num = *str*(*row\_data*.get('INVOICE\_NUMBER', *idx* + 1)).strip()  
 prefix = "ELIGIBLE" if *is\_eligible* else "INELIGIBLE"  
 timestamp = datetime.now().strftime("%Y%m%d\_%H%M%S")  
 return os.path.join(*output\_folder*, f"{prefix}\_ISD\_{invoice\_num}\_{timestamp}.docx")  
  
  
def log\_debug\_info(*row*, *template\_placeholders*, *row\_data*):  
 *"""Enhanced debug logging with more details"""* logging.info("\n=== DEBUG INFORMATION ===")  
 logging.info(f"Template placeholders: {*sorted*(*template\_placeholders*)}")  
 logging.info(f"Data columns: {*sorted*(*row*.index.tolist())}")  
  
 logging.info("\n=== PLACEHOLDER MAPPING ===")  
 for ph in *sorted*(*template\_placeholders*):  
 norm\_ph = ph.lower().replace(' ', '').replace('.', '').replace('-', '')  
 data\_key = COLUMN\_MAPPING.get(norm\_ph, "NO MATCH")  
 logging.info(f"Template: {ph:25} → Data: {data\_key}")  
  
 logging.info("\n=== MATCHED DATA ===")  
 for ph, value in *sorted*(*row\_data*.items()):  
 logging.info(f"{ph:25}: {value}")  
 logging.info("=====================")  
  
def validate\_template(*template\_path*, *required\_placeholders*):  
 doc = Document(*template\_path*)  
 found\_placeholders = scan\_template\_placeholders(*template\_path*)  
 missing = [ph for ph in *required\_placeholders* if ph not in found\_placeholders]  
 if missing:  
 raise *ValueError*(f"Missing placeholders in template: {missing}")

Docx\_Filler.py :-  
  
import os  
import re  
import logging  
from docx import Document  
from docx.shared import Pt  
from typing import Dict, List, Optional  
from datetime import datetime  
  
logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")  
  
*# Hardcoded bold elements*BOLD\_ELEMENTS = {  
 'invoicenumber',  
 'invoicedate',  
 'Details of ISD Distributor: -',  
 'Details of Credit Recipient: -',  
 'Name:',  
 'Adress:',  
 'Pin code:',  
 'State Name:',  
 'State code:',  
 'GSTIN:'  
}  
  
  
def fill\_docx\_template(*template\_path*: *str*, *output\_path*: *str*, *replacements*: Dict[*str*, *str*]) -> *bool*:  
 *"""Fill template with values and apply hardcoded bold formatting"""* try:  
 doc = Document(*template\_path*)  
  
 *# Process all paragraphs* for paragraph in doc.paragraphs:  
 process\_paragraph(paragraph, *replacements*)  
  
 *# Process tables* for table in doc.tables:  
 for row in table.rows:  
 for cell in row.cells:  
 for paragraph in cell.paragraphs:  
 process\_paragraph(paragraph, *replacements*)  
  
 doc.save(*output\_path*)  
 return True  
 except *Exception* as e:  
 logging.error(f"Error: {*str*(e)}")  
 return False  
  
  
def process\_paragraph(*paragraph*, *replacements*):  
 *"""Process paragraph with hardcoded bold formatting"""* original\_text = *paragraph*.text  
 if not original\_text:  
 return  
  
 *# Clear existing content  
 paragraph*.clear()  
  
 *# Split text into parts that need bold formatting* parts = re.split(r'(' + '|'.join(*map*(re.escape, BOLD\_ELEMENTS)) + r')', original\_text)  
  
 for part in parts:  
 if not part:  
 continue  
  
 run = *paragraph*.add\_run(part)  
  
 *# Apply bold if part matches our hardcoded elements* if part in BOLD\_ELEMENTS:  
 run.bold = True  
  
 *# Replace placeholders if they exist in this part* for ph, value in *replacements*.items():  
 if ph in part:  
 run.text = run.text.replace(ph, *str*(value))  
  
  
def replace\_all\_placeholders(*doc*: Document, *row\_data*: Dict[*str*, *str*]) -> *bool*:  
 *"""Replace placeholders throughout document with hardcoded bold elements"""* try:  
 *# Process all document components* components = [  
 *doc*.paragraphs,  
 \*[cell.paragraphs for table in *doc*.tables  
 for row in table.rows  
 for cell in row.cells],  
 \*[section.header.paragraphs for section in *doc*.sections],  
 \*[section.footer.paragraphs for section in *doc*.sections]  
 ]  
  
 for paragraphs in components:  
 for paragraph in paragraphs:  
 process\_paragraph(paragraph, *row\_data*)  
  
