PharmaSUG 2024 - Paper AP- 253

Build Your Own PDF Generator: A Practical Demonstration of Free and Open-Source Tools

James Austrow, C5Research, Cleveland Clinic

ABSTRACT

The PDF is one of the most ubiquitous file formats and can be read on nearly every computing platform. So how, in the year 2024, can it still be so inconvenient to perform basic editing tasks such as concatenating and merging files, inserting page numbers, and creating bookmarks? These features are often locked behind paid licenses in proprietary software or require that the documents be uploaded to a web server, the latter of which poses unacceptable security risks to many would-be users.

In fact, the PDF is a public standard and there exist free, open-source libraries that make it easy to build in-house solutions for these and many other common use cases. In this paper, we demonstrate how to use Python to automatically assemble multiple PDF documents into a unified, polished deliverable. No more than a basic familiarity with Python should be needed to follow along; we thoroughly explain every step from start to finish.

INTRODUCTION

The PDF is the de facto standard for document sharing. And yet, for any task other than reading, all the most readily-available options seem to be proprietary. As these tools are seldom concerned with interoperability, reliance on them also poses a significant barrier to automation and all of its benefits.

What we'd like to demonstrate here is that, in fact, there is another way. Anyone with an understanding of programming and a little resourcefulness can set up an automated document generator customized for their organization's specific needs.

We'll begin with an overview of previous work on this topic and discuss what shortcomings our approach addresses. We'll then take a brief tour of the history of the PDF. This context will help us understand the PDF's rise to ubiquity and explain why we don't need to rely on proprietary software.

Finally, we'll walk though how to set up a program that concatenates any number of PDF files, builds a bookmark outline, and inserts page numbering. We'll also illustrate how to build a custom title sheet and one way to organize the source documents into sections. We'll be using the Python programming language and two open source libraries: pypdf [6] and fpdf2 [7].

PRIOR WORK

There is no shortage of authors and programmers who have tackled the problem of consolidating multiple documents into a single report. Wang and Zhou [4] discuss a C# application they developed that both converts RTF files and combines them into one PDF. Shao and Zhang [3] present a SAS macro for collecting `.lst` outputs. Plevney [1] utilizes SAS and VBScript to preprocess RTFs for consolidation. Xie [5] merges RTFs directly using SAS, then converts to PDF using Powershell.

This list is hardly exhaustive; see Coar [2] for a method using ODS item stores and for their collection of references to yet more approaches.

Our method offers: - Minimal dependencies: we rely only on Python (which comes by default on most Linux distributions) and the ability to install a handful of packages. In particular, we do not require any software licenses (not even SAS). - Simplicity and ease of implementation. - The ability to inject arbitrary text and graphical content alongside the original documents.

Your documents may come in a form other than PDF, in which case they will need to be converted before proceeding as below. In particular, RTF and DOCX files are best handled using Powershell; see Xie [5] for an example script.

ABOUT PDF

The public history of the PDF began in 1992, when Adobe first announced the format in 1992 at tech expo Comdex/Fall [18]. (Incidentally, just six months prior, Microsoft first introduced Windows for PC at the spring meeting of that same conference [17].) The first release of PDF software came one year later [14], along with its technical specifications [19].

From the very beginning, Adobe's plan was to win widespread adoption through opening its standards while providing the best implementation. This strategy had been very successful for them with their preceding product, PostScript, which at one point had as many as 75 competitors and clones [13]. In a 2021 interview, Adobe cofounder John Warnock described their intentionality [12]:

> *We felt that if we wanted this broadly adopted that we had to do exactly the opposite of what Xerox wanted to do. We had to publish it. We had to make it very, very open — because the trick was to get both [software] application developers and operating system developers to support it. Without documentation that was never going to happen.*

Despite Adobe's efforts, however, initial adoption was slow. That all changed in 1996 when they gained none other than the IRS as a customer, who wanted to offer taxpayers the ability to download their own print-ready tax forms as PDFs [16]. On top of Adobe product installations for the 100,000 IRS employees, this move brought exposure of Adobe's product to over 100 million U.S families overnight, quite literally making them a household name.

Even with their ensuing success, Adobe kept the PDF standard open until ultimately relinquishing its control over to the ISO in 2007 [15]. As Duff Johnson, current CEO of the PDF Association, put it: "While PDF was invented by Adobe, the company no longer owns PDF. We all do." [20]

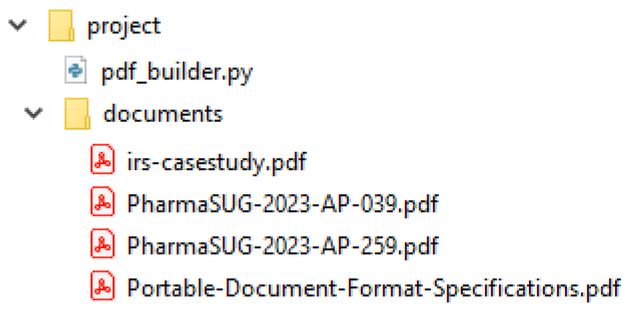
In the time since, a whole ecosystem of tools has grown under the sun of that shared ownership, only a handful of which we will demonstrate here.

