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  
from file\_reader import read\_excel\_csv  
from data\_mapper import map\_data\_to\_docx  
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*.setup\_ui()  
 *self*.setup\_menu()  
  
 *# Initialize variables  
 self*.input\_file = None  
 *self*.template\_file = None  
 *self*.output\_folder = None  
 *self*.current\_data = None  
  
 def setup\_ui(*self*):  
 *"""Setup the main user interface"""  
 self*.root.title("Automated Document Filler")  
 *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\_template = tb.Button(control\_frame, text="📄 Upload DOCX Template", command=*self*.upload\_template)  
 btn\_template.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="🚀 Start Processing", bootstyle="success",  
 command=*self*.start\_processing)  
 btn\_start.pack(fill=tk.X, padx=10, pady=20)  
  
 btn\_docx\_to\_pdf = tb.Button(control\_frame, text="📄 to 📄 Convert DOCX to PDF", bootstyle="info",  
 command=*self*.select\_folders\_and\_convert)  
 btn\_docx\_to\_pdf.pack(fill=tk.X, padx=10, pady=20)  
  
 *# 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\_template = tb.Label(control\_frame, text="No Template File Loaded", bootstyle="secondary")  
 *self*.lbl\_template.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 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 upload\_template(*self*):  
 *"""Handle template file upload"""* file\_path = filedialog.askopenfilename(filetypes=[("Word Documents", "\*.docx")])  
 if file\_path:  
 *self*.template\_file = file\_path  
 *self*.lbl\_template.config(text=f"📄 {os.path.basename(file\_path)} Loaded")  
 logging.info(f"Template file loaded: {file\_path}")  
 messagebox.showinfo("Success", "Template loaded successfully!")  
  
 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}")  
  
 def start\_processing(*self*):  
 *"""Start the document generation process"""* if not *all*([*self*.input\_file, *self*.template\_file, *self*.output\_folder]):  
 messagebox.showerror("Error", "Please upload all required files!")  
 return  
  
 try:  
 data = read\_excel\_csv(*self*.input\_file)  
 if data is None:  
 messagebox.showerror("Error", "Failed to read data file.")  
 return  
  
 generated\_files = map\_data\_to\_docx(  
 template\_path=*self*.template\_file,  
 data=data,  
 output\_folder=*self*.output\_folder  
 )  
  
 if not generated\_files:  
 messagebox.showerror("Error", "No documents were generated.")  
 return  
  
 *# Convert to PDF* pdf\_output\_folder = os.path.join(*self*.output\_folder, "PDF\_Output")  
 os.makedirs(pdf\_output\_folder, exist\_ok=True)  
  
 for docx\_file in generated\_files:  
 pdf\_file = os.path.join(pdf\_output\_folder,  
 os.path.splitext(os.path.basename(docx\_file))[0] + ".pdf")  
 try:  
 convert(docx\_file, pdf\_file)  
 logging.info(f"Converted {docx\_file} to {pdf\_file}")  
 except *Exception* as e:  
 logging.error(f"Error converting {docx\_file} to PDF: {e}")  
 messagebox.showerror("Error", f"Error converting {docx\_file} to PDF: {e}")  
  
 messagebox.showinfo("Success", "Documents generated and converted to PDF successfully!")  
  
 except *Exception* as e:  
 messagebox.showerror("Error", f"Processing failed: {*str*(e)}")  
 logging.error(f"Processing error: {*str*(e)}")  
  
 def convert\_docx\_to\_pdf(*self*, *docx\_folder*, *output\_folder*):  
 *"""Convert DOCX files to PDF"""* if not os.path.exists(*docx\_folder*):  
 logging.error(f"DOCX folder not found: {*docx\_folder*}")  
 messagebox.showerror("Error", f"DOCX folder not found: {*docx\_folder*}")  
 return  
  
 os.makedirs(*output\_folder*, exist\_ok=True)  
 docx\_files = [f for f in os.listdir(*docx\_folder*) if f.endswith(".docx")]  
  
