PDF TOOLKIT

By Joydeep Dutta

25BCY10002

For VITyarthi

Course: Python Essentials

1. Introduction

In the modern digital landscape, the Portable Document Format (PDF) is the standard for document sharing. However, manipulating these files—merging reports, extracting specific pages, or securing sensitive data—often requires expensive proprietary software (like Adobe Acrobat) or risky online tools that compromise data privacy.

The **PDF Toolkit** is a lightweight, offline desktop application developed in Python. It provides essential PDF manipulation capabilities through a user-friendly interface, eliminating the need for internet access or paid subscriptions. It bridges the gap between command-line efficiency and graphical usability.

2. Problem Statement

Users frequently face the following challenges when managing PDF files:

- **Fragmentation:** Related documents (e.g., invoices, lecture notes) are often scattered across multiple files.
- **Bloat:** Large documents often contain unnecessary pages that need to be removed.
- **Security Risks:** Using free online PDF mergers requires uploading sensitive documents to unknown servers.
- **Accessibility:** Command-line tools (CLI) are powerful but difficult for non-technical users to operate.

Goal: To create a standalone executable (.exe) that performs local, secure, and fast PDF operations with a simple Graphical User Interface (GUI).

3. Functional Requirements

The system must perform the following core functions:

- **Merge PDFs:** Combine all PDF files within a user-selected folder into a single document in alphanumeric order.
- **Split PDFs:** Separate every page of a selected PDF into individual files.
- **Extract Range:** specific pages (e.g., pages 3 to 5) from a larger document and save them as a new file.
- **Encrypt/Lock:** Apply AES-128 encryption to a PDF using a user-defined password.
- **GUI Selection:** Allow users to select folders and files using Windows native dialog boxes rather than typing paths manually.
- **Safety Checks:** Prevent overwriting existing files and ensure valid page ranges are entered.

4. Non-Functional Requirements

- **Performance:** The application should process average-sized documents (under 50MB) in under 5 seconds.
- **Security:** All processing must happen locally on the user's machine; no data should be transmitted to the cloud.
- **Portability:** The final application must be a standalone .exe file that runs on Windows without requiring a Python installation.
- **Usability:** The interface should be text-based but intuitive, with clear prompts and error messages.
- **Reliability:** The system must handle exceptions (e.g., corrupt files, missing paths) without crashing.

5. System Architecture

The application follows a modular procedural architecture:

I. **Presentation Layer (UI):** A Console-based menu driven system integrated with tkinter dialogs for file/folder selection.