WALKTHROUGH

PRELIMINARIES

Other than the Python standard library, we'll need just two packages. The first, `pypdf` [6], is great for working with existing PDF content: reading, appending, transforming, extracting text and images, merging pages, creating bookmarks, and more. The main thing it can't do is generate new PDFs from scratch; for that, we will be using the `fpdf2` [7] package. This library will be required for creating page numbers, a title page, and section separators.

Let's begin with the following project structure. This demo will use a couple of our references as source documents, but you should substitute your own files:



```
project
    pdf_builder.py
    documents
        irs-casestudy.pdf
        PharmaSUG-2023-AP-039.pdf
        PharmaSUG-2023-AP-259.pdf
        Portable-Document-Format-Specifications.pdf
```

Figure 1: Project root structure.

Install the required packages on the command line:

```
pip install pypdf fpdf2
```

Enter the following into the file pdf_builder.py as a program skeleton. We will fill in the rest of the code throughout the walkthrough:

```
import io # Used to merge new PDF content we create
from fpdf import FPDF, Align # For creating PDF content
from pypdf import PdfReader, PdfWriter # For file input and output
from pathlib import Path # Interfaces with the file system

# Main driver function
def build_pdf():
    pass

# Will create page numbers for merging onto each page
def make_page_number():
    pass

# Will create our title page
def make_title_page():
    pass

# Will create section separator pages
def make_section_page():
    pass

# Build the PDF and output as a file.
if __name__ == '__main__':
    output = build_pdf()
    print('Outputting final document.')
    output.write('result.pdf')
    output.close()
```

Note that the package we installed is named “fpdf2” but we import it in our actual program as “fpdf.” Yes, this is a bit confusing.

Any time while building the demo, you can test the program so far by invoking it from the command line:

```
python pdf_builder.py
```

CONCATENATING PDFS

We can accomplish basic document concatenation very simply. Just change the body of `build_pdf()` to this:

```
def build_pdf():

    output = PdfWriter() # our output pdf

    # Iterate documents
    for pdf in sorted(Path('documents').glob('*.pdf')):
        print(f'Adding {pdf.stem}')
```

```
            output.append_pages_from_reader(PdfReader(pdf))

        return output
```

It’s difficult to show the results in a screenshot, but if you’re following along and test the program at this point, you should get a nice concatenated file result.pdf in the project root directory.

Before we move on, let’s step through this code carefully to make sure we understand how it works. The first step is to instantiate an instance of the `PdfWriter` class:

```
output = PdfWriter() # our output pdf
```

This class is provided by `pypdf` as an interface for PDF output. As we’ll see, it makes most common tasks very convenient.

Next, we need to loop over each file in our documents directory. Let’s unpack this line from the inside out:

```
for pdf in sorted(Path('documents').glob('*.pdf')):
```

First, we open the directory `'documents'` using the `Path` class from the Python standard library. Then, the `glob()` function will give us an iterator over all the files in that directory that match a particular pattern. In this case, we want all the files that have a `.pdf` file extension. Finally, wrapping this in the `sorted()` function ensures that the documents are, well, sorted (alphabetically).

Within the loop, all we need to do is append the content from the current file to the `PdfWriter` object we created earlier. The `print()` call is just for tracking our progress; the second line is where we want to direct our attention:

```
output.append_pages_from_reader(PdfReader(pdf))
```

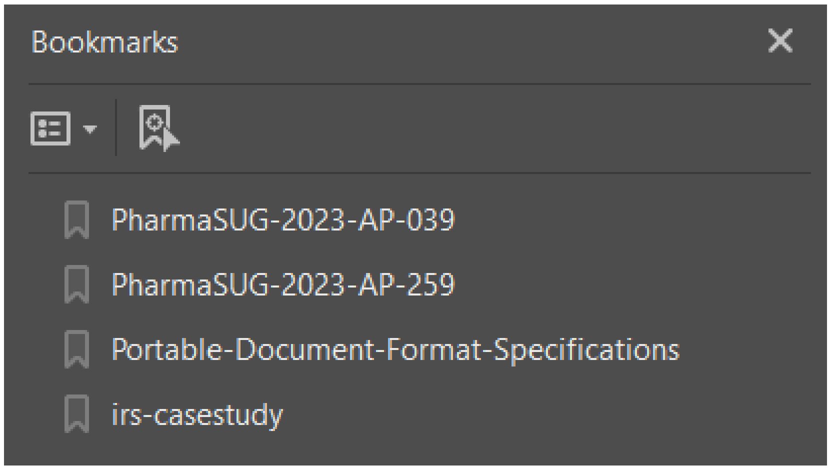
We accomplish two steps with this line. First, we create a `PdfReader` instance using the file handle from the loop header. This is another class provided by `pypdf` that makes it easy to read existing PDF files. Then, we stream the pages from that reader directly into our output object using its `append_pages_from_reader()` function, which just does exactly as its name suggests.

