1. Main.py :-

import tkinter as tk  
import pandas as pd  
import shutil  
import json  
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  
import math  
from num2words import num2words  
  
*# Configure logging*logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")  
  
  
class DocumentFillerApp:  
 def \_\_init\_\_(*self*, *root*):  
 *self*.root = *root  
  
 # Initialize variables first  
 self*.input\_file = None  
 *self*.output\_folder = None  
 *self*.current\_data = None  
 *self*.eligible\_template = None  
 *self*.ineligible\_template = None  
  
 *# Setup UI before loading templates  
 self*.setup\_ui()  
 *self*.setup\_menu()  
 *self*.setup\_template\_access()  
  
 *# Now load templates (which can update the UI elements)  
 self*.load\_default\_templates()  
  
 *# Check system requirements  
 self*.check\_system\_requirements()  
  
 *# Load settings  
 self*.load\_settings()  
  
 def load\_default\_templates(*self*):  
 *"""Load default templates from the templates folder with better error handling"""* try:  
 *# Get the directory where the executable or script is located* if *getattr*(sys, 'frozen', False):  
 *# Running as compiled executable* base\_path = sys.\_MEIPASS  
 templates\_dir = os.path.join(base\_path, "templates")  
  
 *# Fallback if templates aren't in MEIPASS/templates* if not os.path.exists(templates\_dir):  
 templates\_dir = os.path.join(base\_path)  
 else:  
 *# Running as script* templates\_dir = os.path.join(os.path.dirname(os.path.abspath(\_\_file\_\_)), "templates")  
  
 *# Try multiple possible template locations* possible\_paths = {  
 'eligible': [  
 os.path.join(templates\_dir, "eligible\_template.docx"),  
 os.path.join(templates\_dir, "templates", "eligible\_template.docx"),  
 os.path.join(os.getcwd(), "templates", "eligible\_template.docx")  
 ],  
 'ineligible': [  
 os.path.join(templates\_dir, "ineligible\_template.docx"),  
 os.path.join(templates\_dir, "templates", "ineligible\_template.docx"),  
 os.path.join(os.getcwd(), "templates", "ineligible\_template.docx")  
 ]  
 }  
  
 *# Find the first valid path for each template  
 self*.eligible\_template = *next*(  
 (path for path in possible\_paths['eligible'] if os.path.exists(path)),  
 None  
 )  
 *self*.ineligible\_template = *next*(  
 (path for path in possible\_paths['ineligible'] if os.path.exists(path)),  
 None  
 )  
  
 if not *self*.eligible\_template:  
 raise *FileNotFoundError*(f"Eligible template not found in any of: {possible\_paths['eligible']}")  
 if not *self*.ineligible\_template:  
 raise *FileNotFoundError*(f"Ineligible template not found in any of: {possible\_paths['ineligible']}")  
  
 logging.info("Default templates loaded successfully")  
  
 *# Update UI labels if they exist* if *hasattr*(*self*, 'lbl\_eligible\_template'):  
 *self*.lbl\_eligible\_template.config(  
 text=f"✅ Eligible: {os.path.basename(*self*.eligible\_template)}",  
 bootstyle="success"  
 )  
 *self*.lbl\_ineligible\_template.config(  
 text=f"✅ Ineligible: {os.path.basename(*self*.ineligible\_template)}",  
 bootstyle="success"  
 )  
  
 except *Exception* as e:  
 logging.error(f"Failed to load default templates: {*str*(e)}")  
 messagebox.showerror(  
 "Critical Error",  
 f"Failed to load required templates:\n{*str*(e)}\n\n"  
 "Please ensure the template files are in the correct location."  
 )  
 *self*.disable\_template\_functionality()  
  
 def disable\_template\_functionality(*self*):  
 *"""Disable functionality that requires templates"""* if *hasattr*(*self*, 'btn\_start'):  
 *self*.btn\_start.config(state='disabled')  
 if *hasattr*(*self*, 'lbl\_eligible\_template'):  
 *self*.lbl\_eligible\_template.config(  
 text="❌ Eligible Template: Not Found",  
 bootstyle="danger"  
 )  
 if *hasattr*(*self*, 'lbl\_ineligible\_template'):  
 *self*.lbl\_ineligible\_template.config(  
 text="❌ Ineligible Template: Not Found",  
 bootstyle="danger"  
 )  
  
 def setup\_ui(*self*):  
 *"""Setup the main user interface"""  
 self*.root.title("Automated ISD Document Generator")  
 *self*.root.geometry("1920x1080")  
 *self*.root.state("zoomed")  
  
 *# 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)  
  
 *self*.btn\_start = tb.Button(*self*.control\_frame, text="🚀 Generate ISD Invoices",  
 bootstyle="success",  
 command=*self*.start\_processing)  
 *self*.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="⏳ Loading eligible template...",  
 bootstyle="warning")  
 *self*.lbl\_eligible\_template.pack(fill=tk.X, padx=10, pady=5)  
  