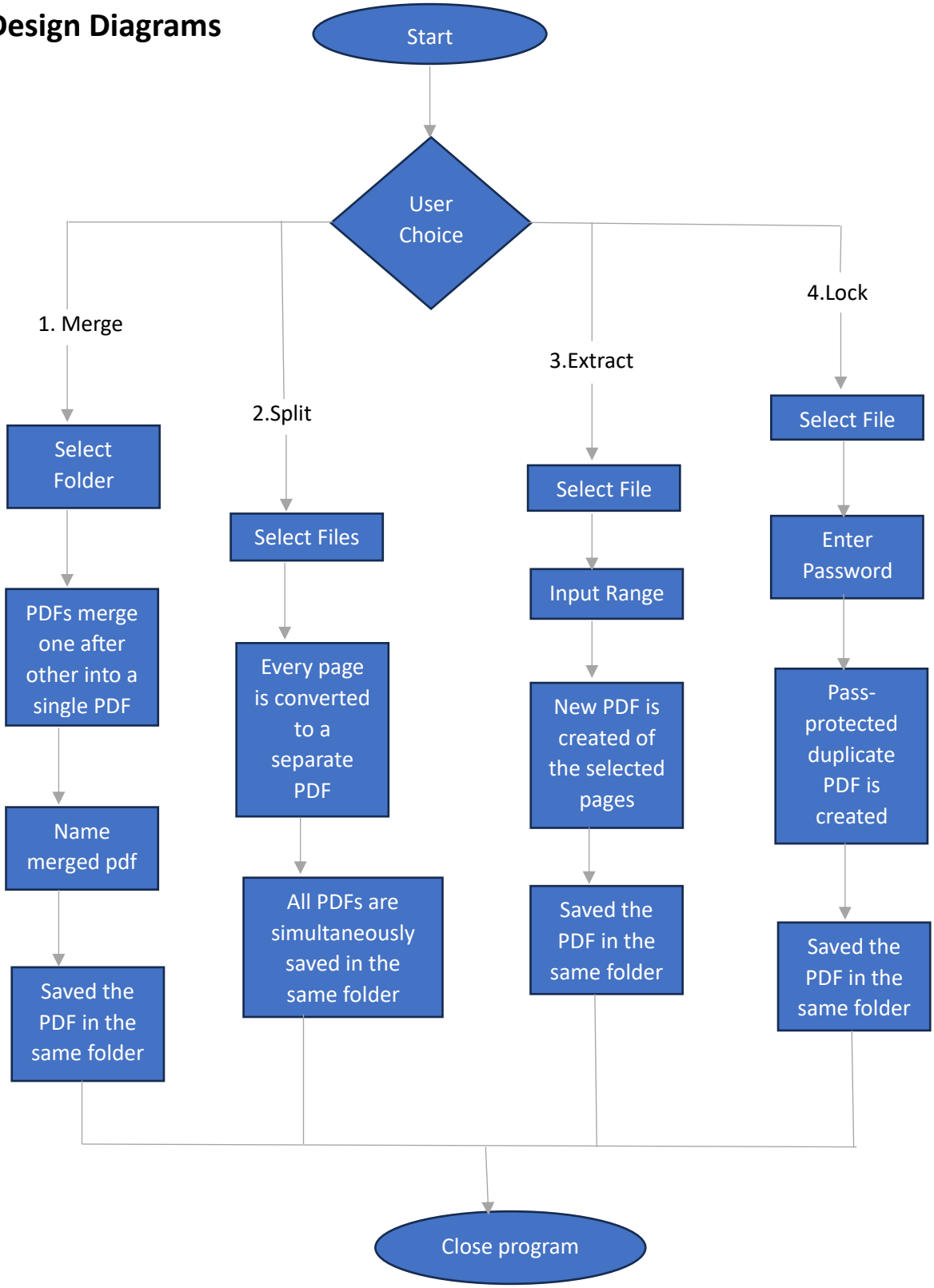
II. **Logic Layer (Controller):** Python functions (mergepdfs, splitpdf, lockpdf) that handle input validation and workflow logic.

III. **Data Access Layer:** The pypdf library acts as the interface between the Python script and the binary PDF files on the hard drive.

IV. **File System:** The OS acts as the storage medium.

Flow: User Input → Input Validation → PyPDF Processing → File I/O → Feedback to User.

6. Design Diagrams



7. Design Decisions & Rationale

- **Language: Python**
    - *Rationale:* Python offers the strongest libraries for file manipulation (os, shutil) and PDF handling, allowing for rapid development.
- **Library: pypdf (formerly PyPDF2)**
    - *Rationale:* It is a pure-Python library (no C dependencies), making it easier to compile into an exe. It supports both reading, writing, and encryption.
- **GUI: Tkinter (FileDialog only)**
    - *Rationale:* We chose a hybrid approach. A full GUI is complex to build. Using a Console for the menu but tkinter for file selection gives the best balance of development speed and user experience.
- **Deployment: PyInstaller**
    - *Rationale:* It bundles the Python interpreter and all dependencies into a single file, ensuring the app works on any Windows PC.

8. Implementation Details

The project is contained in a single script PDF TOOLKIT.py

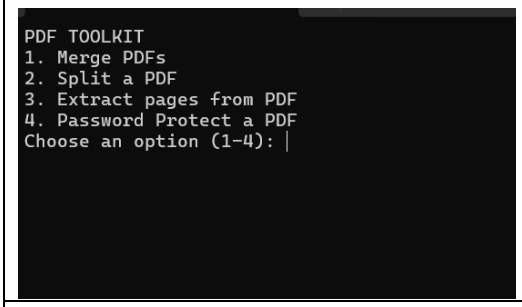
```
import os
from pypdf import PdfWriter, PdfReader
import tkinter as tk
from tkinter import filedialog
```

Core Logic for Merge: The merge function utilizes os.listdir() to populate a list, sorts it to ensure page order, and uses PdfWriter.append() to stitch the binary streams together.