The net result of all of the above is that we stream each PDF from the subdirectory documents sequentially into the `PdfWriter`, in alphabetical order. After we return that writer object, the `if '__name__' == '__main__'` block at the bottom of our program will take care of actually writing the output as a file.

While we are here, it turns out it is very simple to add a bookmark for each incoming document. Let’s take care of that now by inserting one line into the for loop:

```
# Iterate documents
for pdf in sorted(Path('documents').glob('*.pdf')):
    print(f'Adding {pdf.stem}')
    output.add_outline_item(pdf.stem, len(output.pages)) # NEW LINE
    output.append_pages_from_reader(PdfReader(pdf))
```

Before explaining how this works, let’s show how the results appear. The generated PDF is the same as before, except with these entries as bookmarks:



PharmaSUG-2023-AP-039
PharmaSUG-2023-AP-259
Portable-Document-Format-Specifications
irs-casestudy

Figure 2: Basic bookmarks.

The function `add_outline_item()` accepts two arguments: the text to display as the bookmark title and the page number it should link to. For the text, we are using the stem property of the file handle, which just gives the name of the file without its extension. For the target page number, we use the current page count of our output object, which will update dynamically as we add new pages. Neat!

The code in this section alone is likely sufficient for many basic use cases, which shows how powerful these tools are for working with PDFs. The rest of the walkthrough will focus on polishing up our document with custom content and organization.

PAGE NUMBERS

Let's now suppose we would like each page in our consolidated PDF to show its number, relative to the whole document (as opposed to any document-specific page numbering that may already be present). Fortunately, `PdfWriter` provides a mechanism for merging the content streams of two PDFs, but we will first have to create a fresh PDF containing the content we want to merge. We'll be generating this content from scratch though, which `pypdf` can't help us with.

Enter `fpdf2`. We'll use this package to create a blank page, add our page number text to that page, and then merge this into our output object.

Let's first fill in the definition of `make_page_number()`. This will accept a single page and a text string as arguments, and create the new page to be merged with the original:

```
def make_page_number(source_page, text):

    # Obtain dimensions from the original page
    size = (source_page.mediabox.width, source_page.mediabox.height)
    ppi = 72 # convenience variable for points per inch

    # Initialize a new PDF
    pdf = FPDF(unit='pt', format=size)
    pdf.set_font('Courier', '', 11)
    pdf.set_auto_page_break(False)

    # Write the text
    pdf.add_page()
    pdf.set_y(size[1] - ppi//2) # half an inch from the bottom
```

```
        pdf.cell(w=0, text=text, align=Align.C) # centered

        # Open a reader to the pdf we just made and return the first (only)
    page
        return PdfReader(io.BytesIO(pdf.output())).pages[0]
```

This is the first time we're seeing `fpdf2` in action, but most of the commands are self-explanatory. We'll focus our discussion on just the ones that are a bit more mysterious.

First, we have:

```
    size = (source_page.mediabox.width, source_page.mediabox.height)
```

When we create our new page, we must make sure that it is the same size as the original page. Otherwise, our new text will not be positioned correctly once merged. The above line is just capturing the dimensions of the original page so that we can use them later.

Now, we need a brief aside to discuss a technical detail about PDF rendering. PDF uses a coordinate system internally to control the positions of content elements. The standard unit for this coordinate system is called a "point," and is equivalent to 1/72 inches. If we prefer to think about our text positioning in terms of inches, we need to take this conversion factor into account. That is the purpose of the following line:

```
    ppi = 72
```

Now we're ready to move on to the next item:

```
    pdf = FPDF(unit='pt', format=size)
```

All we are doing here is creating an instance of the FPDF class, which is what we use to build our page. We specify that we want to use the "point" unit discussed previously and to use the page dimensions captured from the original.

The rest of the function body is just basic commands to set the font, position, and text on the page. The final line deserves some focus, though:

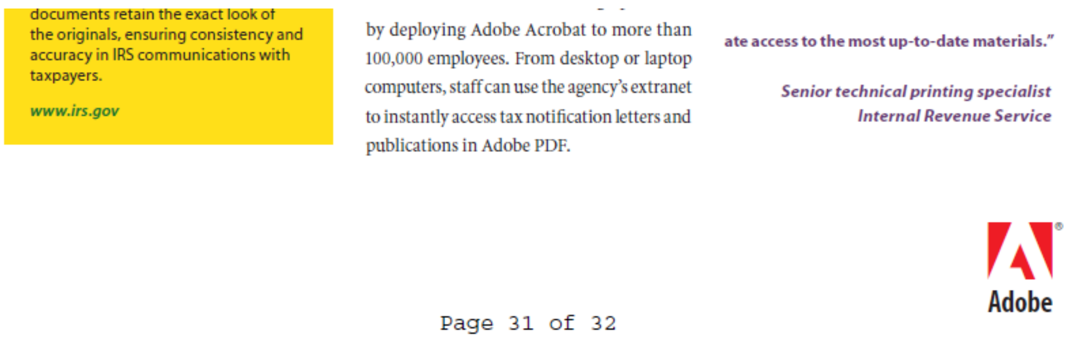
```
    return PdfReader(io.BytesIO(pdf.output())).pages[0]
```

This might look like a lot, but it's essentially just boilerplate that converts our content from an `FPDF` object into a `PdfReader`. From there, we access just the first page and return it.

