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("1200x800")  
  
 *# Main container* main\_frame = tb.Frame(*self*.root)  
 main\_frame.pack(fill=tk.BOTH, expand=True, padx=20, pady=20)  
  
 *# Left panel - Controls* control\_frame = tb.Frame(main\_frame)  
 control\_frame.pack(side=tk.LEFT, fill=tk.Y, padx=10, pady=10)  
  
 *# Control buttons* btn\_data = tb.Button(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(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(control\_frame, text="🚀 Generate ISD Invoices", bootstyle="success",  
 command=*self*.start\_processing)  
 btn\_start.pack(fill=tk.X, padx=10, pady=20)  
  
 *# Template status labels  
 self*.lbl\_eligible\_template = tb.Label(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(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(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(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  
 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\_"  
 tax\_fields = ['CGST', 'SGST', 'UTGST', 'IGST']  
  
 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"""* if not *all*([*self*.input\_file, *self*.output\_folder]):  
 messagebox.showerror("Error", "Please select data file and output folder!")  
 return  
  
 try:  
 data = read\_excel\_csv(*self*.input\_file)  
 if data is None:  
 messagebox.showerror("Error", "Failed to read data file.")  
 return  
  
 *# Create output folders* pdf\_output\_folder = os.path.join(*self*.output\_folder, "PDF\_Output")  
 os.makedirs(pdf\_output\_folder, exist\_ok=True)  
  
 temp\_docx\_folder = os.path.join(*self*.output\_folder, "TEMP\_DOCX")  
 os.makedirs(temp\_docx\_folder, exist\_ok=True)  
  
 success\_count = 0  
  
 for idx, row in data.iterrows():  
 try:  
 logging.info(f"\nProcessing row {idx}:")  
 logging.info(  
 f"Eligible amounts - CGST: {row['ELIGIBLE\_CGST']}, SGST: {row['ELIGIBLE\_SGST']}, IGST: {row['ELIGIBLE\_IGST']}")  
 logging.info(  
 f"Ineligible amounts - CGST: {row['INELIGIBLE\_CGST']}, SGST: {row['INELIGIBLE\_SGST']}, IGST: {row['INELIGIBLE\_IGST']}")  
  
 *# Process both eligible and ineligible documents for each row* for is\_eligible in [True, False]:  
 *# Get the appropriate template path* template\_path = *self*.eligible\_template if is\_eligible else *self*.ineligible\_template  
 prefix = "Eligible" if is\_eligible else "Ineligible"  
  
 *# Skip if no amounts for this type* if not *self*.has\_tax\_amounts(row, is\_eligible):  
 logging.info(f"No {'eligible' if is\_eligible else 'ineligible'} amounts found")  
 continue  
  
 *# 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* pdf\_filename = f"{prefix}\_ISD\_{invoice\_num}\_{timestamp}.pdf"  
 pdf\_path = os.path.join(pdf\_output\_folder, pdf\_filename)  
 convert(docx\_path, pdf\_path)  
  
 *# Delete temporary DOCX immediately after PDF conversion* 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  
  
 *# Remove temporary DOCX folder if empty* try:  
 os.rmdir(temp\_docx\_folder)  
 except *OSError*:  
 pass *# Folder not empty* messagebox.showinfo("Success",  
 f"Processing complete!\n\nGenerated {success\_count} PDF invoices.\n"  
 f"Location: {pdf\_output\_folder}")  
  
 except *Exception* as e:  
 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\_CGST', 'ELIGIBLE\_SGST', 'ELIGIBLE\_UTGST', 'ELIGIBLE\_IGST',  
 'ELIGIBLECGST', 'ELIGIBLESGST', 'ELIGIBLEUTGST', 'ELIGIBLEIGST'  
 ]  
  
 *# 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"""* tree\_frame = tb.Frame(*parent\_frame*)  
 tree\_frame.pack(fill=tk.BOTH, expand=True, padx=20, pady=10)  
  