 if not docx\_files:  
 logging.warning("No DOCX files found in the folder.")  
 messagebox.showwarning("Warning", "No DOCX files found in the folder.")  
 return  
  
 for docx\_file in docx\_files:  
 docx\_path = os.path.join(*docx\_folder*, docx\_file)  
 pdf\_file = os.path.splitext(docx\_file)[0] + ".pdf"  
 pdf\_path = os.path.join(*output\_folder*, pdf\_file)  
  
 try:  
 convert(docx\_path, pdf\_path)  
 logging.info(f"Converted: {docx\_file} to {pdf\_file}")  
 except *Exception* as e:  
 logging.error(f"Error converting {docx\_file}: {e}")  
 messagebox.showerror("Error", f"Error converting {docx\_file}: {e}")  
  
 def select\_folders\_and\_convert(*self*):  
 *"""Handle folder selection for DOCX to PDF conversion"""* docx\_folder = filedialog.askdirectory(title="Select DOCX Folder")  
 if not docx\_folder:  
 return  
  
 output\_folder = filedialog.askdirectory(title="Select Output PDF Folder")  
 if not output\_folder:  
 return  
  
 *self*.convert\_docx\_to\_pdf(docx\_folder, output\_folder)  
 messagebox.showinfo("Conversion Complete", "DOCX to PDF conversion completed.")  
  
  
*# 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  
  
logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")  
  
*# Complete and corrected column mapping*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 (corrected spelling)* '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* 'igst': 'IGST',  
 'cgst': 'CGST',  
 'sgst': 'SGST',  
 'utgst': 'UTGST',  
 'amount': 'AMOUNT',  
  
 *# Contact fields* 'regoffice': 'REG\_OFFICE',  
 'cin': 'CIN',  
 'email': 'E\_MAIL',  
 'e-mail': 'E\_MAIL', *# Alternate mapping* 'website': 'WEBSITE',  
  
 *# Special fields* 'amount\_in\_words': 'AMOUNT' *# Special handling*}  
  
  
def normalize\_column\_names(*df*: pd.DataFrame) -> pd.DataFrame:  
 *"""Normalize column names to ensure consistent matching"""  
 df*.columns = [  
 col.strip().upper().replace(' ', '\_').replace('-', '\_')  
 for col in *df*.columns  
 ]  
 return *df*def map\_data\_to\_docx(*template\_path*: *str*, *data*: pd.DataFrame, *output\_folder*: *str*) -> Optional[List[*str*]]:  
 *"""Main function to generate DOCX files from template and data"""* 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"Template placeholders: {template\_placeholders}")  
 logging.info(f"Data columns: {*data*.columns.tolist()}")  
  
 for idx, row in *data*.iterrows():  
 try:  
 doc = Document(*template\_path*)  
 row\_data = prepare\_row\_data(row, template\_placeholders)  
  
 *# Debug output for first row* if idx == 0:  
 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)  
 doc.save(output\_path)  
 generated\_files.append(output\_path)  
 logging.info(f"Generated: {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: {*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*: pd.Series, *template\_placeholders*: Set[*str*]) -> Dict[*str*, *str*]:  
 *"""Prepare complete row data with all required fields and proper formatting"""* row\_data = {}  
  
 *# Process all placeholders in template* for ph in *template\_placeholders*:  
 *# Normalize the placeholder name* norm\_ph = ph.lower().replace(' ', '').replace('.', '').replace('-', '')  
  
 *# Special handling for amount\_in\_words* if norm\_ph == 'amount\_in\_words':  
 try:  
 amount = *float*(*row*['AMOUNT'])  
 words = num2words(amount, lang='en\_IN').title()  
 *# Ensure proper formatting* words = words.replace('And', 'and') *# Fix capitalization* row\_data['amount\_in\_words'] = f"{words} Rupees Only"  
 except *Exception* as e:  
 logging.error(f"Amount to words failed: {*str*(e)}")  
 row\_data['amount\_in\_words'] = ""  
 continue  
  