Now we are ready to update our `build_pdf()` function to insert this content into the concatenated PDF:

```
    def build_pdf():

        output = PdfWriter() # our output pdf

        # Iterate documents
        for pdf in sorted(Path('documents').glob('*.pdf')):
            print(f'Adding {pdf.stem}')
            output.add_outline_item(pdf.stem, len(output.pages))
            output.append_pages_from_reader(PdfReader(pdf))

        ### NEW CODE
        # Write page numbers onto each output page
        print('Writing page numbers.')
        for i, page in enumerate(output.pages):
            text = f'Page {i+1} of {len(output.pages)}'
            page.merge_page(page2=make_page_number(page, text))
        ### END NEW CODE

        return output
```

Et voila! Behold our new page numbers:



documents retain the exact look of the originals, ensuring consistency and accuracy in IRS communications with taxpayers.

www.irs.gov

by deploying Adobe Acrobat to more than 100,000 employees. From desktop or laptop computers, staff can use the agency's extranet to instantly access tax notification letters and publications in Adobe PDF.

ate access to the most up-to-date materials."

Senior technical printing specialist
Internal Revenue Service

Page 31 of 32

Figure 3: Page numbers in the consolidated report.

As you can see, this step has been added *after* we finish appending all of the source documents. The new loop steps through each of the pages in our output object via the pages property, calls our `make_page_number()` function, then merges the two pages.

The results may not look like much, but it took a fair amount of legwork to get here. The good news is that we've now introduced all the important concepts we need for creating fancier content.

TITLE PAGE

The `fpdf2` package doesn't limit us to just text elements. We can also render many types of graphics, including images, lines, and shape primitives. We'll demonstrate a few of those options by creating a custom title page format for our document.

First, let's fill out the definition of `make_title_page()`:

```
def make_title_page():

    # Set page units and dimensions
    ppi = 72
    dim = (8.5*ppi, 11*ppi) # 8.5" by 11"

    pdf = FPDF(unit='pt', format=dim)
    pdf.set_margins(1*ppi, 1*ppi, 1*ppi) # 1" for left, top, and right

    # Draw all shapes and text white
    pdf.set_text_color(255)
    pdf.set_draw_color(255)
    pdf.set_fill_color(255)
    # Begin content
    pdf.add_page()
    pdf.image('../images/title-bg.png', x=0, y=0, w=dim[0], h=dim[1])
    pdf.image('../images/PharmaSUG_2024_logo_200px_trans.png', x=400, y
=20)

    pdf.set_font('Helvetica', 'B', 48)
    pdf.set_y(6.5*ppi)
    pdf.cell(w=0, text='PDF Demo', align=Align.L)

    pdf.set_font('Helvetica', 'B', 18)
    pdf.set_y(7.25*ppi)
    pdf.cell(w=0, text='Selected documents packet', align=Align.L)

    pdf.set_font('Helvetica', 'B', 14)
    pdf.set_y(10*ppi)
    pdf.cell(w=0, text='[put author here]', align=Align.L)
    pdf.cell(w=0, text='PharmaSUG 2024', align=Align.R)

    ypos = 7.75*ppi
    pdf.set_line_width(4)
    pdf.line(1*ppi, ypos, 7.5*ppi, ypos)

    # Open and return a reader to the pdf we just made
    return PdfReader(io.BytesIO(pdf.output()))
```

We're using two new notable functions here: `image()` and `line()`. When you try this yourself, you'll have to substitute your own images (and respective paths). You'll also most likely want to change the position and/or color of the textual elements depending on your background image. The rest is very similar to what we already used to make the page numbers.

The updates to `build_pdf()` are minimal:

```
def build_pdf():

    output = PdfWriter() # our output pdf

    ### NEW CODE
    # Start with title page
    output.append(make_title())
    no_pagenum = [0] # we don't want a page number on the title page
    ### END NEW CODE

    # Iterate documents
    for pdf in sorted(Path('documents').glob('*.pdf')):
        print(f'Adding {pdf.stem}')
        output.add_outline_item(pdf.stem, len(output.pages))
        output.append_pages_from_reader(PdfReader(pdf))
    # Write page numbers onto each output page
    print('Writing page numbers.')
    for i, page in enumerate(output.pages):
        ### NEW CODE
        if i in no_pagenum: continue # skip specified pages
        ### END NEW CODE
        text = f'Page {i+1} of {len(output.pages)}'
        page.merge_page(page2=make_page_number(page, text))

    return output
```

Look carefully! There are two places where we inserted new code. First is that we are now adding our title page before any of our source documents:

```
output.append(make_title())
```

We have also added this slightly curious line:

```
no_pagenum = [0]
```

The purpose of this is to list the pages that we don't want to have page numbers added to. Otherwise we will get a page number at the bottom of our title page, which we probably don't want. We set up this list to just have a 0 initially, as that is the index of the title page.