 *self*.lbl\_ineligible\_template = tb.Label(*self*.control\_frame,  
 text="⏳ Loading ineligible template...",  
 bootstyle="warning")  
 *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\_"  
 *# Try both prefixed and unprefixed column names* tax\_fields = ['CGST\_AS\_IGST', 'SGST\_AS\_IGST', 'CGST\_AS\_CGST', 'SGST\_UTGST\_AS\_SGST\_UTGST']  
  
 for tax in tax\_fields:  
 *# Try prefixed column first* col\_name = prefix + tax  
 if col\_name in *row* and pd.notna(*row*[col\_name]) and *float*(*row*[col\_name]) > 0:  
 return True  
 *# Fallback to unprefixed column* elif tax in *row* and pd.notna(*row*[tax]) and *float*(*row*[tax]) > 0:  
 return True  
 return False  
  
 def is\_row\_eligible(*self*, *row*):  
 *"""Determine if row contains eligible or ineligible data"""  
 # Check both prefixed and unprefixed columns* eligible\_cols = [  
 'ELIGIBLE\_IGST\_AS\_IGST', 'ELIGIBLE\_CGST\_AS\_IGST',  
 'ELIGIBLE\_SGST\_AS\_IGST', 'ELIGIBLE\_CGST\_AS\_CGST',  
 'ELIGIBLE\_SGST\_UTGST\_AS\_SGST\_UTGST',  
 *# Fallback to unprefixed columns* 'IGST\_AS\_IGST', 'CGST\_AS\_IGST',  
 'SGST\_AS\_IGST', 'CGST\_AS\_CGST',  
 'SGST\_UTGST\_AS\_SGST\_UTGST'  
 ]  
  
 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\_logging(*self*):  
 *"""Configure comprehensive logging for the application"""  
  
 # Create logs directory if it doesn't exist* log\_dir = os.path.join(os.path.expanduser("~"), "ISD\_Document\_Generator\_Logs")  
 os.makedirs(log\_dir, exist\_ok=True)  
  
 *# Create a time-based log filename* timestamp = datetime.now().strftime('%Y%m%d\_%H%M%S')  
 log\_file = os.path.join(log\_dir, f"ISD\_Generator\_{timestamp}.log")  
  
 *# Create logger* logger = logging.getLogger()  
 logger.setLevel(logging.DEBUG) *# Set to lowest level  
  
 # Create formatters* console\_formatter = logging.Formatter("%(levelname)s: %(message)s")  
 file\_formatter = logging.Formatter(  
 "%(asctime)s - %(name)s - %(levelname)s - %(message)s [%(filename)s:%(lineno)d]"  
 )  
  
 *# Console handler (shows simple messages)* console\_handler = logging.StreamHandler()  
 console\_handler.setLevel(logging.INFO) *# Only show INFO and above in console* console\_handler.setFormatter(console\_formatter)  
  
 *# File handler (logs everything with more details)* file\_handler = RotatingFileHandler(  
 log\_file,  
 maxBytes=5 \* 1024 \* 1024, *# 5MB per file* backupCount=3, *# Keep 3 backup files* encoding='utf-8'  
 )  
 file\_handler.setLevel(logging.DEBUG) *# Log everything* file\_handler.setFormatter(file\_formatter)  
  
 *# Add handlers to the logger* logger.addHandler(console\_handler)  
 logger.addHandler(file\_handler)  
  
 *# Special handling for docx and other libraries* logging.getLogger('docx').setLevel(logging.WARNING)  
 logging.getLogger('PIL').setLevel(logging.WARNING)  
 logging.getLogger('comtypes').setLevel(logging.WARNING)  
  
 *# Initial log message* logging.info("=" \* 50)  
 logging.info("Application logging initialized")  
 logging.info(f"Log file: {log\_file}")  
 logging.info("=" \* 50)  
  
 log\_dir = os.path.join(os.path.expanduser("~"), "ISD\_Document\_Generator\_Logs")  
 os.makedirs(log\_dir, exist\_ok=True)  
  
 log\_file = os.path.join(log\_dir, f"ISD\_Generator\_{datetime.now().strftime('%Y%m%d\_%H%M%S')}.log")  
  
 logging.basicConfig(  
 level=logging.INFO,  
 format="%(asctime)s - %(levelname)s - %(message)s",  
 handlers=[  
 logging.FileHandler(log\_file),  
 logging.StreamHandler()  
 ]  
 )  
 logging.info("Application logging initialized")  
  