 *# Find matching column using our mapping* data\_key = COLUMN\_MAPPING.get(norm\_ph)  
  
 if data\_key and data\_key in *row*:  
 value = *row*[data\_key]  
 *# Convert numpy types to native Python* if *hasattr*(value, 'item'):  
 value = value.item()  
 row\_data[ph] = format\_value(value, ph)  
 else:  
 row\_data[ph] = ""  
 logging.warning(f"No data mapping for placeholder: {ph} (normalized: {norm\_ph})")  
  
 return row\_data  
  
  
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  
  
  
from docx.shared import Pt  
from typing import Dict  
  
  
def replace\_in\_paragraph(*paragraph*, *row\_data*: Dict[*str*, *str*]):  
 *"""Replace placeholders in a paragraph while preserving formatting.  
 Now supports \*\*bold\*\* markers in the template text."""  
  
 # 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}}}}}', 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*:  
 *"""Format values with special handling for certain fields"""* if pd.isna(*value*):  
 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*) -> *str*:  
 *"""Generate output path with invoice number if available"""* invoice\_num = *str*(*row\_data*.get('Invoice Number', *idx* + 1)).strip()  
 timestamp = datetime.now().strftime("%Y%m%d\_%H%M%S")  
 return os.path.join(*output\_folder*, f"ISD\_Invoice\_{invoice\_num}\_{timestamp}.docx")  
  
  
def log\_debug\_info(*row*, *template\_placeholders*, *row\_data*):  
 *"""Log debug information for the first row"""* logging.info("\n=== DEBUG INFORMATION ===")  
 logging.info(f"Template placeholders: {*template\_placeholders*}")  
 logging.info(f"Data columns: {*row*.index.tolist()}")  
 logging.info(f"First row data: {*dict*(*row*)}")  
  
 logging.info("\n=== PLACEHOLDER MAPPING ===")  
 for ph in *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 *row\_data*.items():  
 logging.info(f"{ph:25}: {value}")  
 logging.info("=====================")

docx\_filler.py :-

import os  
import re  
import logging  
from docx import Document  
import pandas as pd  
from docx.shared import Pt  
from typing import Dict, List, Optional  
from num2words import num2words *# For amount-to-words conversion*logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")  
  
  
def fill\_docx\_template(*template\_path*, *output\_path*, *replacements*):  
 try:  
 doc = Document(*template\_path*)  
  
 *# Replace in paragraphs* for paragraph in doc.paragraphs:  
 for key, value in *replacements*.items():  
 if f"{{{{{key}}}}}" in paragraph.text:  
 paragraph.text = paragraph.text.replace(f"{{{{{key}}}}}", *str*(value))  
  
 *# Replace in tables* for table in doc.tables:  
 for row in table.rows:  
 for cell in row.cells:  
 for key, value in *replacements*.items():  
 if f"{{{{{key}}}}}" in cell.text:  
 cell.text = cell.text.replace(f"{{{{{key}}}}}", *str*(value))  
  
 doc.save(*output\_path*)  
 return True  
 except *Exception* as e:  
 logging.error(f"Error filling template: {*str*(e)}")  
 return False  
  
 os.makedirs(output\_folder, exist\_ok=True)  
 generated\_files = []  
  
 for idx, row in data.iterrows():  
 try:  
 doc = Document(*template\_path*)  
 row\_data = process\_row(row, amount\_columns, convert\_amount\_to\_words)  
  
 if not replace\_placeholders\_in\_document(doc, row\_data):  
 continue  
  
 output\_path = save\_document(doc, output\_folder, idx)  
 if *output\_path*:  
 generated\_files.append(*output\_path*)  
  
 except *Exception* as e:  
 logging.error(f"Error processing row {idx + 1}: {*str*(e)}")  
 continue  
  
 return generated\_files if generated\_files else None  
  
 except *Exception* as e:  
 logging.error(f"Fatal error: {*str*(e)}")  
 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  
  
 return True  
  
def convert\_amount\_to\_words(*amount*: *float*) -> *str*:  
 *"""  
 Converts numeric amount to words representation.  
 Example: 1234.56 → "One Thousand Two Hundred Thirty-Four Point Five Six"  
  