We then modify the page-numbering loop to care about this list:

```
if i in no_pagenum: continue # skip specified pages
```

The reason we've made this a list instead of just hardcoding to skip the first page will become clear in the next section.

And that's it! Here's what our cool new title page looks like:

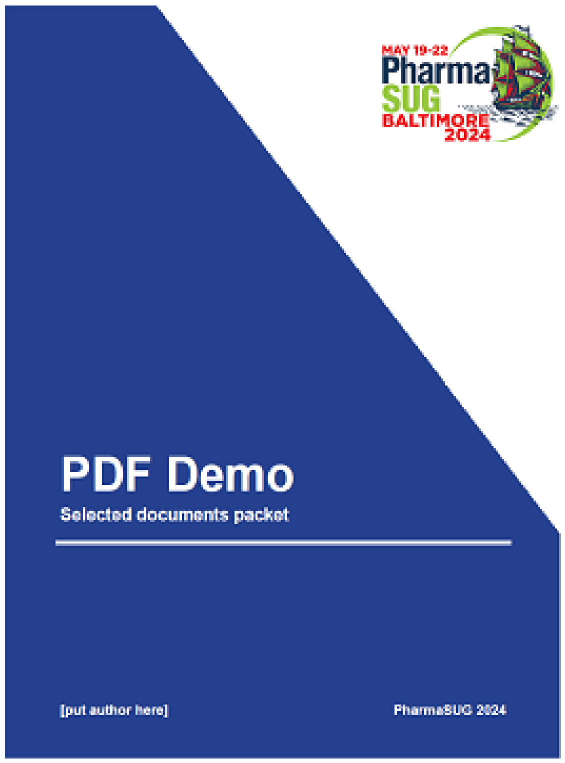


Figure 4: Custom title page.

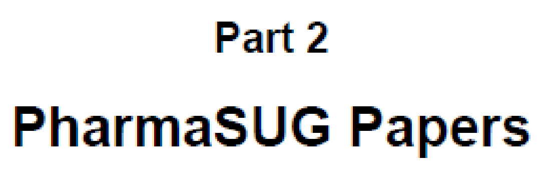
SECTION SEPARATORS

We'll finish up with something slightly more involved. We're going to introduce some structure to our document by organizing the sources into sections and grouping their bookmarks accordingly. We'll also create a custom separator page to insert at the beginning of each section.

Let's start with the most straightforward piece and fill out our definition of `make_section_page()` that will actually create the separator pages. We'll keep the design simple this time and just display the name (name) and number (k) of the section:

```
def make_section_page(k, name):

    ppi = 72
    dim = (8.5*ppi, 11*ppi)

    pdf = FPDF(unit='pt', format=dim)
    pdf.set_margins(1*ppi, 1*ppi, 1*ppi)
    pdf.add_page()

    pdf.set_font('Helvetica', 'B', 18)
    pdf.set_y(4.5*ppi)
    pdf.cell(w = 0, text=f'Part {k}', align=Align.C)

    pdf.set_font('Helvetica', 'B', 24)
    pdf.set_y(5*ppi)
    pdf.cell(w=0, text=name, align=Align.C)

    # Open and return a reader to the pdf we just made
    return PdfReader(io.BytesIO(pdf.output()))
```

This code should be feeling pretty familiar by now; there's nothing here we haven't already seen. Here's an example of how this will appear once rendered:



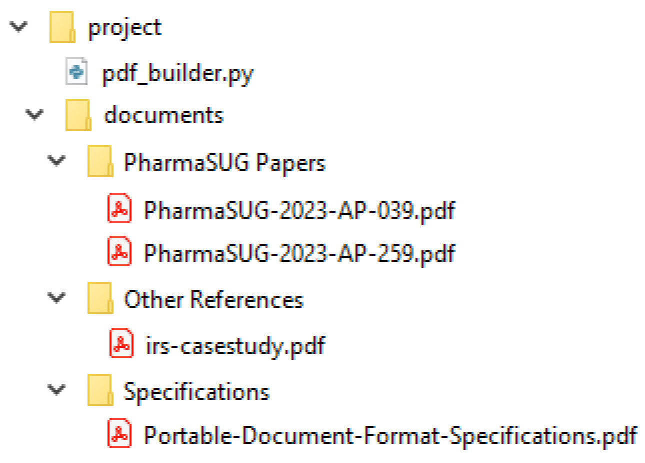
Part 2

PharmaSUG Papers

Figure 5: Example section separator text.

Now we need to actually reorganize our source documents. So far, we've gotten by with a simple `glob()` of the whole directory, but this new feature demands a bit more structure. We need to decide how to specify the section names and which documents go under each of them.

There are many possible methods we could pursue, but for the purposes of this demonstration, we'll use a very simple one: sort our documents into subdirectories! Just name a folder after each section, then drag and drop the PDFs:



```
project
  pdf_builder.py
  documents
    PharmaSUG Papers
      PharmaSUG-2023-AP-039.pdf
      PharmaSUG-2023-AP-259.pdf
    Other References
      irs-casestudy.pdf
    Specifications
      Portable-Document-Format-Specifications.pdf
```

Project root structure with sections.