 return True  
 except *Exception* as e:  
 logging.error(f"Error replacing placeholders: {*str*(e)}")  
 return False  
  
  
def scan\_template\_placeholders(*template\_path*: *str*) -> Set[*str*]:  
 *"""  
 Scan a DOCX template and extract all unique placeholder variables  
 Args:  
 template\_path: Path to the template DOCX file  
 Returns:  
 Set of all unique placeholder names found in the template  
 """* doc = Document(*template\_path*)  
 placeholders = *set*()  
 *# Match both {{ }} and {[ ]} styles, and clean the names* pattern = re.compile(r'\{\{?\s\*([^{}]+?)\s\*\}?\}')  
  
 def scan\_text(*text*: *str*):  
 return {match.group(1).strip() for match in pattern.finditer(*text*)}  
  
 *# Check all document components* components = [  
 doc.paragraphs,  
 \*[cell.paragraphs for table in doc.tables  
 for row in table.rows  
 for cell in row.cells],  
 \*[section.header.paragraphs for section in doc.sections],  
 \*[section.footer.paragraphs for section in doc.sections]  
 ]  
  
 for paragraphs in components:  
 for paragraph in paragraphs:  
 placeholders.update(scan\_text(paragraph.text))  
  
 return placeholders  
  
  
def generate\_output\_filename(*row\_data*: Dict, *idx*: *int*, *is\_eligible*: *bool*) -> *str*:  
 *"""  
 Generate a standardized output filename  
 Args:  
 row\_data: Dictionary containing row data  
 idx: Row index  
 is\_eligible: Whether this is an eligible invoice  
 Returns:  
 str: Generated filename  
 """* invoice\_num = *str*(*row\_data*.get('INVOICE\_NUMBER', *idx* + 1)).strip()  
 prefix = "Eligible" if *is\_eligible* else "Ineligible"  
 timestamp = datetime.now().strftime("%Y%m%d\_%H%M%S")  
 return f"{prefix}\_ISD\_{invoice\_num}\_{timestamp}.docx"  
  
  
def validate\_template(*template\_path*: *str*) -> *bool*:  
 *"""  
 Validate that the template exists and is accessible  
 Args:  
 template\_path: Path to the template file  
 Returns:  
 bool: True if valid, False otherwise  
 """* try:  
 if not os.path.exists(*template\_path*):  
 logging.error(f"Template file not found: {*template\_path*}")  
 return False  
 *# Try opening the document to verify it's valid* Document(*template\_path*)  
 return True  
 except *Exception* as e:  
 logging.error(f"Invalid template file: {*str*(e)}")  
 return False

File\_Reader.py :-

import os  
import re  
import pandas as pd  
import logging  
from typing import Optional  
  
logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")  
  
  
def clean\_data(*df*: pd.DataFrame) -> pd.DataFrame:  
 *"""  
 Cleans and normalizes the loaded DataFrame.  
  
 Args:  
 df: Raw pandas DataFrame  
  
 Returns:  
 Cleaned DataFrame with normalized column names and values  
 """  
 # Normalize column names  
 df*.columns = [  
 col.strip()  
 .upper()  
 .replace(' ', '\_')  
 .replace('-', '\_')  
 .replace('.', '')  
 for col in *df*.columns  
 ]  
  
 *# Clean string values* for col in *df*.columns:  
 if *df*[col].dtype == 'object':  
 *df*[col] = *df*[col].str.strip()  
 *df*[col] = *df*[col].replace({'': None, 'nan': None, 'None': None})  
  
 *# Convert amount columns to numeric* amount\_cols = ['AMOUNT', 'CGST', 'SGST', 'UTGST', 'IGST']  
 for col in amount\_cols:  
 if col in *df*.columns:  
 *df*[col] = pd.to\_numeric(*df*[col], errors='coerce')  
  
 return *df*def validate\_row(*row*):  
 *"""Validate individual row data for required fields and GSTIN format"""* required\_fields = ['INVOICE\_NUMBER', 'INVOICE\_DATE', 'ISD\_DISTRIBUTOR\_GSTIN']  
 missing = [field for field in required\_fields if pd.isna(*row*.get(field))]  
 if missing:  
 raise *ValueError*(f"Missing required fields: {missing}")  
  