 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}")  
  
 def setup\_template\_access(*self*):  
 *"""Setup functionality for accessing the Excel template"""  
 # Add button to UI  
 self*.template\_button = tb.Button(  
 *self*.control\_frame,  
 text="📊 Get Excel Template",  
 command=*self*.provide\_excel\_template,  
 bootstyle="info"  
 )  
 *self*.template\_button.pack(fill=tk.X, padx=10, pady=5)  
  
 def provide\_excel\_template(*self*):  
 *"""Provide the Excel template to the user"""* try:  
 *# Get the template from package resources* source\_path = *self*.get\_template\_path()  
  
 *# Determine where to save it* save\_path = filedialog.asksaveasfilename(  
 defaultextension=".xlsx",  
 filetypes=[("Excel files", "\*.xlsx")],  
 title="Save Excel Template As",  
 initialfile="ISD\_Input\_Template.xlsx"  
 )  
  
 if save\_path: *# User didn't cancel* shutil.copy(source\_path, save\_path)  
 messagebox.showinfo(  
 "Template Saved",  
 f"Excel template saved to:\n{save\_path}\n\n"  
 "Please use this format for your data."  
 )  
 *# Optionally open the file* if messagebox.askyesno("Open Template", "Open the template now?"):  
 os.startfile(save\_path)  
  
 except *Exception* as e:  
 messagebox.showerror("Error", f"Could not provide template:\n{*str*(e)}")  
  
 def get\_template\_path(*self*, *template\_type*="excel"):  
 try:  
 *# Validate template type* template\_files = {  
 'excel': 'ISD\_Input\_Template.xlsx',  
 'eligible': 'eligible\_template.docx',  
 'ineligible': 'ineligible\_template.docx'  
 }  
  
 if *template\_type* not in template\_files:  
 raise *ValueError*(f"Invalid template type. Must be one of: {*list*(template\_files.keys())}")  
  
 filename = template\_files[*template\_type*]  
  
 *# For PyInstaller bundle* if *getattr*(sys, 'frozen', False):  
 base\_path = sys.\_MEIPASS  
 possible\_paths = [  
 os.path.join(base\_path, filename), *# Directly in MEIPASS* os.path.join(base\_path, "templates", filename)  
 ]  
 else:  
 *# Development paths* possible\_paths = [  
 os.path.join(os.path.dirname(\_\_file\_\_), "templates", filename),  
 os.path.join(os.getcwd(), "templates", filename),  
 os.path.join(os.path.expanduser("~"), "templates", filename)  
 ]  
  
 *# Check each possible path* for path in possible\_paths:  
 if os.path.exists(path):  
 return path  
  
 raise *FileNotFoundError*(  
 f"Could not locate {filename} in any of these locations:\n"  
 + "\n".join(possible\_paths)  
 )  
  
 except *Exception* as e:  
 logging.error(f"Error finding template: {*str*(e)}")  
 raise  
  
 def convert\_to\_pdf(*self*, *docx\_path*, *pdf\_path*):  
 *"""Convert DOCX to PDF with multiple fallback methods"""* try:  
 *# Method 1: Try comtypes first (requires MS Word)* try:  
 from comtypes import client  
 word = client.CreateObject("Word.Application")  
 doc = word.Documents.Open(*docx\_path*)  
 doc.SaveAs(*pdf\_path*, FileFormat=17) *# 17 is PDF format* doc.Close()  
 word.Quit()  
 return True  
 except *ImportError*:  
 pass  
  
 *# Method 2: Try docx2pdf* try:  
 from docx2pdf import convert  
 convert(*docx\_path*, *pdf\_path*)  
 return True  
 except *Exception* as e:  
 logging.error(f"docx2pdf conversion failed: {*str*(e)}")  
  
 *# Method 3: Try unoconv (requires LibreOffice)* try:  
 import subprocess  
 subprocess.call(['unoconv', '-f', 'pdf', '-o', *pdf\_path*, *docx\_path*])  
 return os.path.exists(*pdf\_path*)  
 except *Exception* as e:  
 logging.error(f"unoconv conversion failed: {*str*(e)}")  
  
 *# All methods failed* logging.error("All PDF conversion methods failed")  
 return False  
  
 except *Exception* as e:  
 logging.error(f"PDF conversion error: {*str*(e)}")  
 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:  
 *# Verify input file and output folder* if not os.path.exists(*self*.input\_file):  
 logging.error(f"Input file not found: {*self*.input\_file}")  
 messagebox.showerror("Error", "Input file not found!")  
 return  
  
 if not os.path.isdir(*self*.output\_folder):  
 logging.error(f"Output folder not found: {*self*.output\_folder}")  
 messagebox.showerror("Error", "Output folder not found!")  
 return  
  
 *# 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()  
  
 *# Read data with better error handling* try:  
 data = read\_excel\_csv(*self*.input\_file)  
 if data is None or data.empty:  
 messagebox.showerror("Error", "No valid data found in the file.")  
 *self*.progress\_frame.pack\_forget()  
 return  
  