Let's update `build_pdf()` to handle the new file structure and integrate the separator pages:

```
def build_pdf():

    output = PdfWriter()

    # Start with title page
    output.append(make_title())
    no_pagenum = [0]

    ### NEW CODE
    # Iterate by section
    iter = filter(lambda x: x.is_dir(), Path('documents').iterdir())
    for i, section_path in enumerate(sorted(iter)):

        # Create section page (and add to page number ignore list)
        no_pagenum.append(len(output.pages))
        section_bookmark = output.add_outline_item(
            f'Part {i+1} - {section_path.stem}',
            len(output.pages)
        )
        output.append(make_section_page(i+1, section_path.stem))

        # Iterate documents within the section
        for pdf in sorted(section_path.glob('*.pdf')):
            print(f'Adding {pdf.stem}')
            output.add_outline_item(
                pdf.stem,
                len(output.pages),
                parent=section_bookmark
            )
            output.append_pages_from_reader(PdfReader(pdf))
    ### END NEW CODE

    # Write page numbers onto each output page
    print('Writing page numbers.')
    for i, page in enumerate(output.pages):
        if i in no_pagenum: continue
        text = f'Page {i+1} of {len(output.pages)}'
        page.merge_page(page2=make_page_number(page, text))

    return output
```

It looks like a lot of new code, but we're really only doing two things: wrapping the core document-concatenation loop inside a new outer loop, and introducing a handful of steps before each set of documents to handle the start of each section.

The first two new lines establish that we will only proceed with each subfolder under the documents directory, ignoring any actual files. The `iterdir()` function provides all the child items, which we then wrap in a call to `filter()`. This applies a test to each item and discards the ones that fail. In this case, we want it to *filter out* any child items that aren't directories:

```
iter = filter(lambda x: x.is_dir(), Path('documents').iterdir())
for i, section_path in enumerate(sorted(iter)):
```

Before we can append the documents in the current section, we need to create both a separator page and a bookmark for the section itself. The first step is to prevent the section separator page from having a page number added:

```
no_pagenum.append(len(output.pages))
```

Setting this up as a list earlier is coming in handy! We won't have to change the page-numbering code at all.

Next, we create a bookmark for the section. This is the same as before, but we now capture the return value of `add_outline_item()` in a variable. We'll need that value later to nest the document bookmarks underneath:

```
section_bookmark = output.add_outline_item(
    f'Part {i+1} - {section_path.stem}',
    len(output.pages)
)
```

Finally, we add the actual separator page to the document:

```
output.append(make_section_page(i+1, section_path.stem))
```

Now we can proceed with appending each document almost exactly as we did at the very start, but with two important differences. The first is, of course, that we should limit processing to the current section's subdirectory:

```
for pdf in sorted(section_path.glob('*.pdf')):
```

The second is that when we create the bookmark, we specify to create it under the section's overall bookmark with the `parent=` option:

```
output.add_outline_item(
    pdf.stem,
    len(output.pages),
    parent=section_bookmark
)
```

Here's what the resulting bookmarks look like. Nice and orderly! Clicking on a top-level bookmark will take you to that section's separator page:

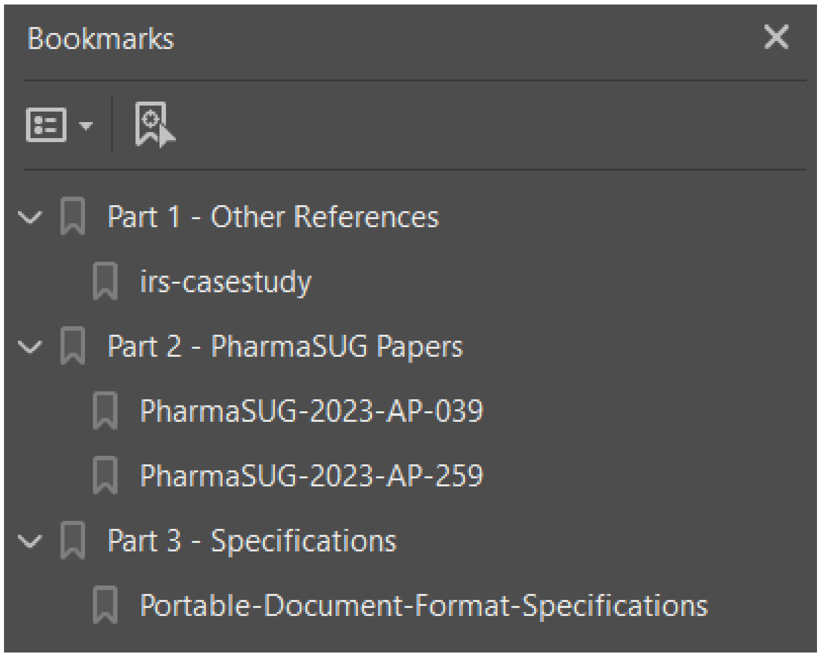

Bookmarks
Part 1 - Other References
irs-casestudy
Part 2 - PharmaSUG Papers
PharmaSUG-2023-AP-039
PharmaSUG-2023-AP-259
Part 3 - Specifications
Portable-Document-Format-Specifications

Figure 6: Document bookmarks with sections.