 Args:  
 amount: Numeric value to convert  
  
 Returns:  
 String representation in words  
 """* try:  
 if pd.isna(*amount*):  
 return ""  
  
 *# Split into dollars and cents* dollars = *int*(*amount*)  
 cents = *round*((*amount* - dollars) \* 100)  
  
 dollar\_words = num2words(dollars, lang='en').title()  
  
 if cents > 0:  
 cent\_words = num2words(cents, lang='en').title()  
 return f"{dollar\_words} And {cent\_words} Cents"  
 return f"{dollar\_words} Only"  
  
 except *Exception* as e:  
 logging.warning(f"Amount-to-words conversion failed: {*str*(e)}")  
 return ""  
  
  
def process\_row(  
 *row*: pd.Series,  
 *amount\_columns*: Optional[List[*str*]],  
 *convert\_to\_words*: *bool* = False  
) -> Dict:  
 *"""Process row data with enhanced amount handling"""* row\_data = *row*.to\_dict()  
  
 *# Handle Amount field* if 'Amount' in row\_data:  
 if pd.isna(row\_data['Amount']) or is\_formula(row\_data['Amount']):  
 if *amount\_columns*:  
 row\_data['Amount'] = sum\_numeric\_columns(*row*, *amount\_columns*)  
 else:  
 row\_data['Amount'] = sum\_all\_numeric\_columns(*row*)  
  
 *# Add amount in words if requested* if *convert\_to\_words*:  
 row\_data['Amount\_In\_Words'] = convert\_amount\_to\_words(row\_data['Amount'])  
  
 *# Format all values* return {k: format\_value(v) for k, v in row\_data.items()}  
  
  
def is\_formula(*value*) -> *bool*:  
 *"""Check if a value might be an Excel formula"""* return *isinstance*(*value*, *str*) and *value*.startswith('=')  
  
  
def sum\_numeric\_columns(*row*: pd.Series, *columns*: List[*str*]) -> *float*:  
 *"""Sum specified numeric columns"""* try:  
 return *sum*(*float*(*row*[col]) for col in *columns* if pd.notna(*row*.get(col)))  
 except (*ValueError*, *TypeError*):  
 logging.warning(f"Couldn't sum columns {*columns*}")  
 return 0.0  
  
  
def sum\_all\_numeric\_columns(*row*: pd.Series) -> *float*:  
 *"""Sum all numeric columns in the row"""* try:  
 return *sum*(v for v in *row* if *isinstance*(v, (*int*, *float*)))  
 except *TypeError*:  
 return 0.0  
  
  
def format\_value(*value*) -> *str*:  
 *"""Format values for document insertion"""* if pd.isna(*value*):  
 return ""  
  
 if *isinstance*(*value*, (*int*, *float*)):  
 return "{:,.2f}".format(*value*)  
  
 return *str*(*value*)  
  
  
def replace\_placeholders\_in\_document(*doc*: Document, *row\_data*: Dict) -> *bool*:  
 *"""Replace placeholders throughout document components"""* try:  
 *# Normalize row\_data keys to lowercase for case-insensitive matching* normalized\_data = {*str*(k).lower(): *str*(v) for k, v in *row\_data*.items()}  
  
 *# Process all document parts* for paragraph in *doc*.paragraphs:  
 replace\_in\_paragraph(paragraph, normalized\_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, normalized\_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, normalized\_data)  
  
 for footer in [section.footer, section.first\_page\_footer]:  
 if footer:  
 for paragraph in footer.paragraphs:  
 replace\_in\_paragraph(paragraph, normalized\_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):  
 *"""Replace placeholders in a paragraph with actual values"""* if not *paragraph*.text.strip():  
 return  
  
 *# Combine runs to handle split placeholders* full\_text = ''.join(run.text for run in *paragraph*.runs)  
 original\_text = full\_text  
  