 *# Clean data - remove any rows with NaN values in critical columns* data = data.dropna(subset=['INVOICE\_NUMBER', 'INVOICE\_DATE', 'ISD\_DISTRIBUTOR\_GSTIN'])  
  
 except *Exception* as e:  
 messagebox.showerror("Error", f"Failed to read data file:\n{*str*(e)}")  
 *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")  
  
 try:  
 os.makedirs(eligible\_folder, exist\_ok=True)  
 os.makedirs(ineligible\_folder, exist\_ok=True)  
 os.makedirs(temp\_docx\_folder, exist\_ok=True)  
 except *PermissionError* as pe:  
 messagebox.showerror("Permission Error",  
 f"Cannot create output folders:\n{*str*(pe)}\n"  
 "Please choose a different output location.")  
 return  
  
 total\_rows = *len*(data)  
 success\_count = 0  
  
 for idx, row in data.iterrows():  
 try:  
 *# Safely calculate progress, handling NaN values* progress = 0  
 if total\_rows > 0 and not pd.isna(idx):  
 progress = (idx + 1) / total\_rows \* 100  
 progress = *max*(0, *min*(100, progress)) *# Clamp between 0-100  
  
 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.get('ELIGIBLE\_CGST\_AS\_IGST', 0)}, "  
 f"SGST: {row.get('ELIGIBLE\_SGST\_AS\_IGST', 0)}, "  
 f"IGST: {row.get('ELIGIBLE\_IGST\_AS\_IGST', 0)}"  
 )  
 logging.info(  
 f"Ineligible amounts - CGST: {row.get('INELIGIBLE\_CGST\_AS\_IGST', 0)}, "  
 f"SGST: {row.get('INELIGIBLE\_SGST\_AS\_IGST', 0)}, "  
 f"IGST: {row.get('INELIGIBLE\_IGST\_AS\_IGST', 0)}"  
 )  
  
 *# 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* try:  
 doc = Document(template\_path)  
 except *Exception* as e:  
 logging.error(f"Failed to open template: {*str*(e)}")  
 continue  
  
 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)  
  
 try:  
 doc.save(docx\_path)  
 except *Exception* as e:  
 logging.error(f"Failed to save DOCX: {*str*(e)}")  
 continue  
  
 *# 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)  
  
 if *self*.convert\_to\_pdf(docx\_path, pdf\_path):  
 success\_count += 1  
 logging.info(f"Generated {pdf\_filename}")  
 else:  
 logging.error(f"Failed to convert {docx\_filename} to PDF")  
  
 *# Delete temporary DOCX* try:  
 os.remove(docx\_path)  
 except *Exception* as e:  
 logging.error(f"Failed to delete temp DOCX: {*str*(e)}")  
  
 except *Exception* as e:  
 logging.error(f"Error processing row {idx}: {*str*(e)}", exc\_info=True)  
 continue  
  
 *# Clean up temporary folder* try:  
 if os.path.exists(temp\_docx\_folder):  
 if not os.listdir(temp\_docx\_folder):  
 os.rmdir(temp\_docx\_folder)  
 else:  
 logging.warning(f"Temporary folder not empty: {temp\_docx\_folder}")  
 except *Exception* as e:  
 logging.error(f"Error cleaning temp folder: {*str*(e)}")  
  
 *# 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 check\_system\_requirements(*self*):  
 *"""Verify all required system components are available"""* requirements = {  
 'Microsoft Word': *self*.check\_word\_installed(),  
 'Python Dependencies': *self*.check\_python\_dependencies()  
 }  
  
 if not *all*(requirements.values()):  
 messagebox.showwarning(  
 "System Requirements Check",  
 "Some system requirements are not met:\n\n" +  
 "\n".join(f"{name}: {'✔' if status else '✖'}"  
 for name, status in requirements.items()) +  
 "\n\nPDF generation may not work properly."  
 )  
  
 def check\_word\_installed(*self*):  
 *"""Check if Microsoft Word is available for PDF conversion"""* try:  
 from win32com.client import Dispatch  
 word = Dispatch('Word.Application')  
 word.Quit()  
 return True  
 except:  
 return False  
  
 def check\_python\_dependencies(*self*):  
 *"""Check if all required Python packages are installed"""* required = ['pandas', 'python-docx', 'docx2pdf', 'comtypes']  
 missing = []  
  
 for package in required:  
 try:  
 *\_\_import\_\_*(package)  
 except *ImportError*:  
 missing.append(package)  
  
 return *len*(missing) == 0  
  
 def load\_settings(*self*):  
 *"""Load user settings from config file"""  
 self*.settings\_file = os.path.join(  
 os.path.expanduser("~"),  
 ".isd\_document\_generator\_settings.json"  
 )  
  
 defaults = {  
 'last\_input\_folder': '',  
 'last\_output\_folder': '',  
 'theme': 'darkly' if darkdetect.isDark() else 'journal',  
 'window\_size': [1920, 1080]  
 }  
  
 try:  
 with *open*(*self*.settings\_file, 'r') as f:  
 *self*.settings = {\*\*defaults, \*\*json.load(f)}  
 except:  
 *self*.settings = defaults  
  
 def save\_settings(*self*):  
 *"""Save user settings to config file"""* try:  
 with *open*(*self*.settings\_file, 'w') as f:  
 json.dump(*self*.settings, f, indent=2)  
 except *Exception* as e:  
 logging.error(f"Error saving settings: {*str*(e)}")  
  