Nothing in the above limited our hierarchy to just two levels. We can nest bookmarks arbitrarily deeply as long as we keep track of the `parents`! Whatever is most appropriate for your specific document's navigation needs.

That wraps up our walkthrough. The final, complete program listing can be found in the appendix for convenience.

CUSTOMIZATION AND NEXT STEPS

Our intent with the preceding walkthrough is to give a starting point for you to develop your own generator. To that end, we'd like to discuss a few of the limitations of our program and offer some ideas for addressing them.

DOCUMENT ORDER

You may have noticed we were unable to control the order of the documents and sections other than alphabetically. This was chosen for simplicity and consistency, but there are many other options.

One common approach is to write a metadata file that explicitly lists every document and its position in the final report (see Plevney [1] for an example of this method). Python has APIs for many types of file formats used for this purpose, including JSON [10], XML [11], and XLSX [8] (although [9] is easier to work with directly). You can then replace the simple scheme we described with some custom logic to ingest your metadata.

DOCUMENT TITLES

We used each document's filename as its bookmark title, but these may not always be aligned. Again, it's often metadata to the rescue, and you can certainly use the same approach as for the document order.

However, we'd like to call attention to another possibility. Depending on the structure of the underlying document, it may be possible to automatically detect the intended title. Each page of a `PdfReader` object has an `extract_text()` function that can be used to examine its contents. By using this to look inside your documents, you may find that you can pull the title from the contents directly, such as if the title is always the first line or is always preceded by some unique marker. If so, you may be able to save a lot of time by avoiding the need to create new metadata manually for each report.

CONCLUSION

We've shown that proprietary software isn't the only option for manipulating PDF documents. In fact, it can be very straightforward to assemble reports that contain both pre-existing and custom-generated content using just free, open-source libraries. These tools are just as reliable as the proprietary options due to the fact that the PDF is an ISO standard.

We hope the information we've presented here empowers you to build your own automated document pipeline, specialized for your unique requirements. Happy coding!

REFERENCES

[1] Plevney, T. "No More Manual PDF Bookmarks! An Automated Approach to Converting RTF files to a Consolidated PDF with Bookmarks." PharmaSUG 2023. https://www.pharmasug.org/proceedings/2023/AP/PharmaSUG-2023-AP-259.pdf

[2] Coar, W. "Combining TLFs into a Single File Deliverable." PharmaSUG 2016. https://www.pharmasug.org/proceedings/2016/HT/PharmaSUG-2016-HT06.pdf

[3] Shao, W. and Zhang, L. "A utility to combine study level outputs to one PDF file for development safety update report (DSUR) regional appendices." PharmaSUG 2023. https://www.pharmasug.org/proceedings/2023/AP/PharmaSUG-2023-AP-039.pdf

[4] Wang, J. and Zhou, Y. "A Tool to Combine TFL Outputs." PharmaSUG China 2021. https://www.pharmasug.org/proceedings/china2021/AD/Pharmasug-China-2021-AD013.pdf

[5] Xie, L. "A Simple SAS Utility to Combine Existing RTF Tables/Figures and Create a MultiLevel Bookmark Hierarchy and a Hyperlinked TOC." PharmaSUG 2019. https://www.pharmasug.org/proceedings/2019/AD/PharmaSUG-2019-AD-104.pdf

[6] pypdf https://github.com/py-pdf/pypdf

[7] fpdf2 https://github.com/py-pdf/fpdf2

[8] openpyxl https://pypi.org/project/openpyxl/

[9] pandas https://pandas.pydata.org/

[10] "JSON encoder and decoder" https://docs.python.org/3/library/json.html

[11] "The ElementTree XML API" https://docs.python.org/3/library/xml.etree.elementtree.html

[12] Knowledge at Wharton. "Adobe Co-founder John Warnock on the Competitive Advantages of Aesthetics and the 'Right' Technology." 20 January 2010. https://knowledge.wharton.upenn.edu/article/adobe-co-founder-john-warnock-on-the-competitive-advantages-of-aesthetics-and-the-right-technology/

[13] Knowledge at Wharton. "Driving Adobe: Co-founder Charles Geschke on Challenges, Change and Values." 3 September 2008. https://knowledge.wharton.upenn.edu/article/driving-adobe-co-founder-charles-geschke-on-challenges-change-and-values/

[14] Adobe. "The History of the PDF: Timeline." Accessed 22 March 2024. https://www.adobe.com/acrobat/resources/pdf-timeline.html

[15] Adobe Corporate Communications. "Who Created the PDF?" 18 June 2015. https://blog.adobe.com/en/publish/2015/06/18/who-created-pdf

[16] "IRS deploys Adobe Acrobat to more than 100,000 staff members nationwide; clear, reliable tax documents in Adobe PDF benefit agency and public." https://images.tedium.co/uploads/irs-casestudy.pdf

[17] Archive.org. “Inside the Comdex [spring] expo trade show in April 1992.” 8 September 2022. https://archive.org/details/youtube-TCSSEkH_A9c