 *# Scrollbars* tree\_scroll\_y = tk.Scrollbar(tree\_frame, orient="vertical")  
 tree\_scroll\_y.pack(side=tk.RIGHT, fill=tk.Y)  
 tree\_scroll\_x = tk.Scrollbar(tree\_frame, orient="horizontal")  
 tree\_scroll\_x.pack(side=tk.BOTTOM, fill=tk.X)  
  
 *# Treeview* tree = ttk.Treeview(tree\_frame, style="Custom.Treeview",  
 yscrollcommand=tree\_scroll\_y.set,  
 xscrollcommand=tree\_scroll\_x.set)  
 tree.pack(pady=10, fill=tk.BOTH, expand=True)  
  
 *# Configure scrollbars* tree\_scroll\_y.config(command=tree.yview)  
 tree\_scroll\_x.config(command=tree.xview)  
  
 return tree  
  
 def display\_data(*self*, *data*):  
 *"""Display data in the Treeview"""  
 self*.tree.delete(\**self*.tree.get\_children())  
 *self*.tree["columns"] = *list*(*data*.columns)  
 *self*.tree["show"] = "headings"  
  
 for col in *data*.columns:  
 *self*.tree.heading(col, text=col)  
 *self*.tree.column(col, width=150, anchor="center")  
  
 for \_, row in *data*.iterrows():  
 *self*.tree.insert("", "end", values=*list*(row))  
  
 *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 cgst': 'ELIGIBLE\_CGST',  
 'eligible sgst': 'ELIGIBLE\_SGST',  
 'eligible utgst': 'ELIGIBLE\_UTGST',  
 'eligible igst': 'ELIGIBLE\_IGST',  
 'ineligible cgst': 'INELIGIBLE\_CGST',  
 'ineligible sgst': 'INELIGIBLE\_SGST',  
 'ineligible utgst': 'INELIGIBLE\_UTGST',  
 'ineligible igst': 'INELIGIBLE\_IGST',  
 '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*, *is\_eligible*):  
 row\_data = {}  
 prefix = "ELIGIBLE\_" if *is\_eligible* else "INELIGIBLE\_"  
  
 *# Process tax fields* tax\_types = ['CGST', 'SGST', 'UTGST', 'IGST']  
 for tax in tax\_types:  
 col\_name = prefix + tax  
 value = safe\_float\_conversion(*row*[col\_name]) if col\_name in *row* else 0  
 row\_data[tax] = format\_value(value, tax)  
  
 *# Calculate total amount* total\_amount = *sum*(*float*(row\_data.get(tax, 0)) for tax in tax\_types)  
 row\_data['Amount'] = format\_value(total\_amount, 'Amount')  
  
 *# Process amount in words* if *any*('amount\_in\_words' in ph.lower() for ph in *template\_placeholders*):  
 try:  
 words = num2words(total\_amount, lang='en\_IN').title()  
 words = words.replace('And', 'and')  
 row\_data['amount\_in\_words'] = f"{words} Rupees Only"  
 except *Exception* as e:  
 logging.error(f"Amount to words conversion failed: {*str*(e)}")  
 row\_data['amount\_in\_words'] = ""  
  
 *# Map other fields* field\_mapping = {  
 *# Invoice fields* 'Invoice Number': 'INVOICE\_NUMBER',  
 'Invoice Date': 'INVOICE\_DATE',  
  
 *# ISD Distributor fields* 'ISD Distributor GSTIN': 'ISD\_DISTRIBUTOR\_GSTIN',  
 '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',  
  
 *# Credit Recipient fields* 'Credit Recipient GSTIN': 'CREDIT\_RECIPIENT\_GSTIN',  
 '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',  
  
 *# Tax fields - Handle both eligible and ineligible* 'Eligible cgst': 'ELIGIBLE\_CGST',  
 'Eligible sgst': 'ELIGIBLE\_SGST',  
 'Eligible utgst': 'ELIGIBLE\_UTGST',  
 'Eligible igst': 'ELIGIBLE\_IGST',  
 'Ineligible cgst': 'INELIGIBLE\_CGST',  
 'Ineligible sgst': 'INELIGIBLE\_SGST',  
 'Ineligible utgst': 'INELIGIBLE\_UTGST',  
 'Ineligible igst': 'INELIGIBLE\_IGST',  
 'cgst': 'CGST', *# Fallback* 'sgst': 'SGST', *# Fallback* 'utgst': 'UTGST', *# Fallback* 'igst': 'IGST', *# Fallback  
  