  
*# 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()

1. Data\_Mapper.py :-

import os  
import re  
import logging  
from docx import Document  
import tkinter as tk  
from tkinter import filedialog, messagebox, ttk  
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 preprocess\_data(*self*, *data*):  
 *"""Add ELIGIBLE\_ or INELIGIBLE\_ prefixes to tax columns based on the type"""* for idx, row in *data*.iterrows():  
 is\_eligible = *str*(row['ELIGIBLE/INELIGIBLE']).strip().lower() == 'eligible'  
 prefix = 'ELIGIBLE\_' if is\_eligible else 'INELIGIBLE\_'  
  
 tax\_fields = ['IGST\_AS\_IGST', 'CGST\_AS\_IGST', 'SGST\_UTGST\_AS\_IGST',  
 'IGST\_SUM', 'CGST\_AS\_CGST', 'CGST\_SUM',  
 'SGST\_UTGST\_AS\_SGST\_UTGST', 'SGST\_UTGST\_SUM', 'AMOUNT']  
  
 for field in tax\_fields:  
 if field in row:  
 *data*.at[idx, prefix + field] = row[field]  
  
 return *data*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\_amounts(*row\_data*, *prefix*):  
 *"""Validate that sums match their components"""* try:  
 *# Calculate expected sums* calc\_igst = *sum*([  
 *float*(*row\_data*.get('IGST\_AS\_IGST', '0').replace(',', '')),  
 *float*(*row\_data*.get('CGST\_AS\_IGST', '0').replace(',', '')),  
 *float*(*row\_data*.get('SGST\_AS\_IGST', '0').replace(',', ''))  
 ])  
 actual\_igst = *float*(*row\_data*.get('IGST\_SUM', '0').replace(',', ''))  
  
 calc\_cgst = *float*(*row\_data*.get('CGST\_AS\_CGST', '0').replace(',', ''))  
 actual\_cgst = *float*(*row\_data*.get('CGST\_SUM', '0').replace(',', ''))  
  
 calc\_sgst = *float*(*row\_data*.get('SGST\_UTGST\_AS\_SGST\_UTGST', '0').replace(',', ''))  
 actual\_sgst = *float*(*row\_data*.get('SGST\_UTGST\_SUM', '0').replace(',', ''))  
  
 *# Check for mismatches* if not math.isclose(calc\_igst, actual\_igst, rel\_tol=0.01):  
 logging.warning(f"IGST\_SUM mismatch: Calculated {calc\_igst} vs {actual\_igst}")  
  
 if not math.isclose(calc\_cgst, actual\_cgst, rel\_tol=0.01):  
 logging.warning(f"CGST\_SUM mismatch: Calculated {calc\_cgst} vs {actual\_cgst}")  
  
 if not math.isclose(calc\_sgst, actual\_sgst, rel\_tol=0.01):  
 logging.warning(f"SGST\_UTGST\_SUM mismatch: Calculated {calc\_sgst} vs {actual\_sgst}")  
  
 except *Exception* as e:  
 logging.error(f"Validation error: {*str*(e)}")  
 return False  
  
  
def prepare\_row\_data(*row*, *template\_placeholders*=None, *is\_eligible*=True):  
 prefix = "ELIGIBLE\_" if *is\_eligible* else "INELIGIBLE\_"  
 row\_data = {}  
  
 *# Process individual tax components* tax\_components = {  
 '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'  
 }  
  
 *# Process each tax component* for placeholder, col in tax\_components.items():  
 if col in *row*:  
 row\_data[placeholder] = format\_value(*row*[col], placeholder)  
 else:  
 row\_data[placeholder] = "0.00"  
  
 *# Calculate sums correctly* try:  
 *# IGST SUM (sum of IGST\_AS\_IGST, CGST\_AS\_IGST, SGST\_AS\_IGST)* igst\_sum = *sum*([  
 *float*(*row*.get(f'{prefix}IGST\_AS\_IGST', 0)),  
 *float*(*row*.get(f'{prefix}CGST\_AS\_IGST', 0)),  
 *float*(*row*.get(f'{prefix}SGST\_AS\_IGST', 0))  
 ])  
 row\_data['IGST\_SUM'] = format\_value(igst\_sum, 'IGST\_SUM')  
  