[18] The Age. “Penny-pinching way to the bank.” 24 November 1992. Available at https://www.newspapers.com/article/the-age/17856365/

[19] Adobe. “Portable Document Format Reference Manual.” 1993. Available at https://web.archive.org/web/20150617123515/http://acroeng.adobe.com/PDFReference/PDF%20Reference%201.0.pdf

[20] Johnson, Duff. “No, PDF is NOT Owned by Adobe!” 26 April 2010. https://talkingpdf.org/no-pdf-is-not-owned-by-adobe/

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:
James Austrow
C5 Research, Cleveland Clinic
austroj AT ccf DOT org

APPENDIX: COMPLETE PROGRAM LISTING

```
import io
from fpdf import FPDF, Align
from pypdf import PdfReader, PdfWriter
from pathlib import Path

def build_pdf():

    output = PdfWriter()

    # Start with title page
    output.append(make_title_page())
    no_pagenum = [0]

    # Iterate by section
    section_iter = filter(lambda x: x.is_dir(), Path('documents').iterdir())
    for i, section_path in enumerate(sorted(section_iter)):

        # Create section separator page (and add to page number ignore list)
        no_pagenum.append(len(output.pages))
        section_bookmark = output.add_outline_item(
            f'Part {i+1} - {section_path.stem}',
            len(output.pages)
        )
        output.append(make_section_page(i+1, section_path.stem))

        # Iterate documents within the section
        for pdf in sorted(section_path.glob('*.pdf')):
            print(f'Adding {pdf.stem}')
            output.add_outline_item(
                pdf.stem,
                len(output.pages),
                parent=section_bookmark
            )
            output.append_pages_from_reader(PdfReader(pdf))

    # Write page numbers onto each output page
    print('Writing page numbers.')
    for i, page in enumerate(output.pages):
        if i in no_pagenum: continue # skip specified pages
        text = f'Page {i+1} of {len(output.pages)}'
        page.merge_page(page2=make_page_number(page, text))

    return output

def make_page_number(source_page, text):
    # Obtain dimensions from the original page
    size = (source_page.mediabox.width, source_page.mediabox.height)
    ppi = 72 # convenience variable for points per inch

    # Initialize a new PDF
    pdf = FPDF(unit='pt', format=size)
    pdf.set_font('Courier', '', 11)
    pdf.set_auto_page_break(False)

    # Write the text
    pdf.add_page()
    pdf.set_y(size[1] - ppi//2) # half an inch from the bottom
    pdf.cell(w=0, text=text, align=Align.C)

    # Open a reader to the pdf we just made and return the first (only) page
    return PdfReader(io.BytesIO(pdf.output())).pages[0]

def make_title_page():

    # Set page units and dimensions
    ppi = 72
    dim = (8.5*ppi, 11*ppi) # 8.5" by 11"

    pdf = FPDF(unit='pt', format=dim)
    pdf.set_margins(1*ppi, 1*ppi, 1*ppi) # 1" for left, top, and right

    # Draw all shapes and text white
    pdf.set_text_color(255)
    pdf.set_draw_color(255)
    pdf.set_fill_color(255)

    # Begin content
    pdf.add_page()
    pdf.image('../images/title-bg.png', x=0, y=0, w=dim[0], h=dim[1])
    pdf.image('../images/PharmaSUG_2024_logo_200px_trans.png', x=400, y=20)

    pdf.set_font('Helvetica', 'B', 48)
    pdf.set_y(6.5*ppi)
    pdf.cell(w=0, text='PDF Demo', align=Align.L)

    pdf.set_font('Helvetica', 'B', 18)
    pdf.set_y(7.25*ppi)
    pdf.cell(w=0, text='Selected documents packet', align=Align.L)

    pdf.set_font('Helvetica', 'B', 14)
    pdf.set_y(10*ppi)
    pdf.cell(w=0, text='[put author here]', align=Align.L)
    pdf.cell(w=0, text='PharmaSUG 2024', align=Align.R)

    ypos = 7.75*ppi
    pdf.set_line_width(4)
    pdf.line(1*ppi, ypos, 7.5*ppi, ypos)

    # Open and return a reader to the pdf we just made
    return PdfReader(io.BytesIO(pdf.output()))

def make_section_page(k, name):

    ppi = 72
    dim = (8.5*ppi, 11*ppi)

    pdf = FPDF(unit='pt', format=dim)
    pdf.set_margins(1*ppi, 1*ppi, 1*ppi)
    pdf.add_page()

    pdf.set_font('Helvetica', 'B', 18)
    pdf.set_y(4.5*ppi)
    pdf.cell(w = 0, text=f'Part {k}', align=Align.C)

    pdf.set_font('Helvetica', 'B', 24)
    pdf.set_y(5*ppi)
    pdf.cell(w=0, text=name, align=Align.C)

    # Open and return a reader to the pdf we just made
    return PdfReader(io.BytesIO(pdf.output()))

if __name__ == '__main__':
    output = build_pdf()
    print('Outputting final document.')
    output.write('result.pdf')
    output.close()
```