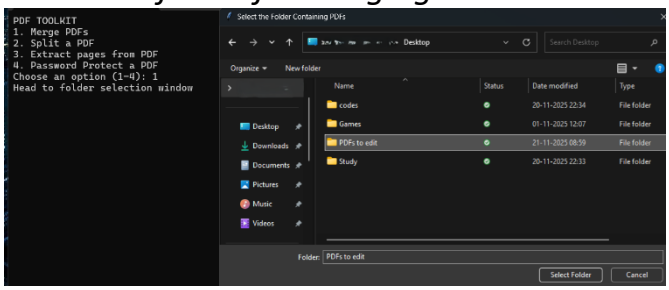
Core Logic for Encryption: Encryption is achieved by copying pages from a PdfReader object to a PdfWriter object, and then applying writer.encrypt(password) before the binary write operation.

9. Screenshots / Results

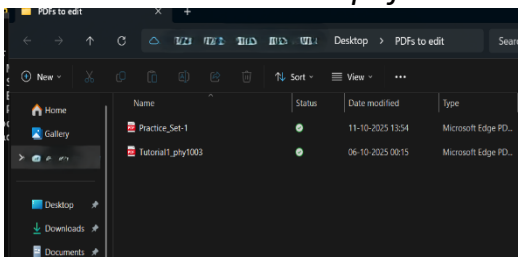
01. Menu



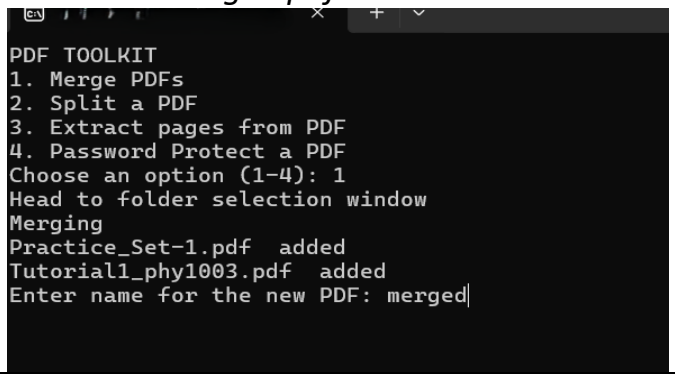
02. Select folder for merging