 *# CGST SUM (just CGST\_AS\_CGST)* cgst\_sum = *float*(*row*.get(f'{prefix}CGST\_AS\_CGST', 0))  
 row\_data['CGST\_SUM'] = format\_value(cgst\_sum, 'CGST\_SUM')  
  
 *# SGST/UTGST SUM (just SGST\_UTGST\_AS\_SGST\_UTGST)* sgst\_sum = *float*(*row*.get(f'{prefix}SGST\_UTGST\_AS\_SGST\_UTGST', 0))  
 row\_data['SGST\_UTGST\_SUM'] = format\_value(sgst\_sum, 'SGST\_UTGST\_SUM')  
  
 *# TOTAL AMOUNT (sum of all sums)* total\_amount = igst\_sum + cgst\_sum + sgst\_sum  
 row\_data['AMOUNT'] = format\_value(total\_amount, 'AMOUNT')  
  
 except *Exception* as e:  
 logging.error(f"Error calculating sums: {*str*(e)}")  
 row\_data['IGST\_SUM'] = "0.00"  
 row\_data['CGST\_SUM'] = "0.00"  
 row\_data['SGST\_UTGST\_SUM'] = "0.00"  
 row\_data['AMOUNT'] = "0.00"  
  
 *# 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)  
  
 *# First process all tax components* for placeholder, col in tax\_components.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\_AS\_IGST', 0)),  
 *float*(*row*.get(f'{prefix}CGST\_AS\_IGST', 0)),  
 *float*(*row*.get(f'{prefix}SGST\_AS\_IGST', 0)),  
 *float*(*row*.get(f'{prefix}CGST\_AS\_CGST', 0)),  
 *float*(*row*.get(f'{prefix}SGST\_UTGST\_AS\_SGST\_UTGST', 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"  
  
 *# Generate amount in words if needed  
 # In prepare\_row\_data function* if *any*('amount\_in\_words' in ph.lower() for ph in (*template\_placeholders* or [])):  
 try:  
 amount\_str = row\_data.get('AMOUNT', '0').replace(',', '').replace('[', '').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 "  
 f"{paise\_words} Paise Only"  
 )  
 except *Exception* as e:  
 logging.error(f"Amount conversion error: {*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 generate\_documents(*input\_excel*, *output\_folder*=None):  
 *"""  
 Generate ISD documents from Excel data with custom output folder support  
 Args:  
 input\_excel (str): Path to input Excel file  
 output\_folder (str): Custom output directory (optional)  
 Returns:  
 tuple: (success\_count, error\_count, output\_folder\_path)  
 """* from openpyxl import load\_workbook  
 from docx import Document  
 import pandas as pd  
 import logging  
 from num2words import num2words  
  
 *# Configure logging* logging.basicConfig(  
 level=logging.INFO,  
 format='%(asctime)s - %(levelname)s - %(message)s',  
 filename='document\_generator.log'  
 )  
  
 *# Validate input file* if not os.path.exists(*input\_excel*):  
 logging.error(f"Input file not found: {*input\_excel*}")  
 raise *FileNotFoundError*(f"Input file not found: {*input\_excel*}")  
  
 *# Set up output folders* if not *output\_folder*:  
 output\_folder = filedialog.askdirectory(title="Select Output Folder for ISD Documents")  
 if not *output\_folder*:  
 logging.warning("User cancelled output folder selection")  
 return 0, 0, None  
  
 eligible\_folder = os.path.join(*output\_folder*, "Eligible")  
 ineligible\_folder = os.path.join(*output\_folder*, "Ineligible")  
  
 try:  
 os.makedirs(eligible\_folder, exist\_ok=True)  
 os.makedirs(ineligible\_folder, exist\_ok=True)  
 except *PermissionError*:  
 logging.error(f"Permission denied creating folders in: {*output\_folder*}")  
 raise *PermissionError*(f"Cannot create folders in: {*output\_folder*}")  
  
 *# Load Excel data with formula support* try:  
 *# First pass to get calculated values* wb\_calc = load\_workbook(*input\_excel*, data\_only=True)  
 ws\_calc = wb\_calc.active  
 calculated\_data = *list*(ws\_calc.values)  
 headers = [*str*(h).strip().upper().replace(' ', '\_') for h in calculated\_data[0]]  
 calculated\_rows = calculated\_data[1:]  
  
 *# Second pass to preserve formulas if needed* wb\_formulas = load\_workbook(*input\_excel*, data\_only=False)  
 ws\_formulas = wb\_formulas.active  
 except *Exception* as e:  
 logging.error(f"Excel loading error: {*str*(e)}", exc\_info=True)  
 raise *RuntimeError*(f"Failed to load Excel file: {*str*(e)}")  
  
 success\_count = 0  
 error\_count = 0  
  
 *# Process each row* for row\_idx, row\_data in *enumerate*(calculated\_rows, start=1):  
 try:  
 *# Create dictionary of header:value pairs* row\_dict = *dict*(*zip*(headers, row\_data))  
  