 *# Validate GSTIN format* gstin\_fields = ['ISD\_DISTRIBUTOR\_GSTIN', 'CREDIT\_RECIPIENT\_GSTIN']  
 for field in gstin\_fields:  
 if field in *row* and not re.match(r'^[0-9]{2}[A-Z]{5}[0-9]{4}[A-Z]{1}[1-9A-Z]{1}Z[0-9A-Z]{1}$', *str*(*row*[field])):  
 logging.warning(f"Invalid GSTIN format in {field}: {*row*[field]}")  
  
  
def read\_excel\_csv(*file\_path*: *str*) -> Optional[pd.DataFrame]:  
 try:  
 if not os.path.exists(*file\_path*):  
 logging.error(f"File not found: {*file\_path*}")  
 return None  
  
 file\_ext = os.path.splitext(*file\_path*)[1].lower()  
  
 if file\_ext in ['.xlsx', '.xls']:  
 *# Read the Excel file with data\_only=True to evaluate formulas* df = pd.read\_excel(  
 *file\_path*,  
 engine='openpyxl',  
 header=0,  
 skiprows=[1], *# Skip the tax type labels row* names=[  
 *# Basic invoice info* 'INVOICE\_NUMBER', 'INVOICE\_DATE',  
 *# ISD Distributor info* 'ISD\_DISTRIBUTOR\_GSTIN', 'ISD\_DISTRIBUTOR\_NAME',  
 'ISD\_DISTRIBUTOR\_ADDRESS', 'ISD\_DISTRIBUTOR\_STATE',  
 'ISD\_DISTRIBUTOR\_PINCODE', 'ISD\_DISTRIBUTOR\_STATE\_CODE',  
 *# Credit Recipient info* 'CREDIT\_RECIPIENT\_GSTIN', 'CREDIT\_RECIPIENT\_NAME',  
 'CREDIT\_RECIPIENT\_ADDRESS', 'CREDIT\_RECIPIENT\_STATE',  
 'CREDIT\_RECIPIENT\_PINCODE', 'CREDIT\_RECIPIENT\_STATE\_CODE',  
 *# Eligible tax breakdown* 'ELIGIBLE\_IGST\_AS\_IGST', 'ELIGIBLE\_CGST\_AS\_IGST',  
 'ELIGIBLE\_SGST\_AS\_IGST', 'ELIGIBLE\_IGST\_SUM',  
 'ELIGIBLE\_CGST\_AS\_CGST', 'ELIGIBLE\_CGST\_SUM',  
 'ELIGIBLE\_SGST\_UTGST\_AS\_SGST\_UTGST', 'ELIGIBLE\_SGST\_UTGST\_SUM',  
 'ELIGIBLE\_AMOUNT',  
 *# Ineligible tax breakdown* 'INELIGIBLE\_IGST\_AS\_IGST', 'INELIGIBLE\_CGST\_AS\_IGST',  
 'INELIGIBLE\_SGST\_AS\_IGST', 'INELIGIBLE\_IGST\_SUM',  
 'INELIGIBLE\_CGST\_AS\_CGST', 'INELIGIBLE\_CGST\_SUM',  
 'INELIGIBLE\_SGST\_UTGST\_AS\_SGST\_UTGST', 'INELIGIBLE\_SGST\_UTGST\_SUM',  
 'INELIGIBLE\_AMOUNT',  
 *# Contact info* 'REG\_OFFICE', 'CIN', 'E\_MAIL', 'WEBSITE'  
 ],  
 dtype=*str*,  
 na\_values=['', 'NA', 'N/A', 'NULL'],  
 keep\_default\_na=False  
 )  
  
 *# Convert numeric columns to float* numeric\_cols = [  
 'ELIGIBLE\_IGST\_AS\_IGST', 'ELIGIBLE\_CGST\_AS\_IGST', 'ELIGIBLE\_SGST\_AS\_IGST',  
 'ELIGIBLE\_IGST\_SUM', 'ELIGIBLE\_CGST\_AS\_CGST', 'ELIGIBLE\_CGST\_SUM',  
 'ELIGIBLE\_SGST\_UTGST\_AS\_SGST\_UTGST', 'ELIGIBLE\_SGST\_UTGST\_SUM', 'ELIGIBLE\_AMOUNT',  
 'INELIGIBLE\_IGST\_AS\_IGST', 'INELIGIBLE\_CGST\_AS\_IGST', 'INELIGIBLE\_SGST\_AS\_IGST',  
 'INELIGIBLE\_IGST\_SUM', 'INELIGIBLE\_CGST\_AS\_CGST', 'INELIGIBLE\_CGST\_SUM',  
 'INELIGIBLE\_SGST\_UTGST\_AS\_SGST\_UTGST', 'INELIGIBLE\_SGST\_UTGST\_SUM', 'INELIGIBLE\_AMOUNT'  
 ]  
  
 for col in numeric\_cols:  
 df[col] = pd.to\_numeric(df[col], errors='coerce').fillna(0)  
  