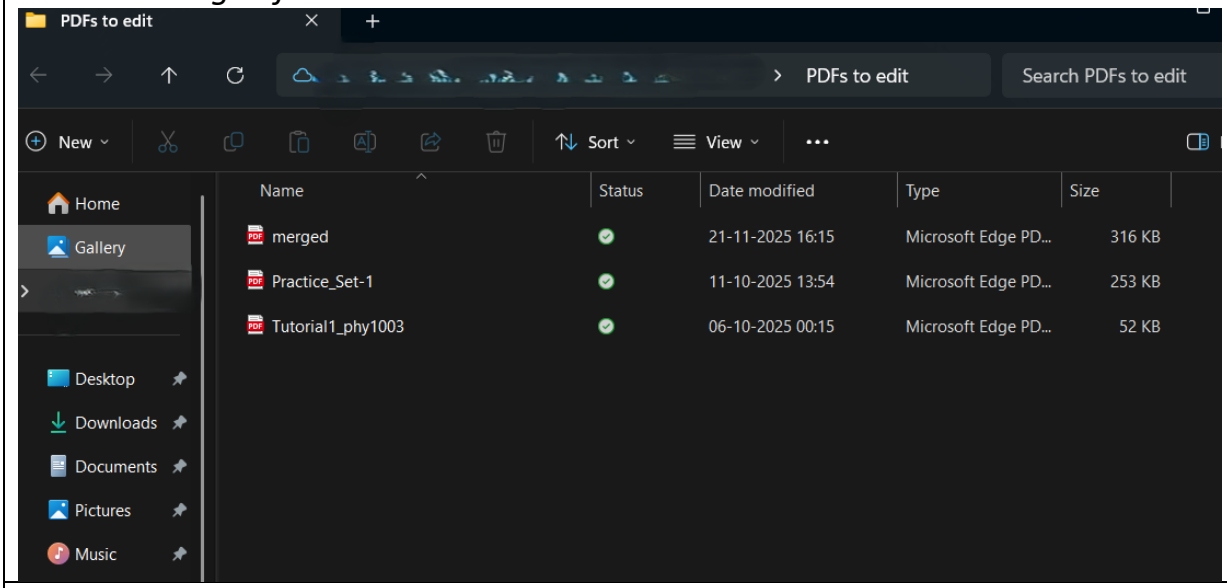
03. Folder should have pdfs



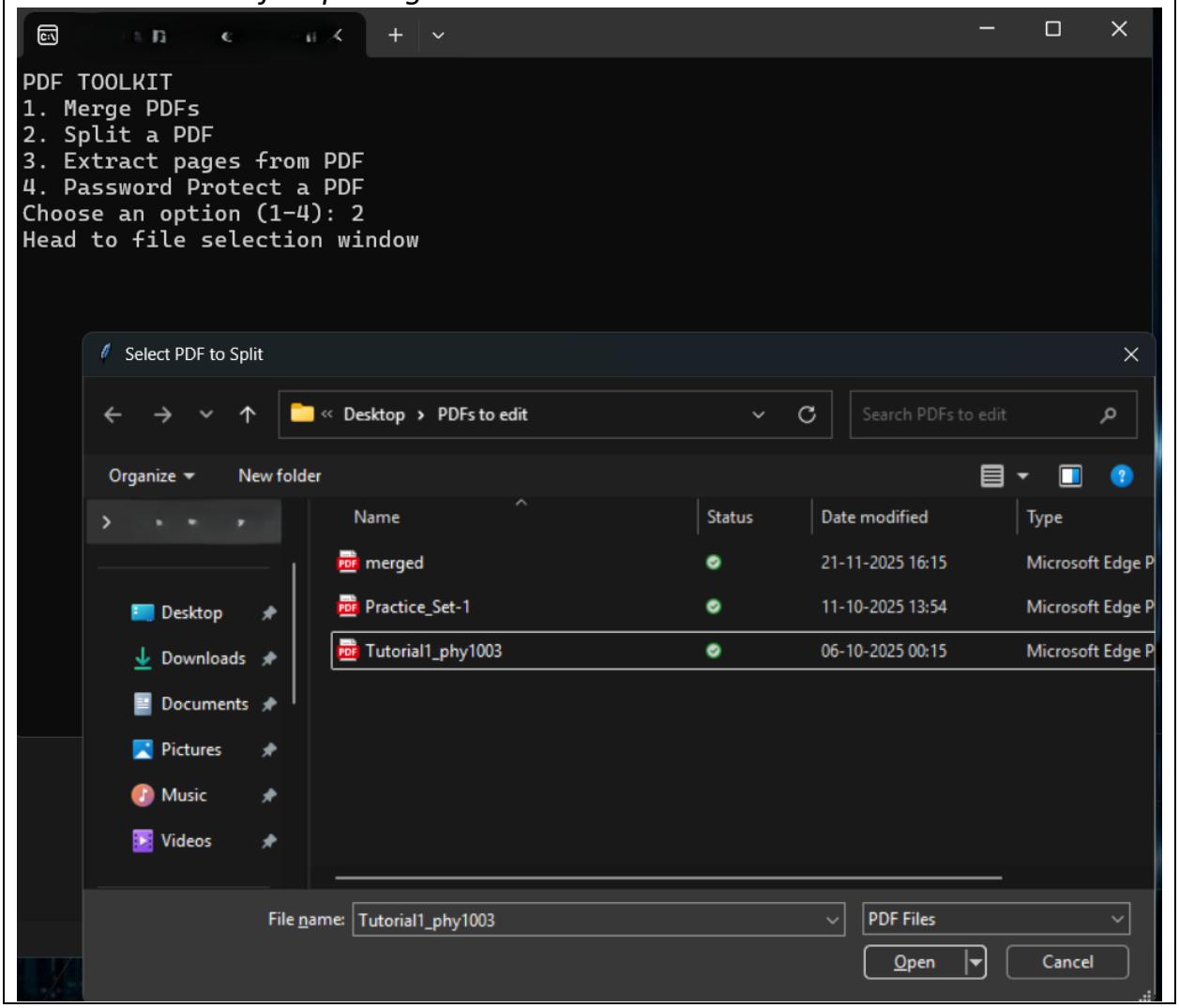
04. Name merged pdf



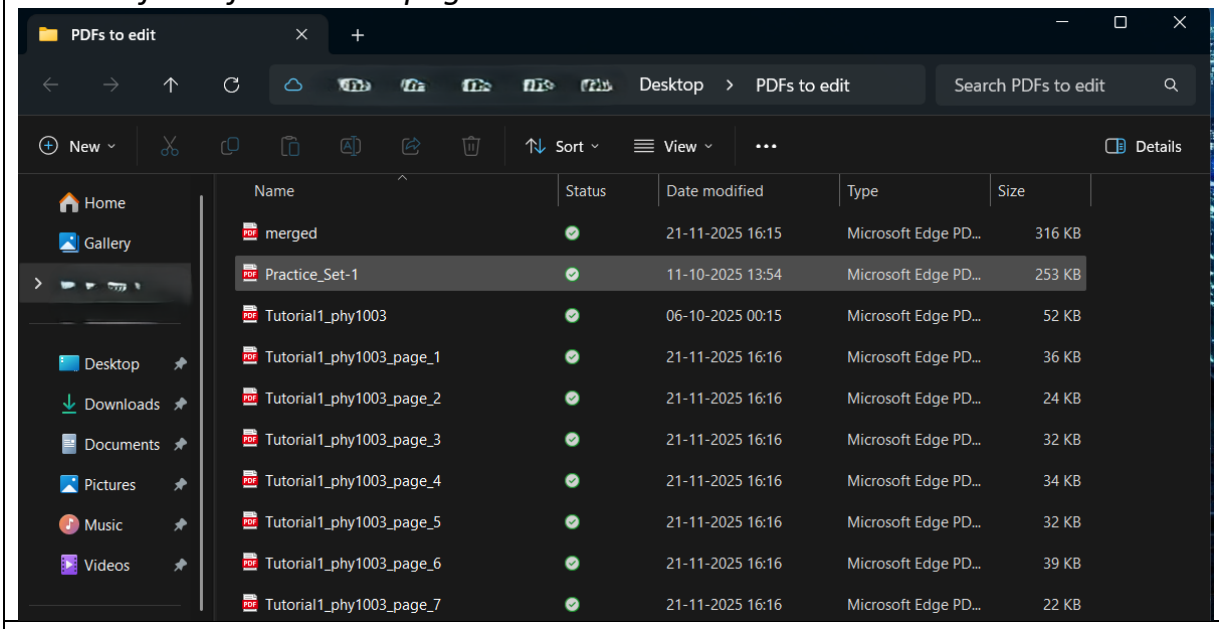