 *# Skip empty rows* if not any(row\_dict.values()):  
 continue  
  
 *# Determine document type* doc\_type = str(row\_dict.get('ELIGIBLE/INELIGIBLE', '')).strip().lower()  
 if doc\_type not in ['eligible', 'ineligible']:  
 logging.warning(f"Row {row\_idx}: Invalid document type '{doc\_type}'")  
 error\_count += 1  
 continue  
  
 *# Load appropriate template* template\_file = f"{doc\_type}\_template.docx"  
 template\_path = os.path.join("templates", template\_file)  
  
 if not os.path.exists(template\_path):  
 logging.error(f"Template not found: {template\_path}")  
 raise FileNotFoundError(f"Template not found: {template\_path}")  
  
 doc = Document(template\_path)  
  
 *# Format numeric values* def format\_currency(value):  
 try:  
 return "{:,.2f}".format(float(value)) if value not in [None, ""] else "0.00"  
 except (ValueError, TypeError):  
 return "0.00"  
  
 *# Prepare replacements with fallbacks* replacements = {  
 'INVOICE\_NUMBER': str(row\_dict.get('INVOICE\_NUMBER', '')).strip(),  
 'INVOICE\_DATE': pd.to\_datetime(row\_dict.get('INVOICE\_DATE')).strftime('%d-%m-%Y') if row\_dict.get(  
 'INVOICE\_DATE') else '',  
 'ISD\_DISTRIBUTOR\_GSTIN': str(row\_dict.get('ISD\_DISTRIBUTOR\_GSTIN', '')).strip(),  
 'ISD\_DISTRIBUTOR\_NAME': str(row\_dict.get('ISD\_DISTRIBUTOR\_NAME', '')).strip(),  
 'CREDIT\_RECIPIENT\_GSTIN': str(row\_dict.get('CREDIT\_RECIPIENT\_GSTIN', '')).strip(),  
 'CREDIT\_RECIPIENT\_NAME': str(row\_dict.get('CREDIT\_RECIPIENT\_NAME', '')).strip(),  
 'IGST\_AS\_IGST': format\_currency(row\_dict.get('IGST\_AS\_IGST')),  
 'CGST\_AS\_IGST': format\_currency(row\_dict.get('CGST\_AS\_IGST')),  
 'SGST\_UTGST\_AS\_IGST': format\_currency(row\_dict.get('SGST\_UTGST\_AS\_IGST')),  
 'IGST\_SUM': format\_currency(row\_dict.get('IGST\_SUM')),  
 'CGST\_AS\_CGST': format\_currency(row\_dict.get('CGST\_AS\_CGST')),  
 'CGST\_SUM': format\_currency(row\_dict.get('CGST\_SUM')),  
 'SGST\_UTGST\_AS\_SGST\_UTGST': format\_currency(row\_dict.get('SGST\_UTGST\_AS\_SGST\_UTGST')),  
 'SGST\_UTGST\_SUM': format\_currency(row\_dict.get('SGST\_UTGST\_SUM')),  
 'AMOUNT': format\_currency(row\_dict.get('AMOUNT')),  
 'REG\_OFFICE': str(row\_dict.get('REG\_OFFICE', '')).strip(),  
 'CIN': str(row\_dict.get('CIN', '')).strip(),  
 'E\_MAIL': str(row\_dict.get('E\_MAIL', '')).strip(),  
 'WEBSITE': str(row\_dict.get('WEBSITE', '')).strip()  
 }  
  
 *# Add amount in words* try:  
 amount = float(str(row\_dict.get('AMOUNT', 0)).replace(',', ''))  
 replacements['AMOUNT\_IN\_WORDS'] = amount\_to\_words(amount)  
 except:  
 replacements['AMOUNT\_IN\_WORDS'] = "Rupees Only"  
  
 *# Replace placeholders in document* replace\_all\_placeholders(doc, replacements)  
  
 *# Generate output filename* timestamp = datetime.now().strftime('%Y%m%d\_%H%M%S')  
 invoice\_num = replacements['INVOICE\_NUMBER'] or f"ROW\_{row\_idx}"  
 output\_filename = f"{doc\_type.upper()}\_ISD\_{invoice\_num}\_{timestamp}.docx"  
  
 *# Save to appropriate folder* output\_path = os.path.join(  
 eligible\_folder if doc\_type == 'eligible' else ineligible\_folder,  
 output\_filename  
 )  
  
 doc.save(output\_path)  
 logging.info(f"Generated: {output\_path}")  
 success\_count += 1  
  
 except Exception as e:  
 logging.error(f"Error processing row {row\_idx}: {str(e)}", exc\_info=True)  
 error\_count += 1  
 continue  
  
 logging.info(f"Processing complete. Success: {success\_count}, Errors: {error\_count}")  
 return success\_count, error\_count, output\_folder  
  
  
def amount\_to\_words(amount):  
 *"""Convert numeric amount to words"""* try:  
 if amount % 1 == 0:  
 return f"{num2words(int(amount), lang='en\_IN').title()} Rupees Only"  
 else:  
 rupees = int(amount)  
 paise = round((amount - rupees) \* 100)  
 return (f"{num2words(rupees, lang='en\_IN').title()} Rupees and "  
 f"{num2words(paise, lang='en\_IN').title()} Paise Only")  
 except:  
 return "Rupees Only"  
  