 # Amount fields* 'Amount': 'AMOUNT',  
 'Total': 'AMOUNT',  
  
 *# Contact fields* 'Reg. Office': 'REG\_OFFICE',  
 'CIN': 'CIN',  
 'E-Mail': 'E\_MAIL',  
 'Website': 'WEBSITE',  
  
 *# Special fields* 'amount\_in\_words': 'AMOUNT\_IN\_WORDS'  
 }  
  
 for template\_ph, data\_col in field\_mapping.items():  
 if template\_ph in *template\_placeholders*:  
 row\_data[template\_ph] = format\_value(*row*.get(data\_col, ''), template\_ph)  
  
 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:  
 *# Process all paragraphs in main document* 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*: Dict[*str*, *str*]):  
 *"""Enhanced placeholder replacement with bold formatting preservation"""  
 # First combine all runs to handle split placeholders* full\_text = ''.join(run.text for run in *paragraph*.runs)  
  
 *# Skip if no placeholders or bold markers* if not (*any*(f'{{{{{ph}}}}}' in full\_text for ph in *row\_data*) or '\*\*' in full\_text):  
 return  
  
 *# Perform all placeholder replacements first* modified\_text = full\_text  
 for ph, value in *row\_data*.items():  
 modified\_text = modified\_text.replace(f'{{{{{ph}}}}}', *str*(value))  
  
 *# Only update if changes were made* if modified\_text != full\_text:  
 *# Clear existing content  
 paragraph*.clear()  
  
 *# Split text by bold markers and process each segment* parts = modified\_text.split('\*\*')  
 for i, part in *enumerate*(parts):  
 run = *paragraph*.add\_run(part)  
 run.font.size = Pt(10)  
  
 *# Apply bold to every odd segment (text between \*\* markers)* if i % 2 == 1: *# This is text between \*\* markers* run.bold = True  
  
 *# Preserve original font if available* if *paragraph*.runs and *paragraph*.runs[0].font.name:  
 run.font.name = *paragraph*.runs[0].font.name  
  
  
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()  
  
 *# 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*]:  
 *"""Extract all unique placeholders from a DOCX template"""* doc = Document(*template\_path*)  
 placeholders = *set*()  
 placeholder\_pattern = re.compile(r"\{\{\s\*(.\*?)\s\*\}\}") *# Handles whitespace* def extract\_from\_text(*text*: *str*):  
 return {match.strip() for match in placeholder\_pattern.findall(*text*)}  
  
 *# 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:  
 placeholders.update(extract\_from\_text(paragraph.text))  
 for run in paragraph.runs:  
 placeholders.update(extract\_from\_text(run.text))  
  
 return {ph for ph in placeholders if ph} *# Remove empty strings*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("=====================")

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  
from data\_mapper import prepare\_row\_data, map\_data\_to\_docx  
  
  
logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")  
  
  
def fill\_docx\_template(*template\_path*: *str*, *output\_path*: *str*, *replacements*: Dict) -> *bool*:  
 *"""  
 Fill a DOCX template with provided replacements while preserving formatting  
 Args:  
 template\_path: Path to the template DOCX file  
 output\_path: Path to save the filled document  
 replacements: Dictionary of placeholder-value pairs  
 Returns:  
 bool: True if successful, False otherwise  
 """* try:  
 doc = Document(*template\_path*)  
  
 if not replace\_all\_placeholders(doc, *replacements*):  
 return False  
  
 doc.save(*output\_path*)  
 return True  
  
 except *Exception* as e:  
 logging.error(f"Error filling template: {*str*(e)}")  
 return False  
  
  
def replace\_all\_placeholders(*doc*: Document, *row\_data*: Dict[*str*, *str*]) -> *bool*:  
 *"""  
 Replace placeholders throughout all document components  
 Args:  
 doc: The Document object to process  
 row\_data: Dictionary of placeholder replacements  
 Returns:  
 bool: True if successful, False if errors occurred  
 """* try:  
 *# Process main document paragraphs* for paragraph in *doc*.paragraphs:  
 replace\_in\_paragraph(paragraph, *row\_data*)  
  