05. New merged file created



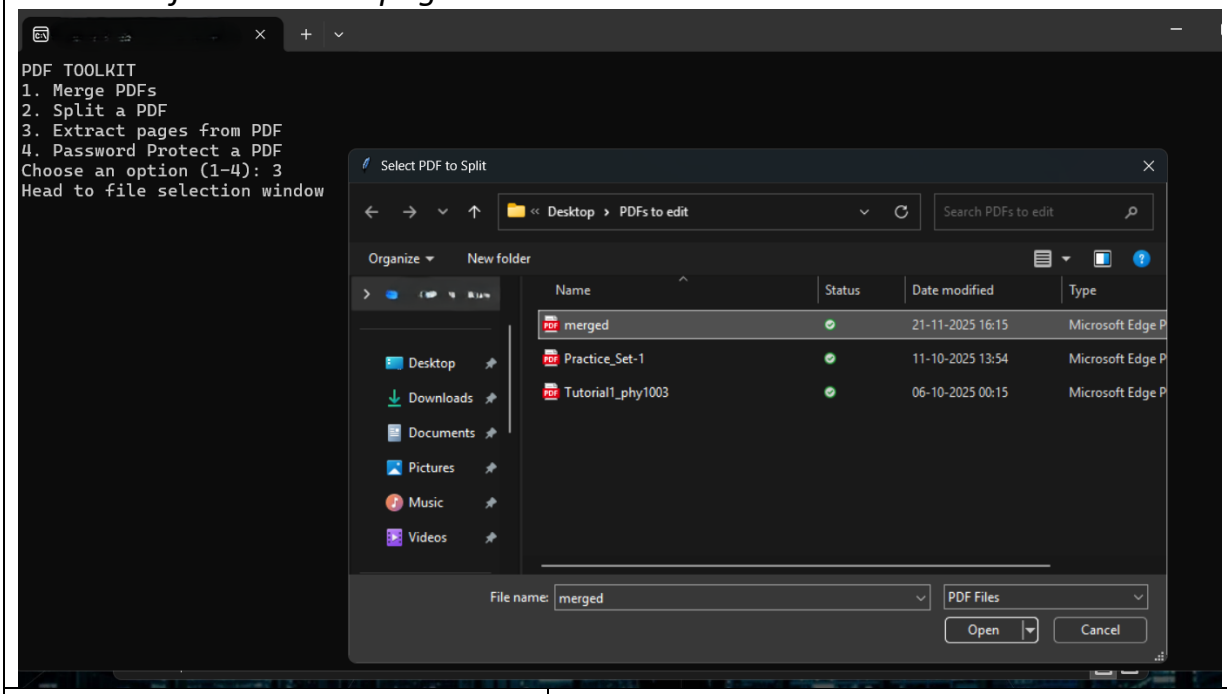
06. File Selection for splitting



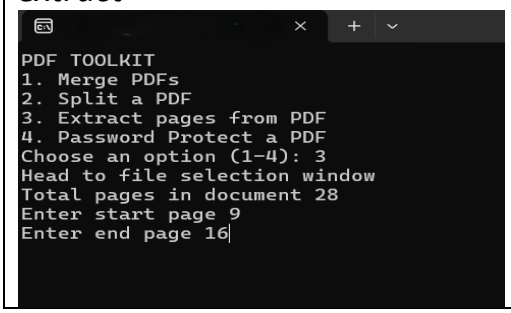
07. PDF files of individual pages created