  
*# In data\_mapper.py*def replace\_all\_placeholders(doc, replacements):  
 *"""  
 Enhanced replacement function that:  
 1. Handles both {{placeholder}} and {[placeholder]} formats  
 2. Preserves formatting during replacement  
 3. Processes all document components efficiently  
 4. Logs any unmatched placeholders  
  
 Args:  
 doc: python-docx Document object  
 replacements: dict of {placeholder: value} pairs  
 """  
 # Track which placeholders we actually replaced* used\_placeholders = set()  
  
 def process\_text(text):  
 *"""Inner function to process text with all replacements"""* new\_text = text  
 for ph, value in replacements.items():  
 *# Handle both {{ph}} and {[ph]} formats* for fmt in [f"{{{{{ph}}}}}", f"{{[{ph}]}}"]:  
 if fmt in new\_text:  
 new\_text = new\_text.replace(fmt, str(value))  
 used\_placeholders.add(ph)  
 return new\_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],  
 \*[section.first\_page\_header.paragraphs for section in doc.sections],  
 \*[section.first\_page\_footer.paragraphs for section in doc.sections]  
 ]  
  
 for paragraphs in components:  
 for paragraph in paragraphs:  
 *# Combine runs first to handle placeholders split across runs* full\_text = ''.join(run.text for run in paragraph.runs)  
 if not any(ph in full\_text for ph in replacements):  
 continue  
  
 *# Clear existing runs* paragraph.clear()  
  
 *# Process the full text with replacements* processed\_text = process\_text(full\_text)  
  
 *# Add new run with processed text* if processed\_text:  
 run = paragraph.add\_run(processed\_text)  
 *# Preserve formatting from first original run if exists* if paragraph.runs and paragraph.runs[0].font.name:  
 run.font.name = paragraph.runs[0].font.name  
  
 *# Log any unused placeholders from the replacements dict* unused = set(replacements.keys()) - used\_placeholders  
 if unused:  
 logging.warning(f"Unused placeholders in replacements: {unused}")  
  
 *# Log any remaining placeholders in document (not found in replacements)* remaining\_placeholders = scan\_template\_placeholders(doc)  
 if remaining\_placeholders:  
 logging.warning(f"Unreplaced placeholders remain in document: {remaining\_placeholders}")  
  
 return True  
  
  
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:  
 *# Remove any existing formatting* if isinstance(value, str):  
 value = value.replace(',', '').replace('[', '').replace(']', '')  
 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*[0:25]}"  
  
 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}")

1. 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

1. 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."""  
 # Normalize column names  
 df*.columns = [  
 col.strip()  
 .upper()  
 .replace(' ', '\_')  
 .replace('-', '\_')  
 .replace('.', '')  
 for col in *df*.columns  
 ]  
  
 *# Clean string values (only for object/string columns)* for col in *df*.columns:  
 if *df*[col].dtype == 'object':  
 *df*[col] = *df*[col].astype(*str*).str.strip()  
 *df*[col] = *df*[col].replace({'': None, 'nan': None, 'None': None})  
  
 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 (only if the field exists and is string type)* gstin\_fields = ['ISD\_DISTRIBUTOR\_GSTIN', 'CREDIT\_RECIPIENT\_GSTIN']  
 for field in gstin\_fields:  
 if field in *row* and *isinstance*(*row*[field], *str*):  
 if 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']:  
 *# First read the file without forcing string dtype* df = pd.read\_excel(  
 *file\_path*,  
 engine='openpyxl',  
 header=0,  
 skiprows=[1], *# Skip the tax type labels row* na\_values=['', 'NA', 'N/A', 'NULL', 'NaN'],  
 keep\_default\_na=False  
 )  
  
 *# Convert specific columns to numeric* 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:  
 if col in df.columns:  
 df[col] = pd.to\_numeric(df[col], errors='coerce').fillna(0)  
  
 logging.info(f"Successfully loaded Excel file: {*file\_path*}")  
  
 elif file\_ext == '.csv':  
 df = pd.read\_csv(  
 *file\_path*,  
 encoding='utf-8',  
 na\_values=['', 'NA', 'N/A', 'NULL', 'NaN'],  
 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)}")  
 raise *ValueError*(f"Row {idx + 1} invalid: {*str*(e)}")  
  
 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

1. 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)