 *# Calculate any missing sums* df['ELIGIBLE\_IGST\_SUM'] = df[  
 ['ELIGIBLE\_IGST\_AS\_IGST', 'ELIGIBLE\_CGST\_AS\_IGST', 'ELIGIBLE\_SGST\_AS\_IGST', 'ELIGIBLE\_AMOUNT']].sum(axis=1)  
 df['INELIGIBLE\_IGST\_SUM'] = df[  
 ['INELIGIBLE\_IGST\_AS\_IGST', 'INELIGIBLE\_CGST\_AS\_IGST', 'INELIGIBLE\_SGST\_AS\_IGST', 'INELIGIBLE\_AMOUNT']].sum(axis=1)  
  
 logging.info(f"Successfully loaded Excel file: {*file\_path*}")  
  
 elif file\_ext == '.csv':  
 *# Read CSV with flexible parsing* df = pd.read\_csv(  
 *file\_path*,  
 dtype=*str*,  
 encoding='utf-8',  
 na\_values=['', 'NA', 'N/A', 'NULL'],  
 keep\_default\_na=False  
 )  
 logging.info(f"Successfully loaded CSV file: {*file\_path*}")  
  
 else:  
 logging.error(f"Unsupported file format: {*file\_path*}")  
 return None  
  
 *# Clean column names and data* df = clean\_data(df)  
  
 *# Validate each row* for idx, row in df.iterrows():  
 try:  
 validate\_row(row)  
 except *ValueError* as e:  
 logging.error(f"Row {idx + 1} validation failed: {*str*(e)}")  
 *# Either remove invalid rows or raise exception  
 # df.drop(index=idx, inplace=True) # Option 1: Skip invalid rows* raise *ValueError*(f"Row {idx + 1} invalid: {*str*(e)}") *# Option 2: Fail fast* logging.info(f"Columns in data: {df.columns.tolist()}")  
 logging.info(f"First row sample:\n{df.iloc[0].to\_dict()}")  
  
 return df  
  
 except *PermissionError*:  
 logging.error(f"Permission denied when reading: {*file\_path*}")  
 return None  
 except *Exception* as e:  
 logging.error(f"Error reading {*file\_path*}: {*str*(e)}")  
 return None

PDF\_Generator :-  
  
from PyPDF2 import PdfReader, PdfWriter  
import os  
import logging  
  
*# Configure logging*logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")  
  
def merge\_pdfs(*input\_folder*, *output\_pdf*, *sort\_key*=None):  
 *"""  
 Merges all PDFs in the given folder into a single PDF.* ***:param*** *input\_folder: Folder containing individual PDFs* ***:param*** *output\_pdf: Path to save the merged PDF* ***:param*** *sort\_key: Optional function to sort PDF filenames (e.g., lambda x: int(x.split('\_')[1]))  
 """* pdf\_writer = PdfWriter()  
 pdf\_files = [f for f in os.listdir(*input\_folder*) if f.endswith(".pdf")]  
  
 if not pdf\_files:  
 logging.error("❌ No PDFs found in the folder.")  
 return  
  
 *# Sort PDF files* if *sort\_key*:  
 pdf\_files.sort(key=*sort\_key*)  
 else:  
 pdf\_files.sort()  
  
 logging.info(f"Found {*len*(pdf\_files)} PDFs to merge.")  
  
 for i, pdf\_file in *enumerate*(pdf\_files, start=1):  
 pdf\_path = os.path.join(*input\_folder*, pdf\_file)  
 try:  
 pdf\_reader = PdfReader(pdf\_path)  
 for page in pdf\_reader.pages:  
 pdf\_writer.add\_page(page)  
 logging.info(f"✅ Added {pdf\_file} ({i} of {*len*(pdf\_files)})")  
 except *Exception* as e:  
 logging.error(f"❌ Error reading {pdf\_file}: {e}")  
  
 try:  
 with *open*(*output\_pdf*, "wb") as output:  
 pdf\_writer.write(output)  
 logging.info(f"✅ Merged PDF saved: {*output\_pdf*}")  
 except *Exception* as e:  
 logging.error(f"❌ Error saving merged PDF: {e}")  
  
*# Example usage*if \_\_name\_\_ == "\_\_main\_\_":  
 input\_folder = "C:/Users/anich/Downloads/Output" *# Folder containing individual PDFs* output\_pdf = "C:/Users/anich/Downloads/Merged.pdf" *# Path to save the merged PDF* merge\_pdfs(input\_folder, output\_pdf)