 *# Find all placeholders in the text* placeholders\_in\_text = re.findall(r'\{\{\s\*(.\*?)\s\*\}\}', full\_text, re.IGNORECASE)  
  
 *# Replace each found placeholder* for ph in *set*(placeholders\_in\_text): *# Use set to avoid duplicates* ph\_lower = ph.lower()  
 if ph\_lower in *row\_data*:  
 full\_text = full\_text.replace(f'{{{{{ph}}}}}', *row\_data*[ph\_lower])  
  
 *# Only modify if changes were made* if full\_text != original\_text:  
 *paragraph*.clear()  
 if full\_text.strip(): *# Only add run if there's content  
 paragraph*.add\_run(full\_text)  
  
def save\_document(*doc*: Document, *output\_folder*: *str*, *idx*: *int*) -> Optional[*str*]:  
 *"""Save filled document with sequential naming"""* try:  
 filename = f"document\_{*idx* + 1}.docx"  
 output\_path = os.path.join(*output\_folder*, filename)  
 *doc*.save(output\_path)  
 logging.info(f"Generated: {output\_path}")  
 return output\_path  
 except *Exception* as e:  
 logging.error(f"Error saving document {*idx* + 1}: {*str*(e)}")  
 return None  
  
  
def extract\_placeholders(*template\_path*: *str*) -> *set*:  
 *"""  
 Extracts all placeholders from a DOCX template, including headers/footers.  
  
 Args:  
 template\_path: Path to the DOCX template file  
  
 Returns:  
 Set of cleaned placeholder names (e.g., {'invoice\_number', 'amount'})  
  
 Raises:  
 FileNotFoundError: If template doesn't exist  
 ValueError: If template is invalid  
 """* try:  
 if not os.path.exists(*template\_path*):  
 raise *FileNotFoundError*(f"Template file not found: {*template\_path*}")  
  
 doc = Document(*template\_path*)  
 placeholders = *set*()  
 placeholder\_pattern = re.compile(r"\{\{\s\*(.\*?)\s\*\}\}") *# Handles whitespace* def extract\_from\_text(*text*: *str*):  
 *"""Inner function to extract placeholders from text"""* return {match.strip().lower() for match in placeholder\_pattern.findall(*text*)}  
  
 *# Process main document paragraphs (including formatted runs)* for paragraph in doc.paragraphs:  
 placeholders.update(extract\_from\_text(paragraph.text))  
 for run in paragraph.runs:  
 placeholders.update(extract\_from\_text(run.text))  
  
 *# Process tables* for table in doc.tables:  
 for row in table.rows:  
 for cell in row.cells:  
 for paragraph in cell.paragraphs:  
 placeholders.update(extract\_from\_text(paragraph.text))  
 for run in paragraph.runs:  
 placeholders.update(extract\_from\_text(run.text))  
  
 *# 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:  
 placeholders.update(extract\_from\_text(paragraph.text))  
  
 for footer in [section.footer, section.first\_page\_footer]:  
 if footer:  
 for paragraph in footer.paragraphs:  
 placeholders.update(extract\_from\_text(paragraph.text))  
  
 *# Clean results - remove empty matches and normalize* return {ph for ph in placeholders if ph} *# Remove empty strings* except *Exception* as e:  
 raise *ValueError*(f"Failed to extract placeholders: {*str*(e)}")

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]:  
 *"""  
 Reads Excel or CSV file into a pandas DataFrame with robust error handling.  
  
 Args:  
 file\_path: Path to the input file (Excel or CSV)  
  
 Returns:  
 pandas DataFrame if successful, None otherwise  
 """* 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 Excel with formula evaluation and proper dtype handling* df = pd.read\_excel(  
 *file\_path*,  
 engine='openpyxl',  
 dtype=*str*, *# Read all as string to preserve formatting* na\_values=['', 'NA', 'N/A', 'NULL'],  
 keep\_default\_na=False  
 )  
 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)  
 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)