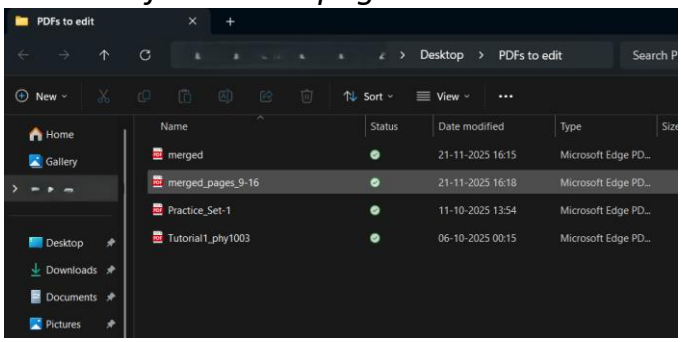
08. Select file to extract pages



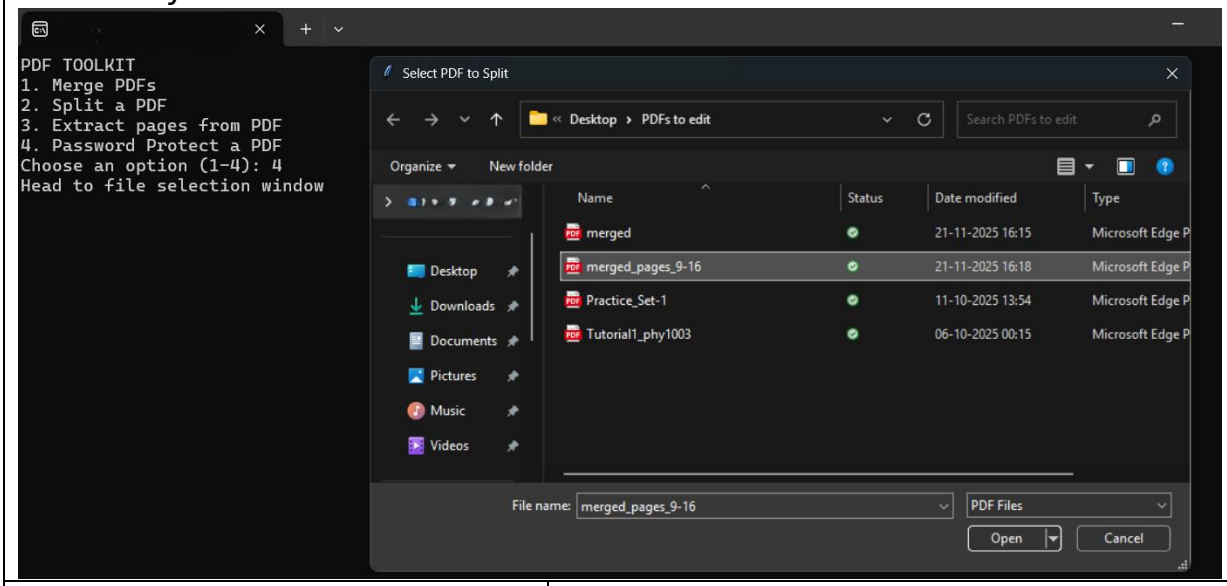
09. Enter range of pages to extract



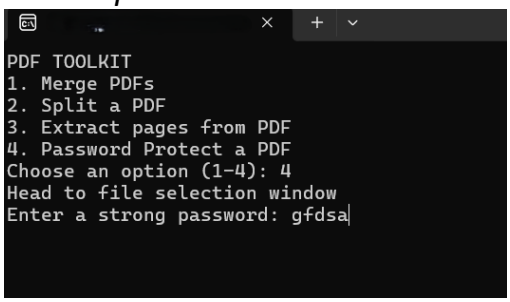
10. PDF of extracted pages created



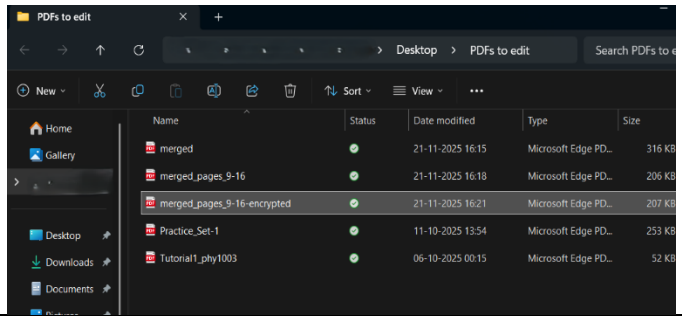
11. Select file to lock



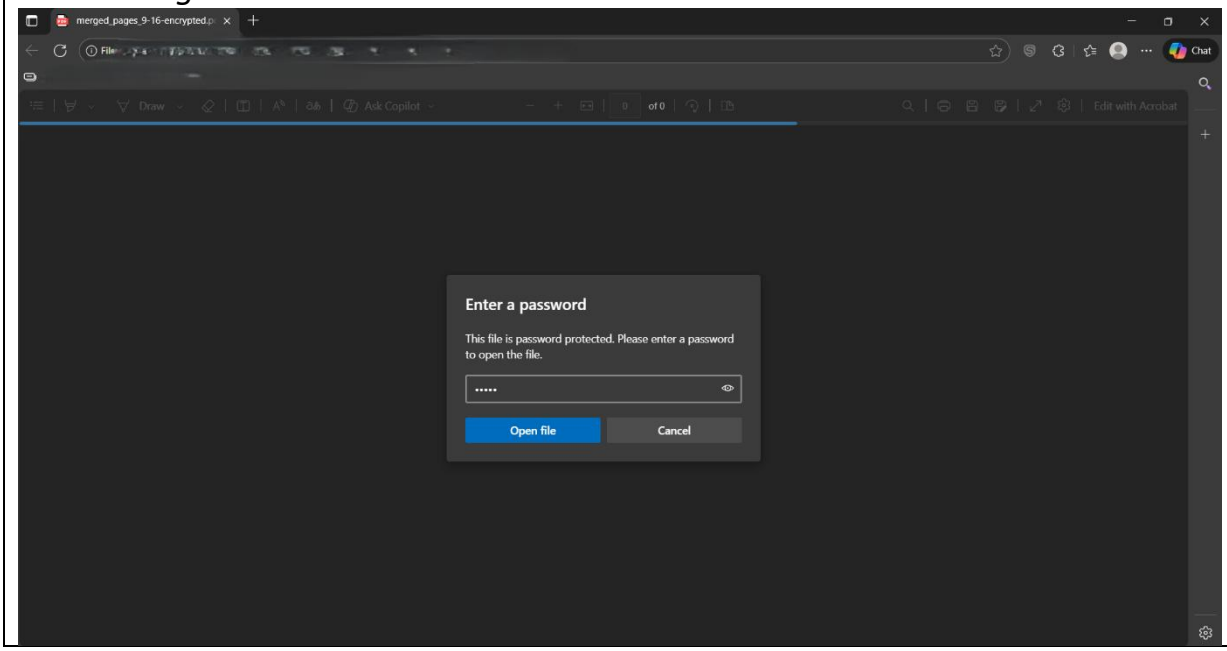
12. Set password



13. Locked PDF created



14. Viewing locked PDF



10. Testing Approach

Testing was conducted manually using "Black Box" techniques:

I. **Positive Testing:**
   - Selected a folder with 3 PDFs -> Result: Merged successfully.
   - Selected a 10-page PDF and extracted 2-5 -> Result: Created 4-page PDF correctly.

II. **Negative Testing:**
   - **Input:** Entering page range "10-5" (Start > End).
   - **Result:** System detected invalid range and displayed error message.
   - **Input:** Entering an empty password.
   - **Result:** System reprompted until a password was entered.

III. **Edge Cases:**
   - **Input:** Selecting a folder with zero PDFs.
   - **Result:** System printed "No PDF files found" and returned to menu safely.

11. Challenges Faced

During development, several technical challenges were encountered:

I. **PyInstaller sys.stdin Error:**
   - *Issue:* When compiling with --noconsole, the application crashed because input() functions require a terminal window to receive text.
   - *Solution:* compiled without the --noconsole flag to allow user interaction via the command prompt.

II. **File Sorting:**

- *Issue:* os.listdir() retrieves files in arbitrary order, causing merged PDFs to be jumbled.