 *# Process 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*: Dict[*str*, *str*]):  
 *"""  
 Replace placeholders in a paragraph while preserving formatting  
 Handles bold formatting (\*\*text\*\*) in the template  
 Args:  
 paragraph: The paragraph to process  
 row\_data: Dictionary of placeholder replacements  
 """  
 # Combine runs to handle split placeholders* full\_text = ''.join(run.text for run in *paragraph*.runs)  
  
 *# Skip if no placeholders or bold markers* if not (*any*(f'{{{{{ph}}}}}' in full\_text for ph in *row\_data*) or '\*\*' in full\_text):  
 return  
  
 *# Perform all placeholder replacements* modified\_text = full\_text  
 for ph, value in *row\_data*.items():  
 modified\_text = modified\_text.replace(f'{{{{{ph}}}}}', *str*(value))  
  
 *# Only update if changes were made* if modified\_text != full\_text:  
 *paragraph*.clear()  
  
 *# Split text by bold markers and process each segment* parts = modified\_text.split('\*\*')  
 for i, part in *enumerate*(parts):  
 run = *paragraph*.add\_run(part)  
 run.font.size = Pt(10) *# Maintain font size  
  
 # Apply bold to text between \*\* markers* if i % 2 == 1:  
 run.bold = True  
  
 *# Preserve original font if available* if *paragraph*.runs and *paragraph*.runs[0].font.name:  
 run.font.name = *paragraph*.runs[0].font.name  
  
  
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 pandas as pd  
import logging  
from typing import Optional  
  
logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")  
  
  
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 second row to get tax type labels* header\_df = pd.read\_excel(*file\_path*, header=None, nrows=2)  
 tax\_labels = header\_df.iloc[1, 14:22].tolist() *# Columns O to V  
  
 # Read data with proper column names* df = pd.read\_excel(  
 *file\_path*,  
 engine='openpyxl',  
 header=0,  
 skiprows=[1], *# Skip the tax type labels row* names=[  
 'INVOICE\_NUMBER', 'INVOICE\_DATE', 'ISD\_DISTRIBUTOR\_GSTIN',  
 'ISD\_DISTRIBUTOR\_NAME', 'ISD\_DISTRIBUTOR\_ADDRESS',  
 'ISD\_DISTRIBUTOR\_STATE', 'ISD\_DISTRIBUTOR\_PINCODE',  
 'ISD\_DISTRIBUTOR\_STATE\_CODE', 'CREDIT\_RECIPIENT\_GSTIN',  
 'CREDIT\_RECIPIENT\_NAME', 'CREDIT\_RECIPIENT\_ADDRESS',  
 'CREDIT\_RECIPIENT\_STATE', 'CREDIT\_RECIPIENT\_PINCODE',  
 'CREDIT\_RECIPIENT\_STATE\_CODE',  
 'ELIGIBLE\_CGST', 'ELIGIBLE\_SGST', 'ELIGIBLE\_UTGST', 'ELIGIBLE\_IGST',  
 'INELIGIBLE\_CGST', 'INELIGIBLE\_SGST', 'INELIGIBLE\_UTGST', 'INELIGIBLE\_IGST',  
 'AMOUNT', 'REG\_OFFICE', 'CIN', 'E\_MAIL', 'WEBSITE'  
 ],  
 dtype=*str*,  
 na\_values=['', 'NA', 'N/A', 'NULL'],  
 keep\_default\_na=False  
 )  
 logging.info(f"Successfully loaded Excel file: {*file\_path*}")  
  
 elif file\_ext == '.csv':  
 *# Existing CSV handling  
 # 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)  
 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  
  
  
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*

pdf\_generator.py :-

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)