- *Solution:* Implemented files.sort() and adopted a naming convention (01.pdf, 02.pdf) to ensure correct sequencing.

III. **Ghost GUI Windows:**

- *Issue:* Tkinter would leave a small blank window open after file selection.
- *Solution:* Implemented root.withdraw() to hide the main window during file dialog operations.

12. Learnings & Key Takeaways

- **Binary File Handling:** Learned the importance of rb (read binary) and wb (write binary) modes when handling non-text files.
- **Zero-based Indexing:** Gained a deeper understanding of how Python handles list indexes vs. how humans count pages (Mapping Page 1 to Index 0).
- **Executable Compilation:** Learned the process of bundling Python scripts with dependencies using PyInstaller and managing environmental variables.

13. Future Enhancements

To further improve the utility, the following features are proposed for version 2.0:

- **OCR Capabilities:** Integrating Tesseract-OCR to convert scanned image-based PDFs into searchable text.
- **Watermarking:** Adding a feature to stamp logos or "Confidential" text over document pages.
- **Metadata Editor:** A tool to view and strip hidden metadata (author, creation date) for privacy.

- **Drag-and-Drop GUI:** Migrating from a console menu to a full CustomTkinter interface with drag-and-drop support.

14. References

I. **Python Documentation:** https://docs.python.org/3/

II. **PyPDF Documentation:** https://pypdf.readthedocs.io/en/stable/

III. **PyInstaller Manual:** https://pyinstaller.org/en/stable/

IV. **Tkinter Resources:** https://tkdocs.com/