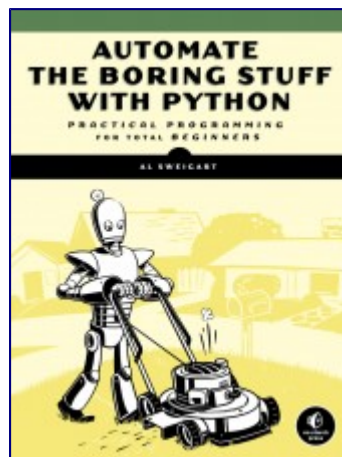


CHAPTER 13 – WORKING WITH PDF AND WORD DOCUMENTS



Support the Author: Buy the book on [Amazon](#) or [the book/ebook bundle directly from No Starch Press](#).



[Read the author's other free Python books:](#)



WORKING WITH PDF AND WORD DOCUMENTS

PDF and Word documents are binary files, which makes them much more complex than plaintext files. In addition to text, they store lots of font, color, and layout information. If you want your programs to read or write to PDFs or Word documents, you'll need to do more than simply pass their filenames to `open()`.

Fortunately, there are Python modules that make it easy for you to interact with PDFs and Word documents. This chapter will cover two such modules: `PyPDF2` and `Python-Docx`.

PDF DOCUMENTS

PDF stands for Portable Document Format and uses the `.pdf` file extension. Although PDFs support many features, this chapter will focus on the two things you'll be doing most often with them: reading text content from PDFs and crafting new PDFs from existing documents.

The module you'll use to work with PDFs is `PyPDF2`. To install it, run `pip install PyPDF2` from the command line. This module name is case sensitive, so make sure the `y` is lowercase and everything else is uppercase. (Check out Appendix A for full details about installing third-party modules.) If the module was installed correctly, running `import PyPDF2` in the interactive shell shouldn't display any errors.

The Problematic PDF Format

While PDF files are great for laying out text in a way that's easy for people to print and read, they're not straightforward for software to parse into plaintext. As such, `PyPDF2` might make mistakes when extracting text from a PDF and may even be unable to open some PDFs at all. There isn't much you can do about this, unfortunately. `PyPDF2` may simply be unable to work with some of your particular PDF files. That said, I haven't found any PDF files so far that can't be opened with `PyPDF2`.

EXTRACTING TEXT FROM PDFS

`PyPDF2` does not have a way to extract images, charts, or other media from PDF documents, but it can extract text and return it as a Python

string. To start learning how PyPDF2 works, we'll use it on the example PDF shown in [Figure 13-1](#).

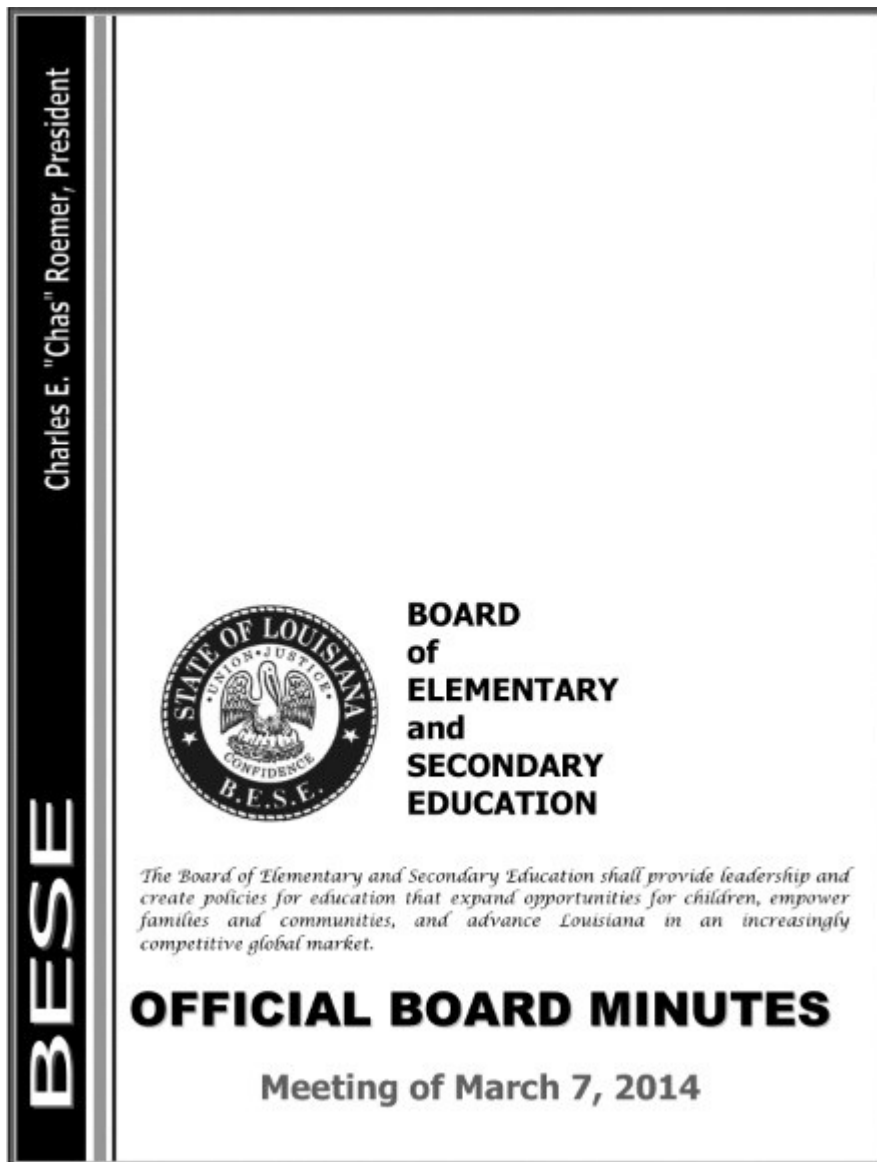


Figure 13-1. The PDF page that we will be extracting text from

Download this PDF from <http://nostarch.com/automatestuff/>, and enter the following into the interactive shell:

```
>>> import PyPDF2

>>> pdfFileObj = open('meetingminutes.pdf', 'rb')

>>> pdfReader = PyPDF2.PdfFileReader(pdfFileObj)

❶ >>> pdfReader.numPages
```

❷ >>> pageObj = pdfReader.getPage(0)

❸ >>> pageObj.extractText()

'OOFFFFIICCIIAALL BBOOAARRDD MMIINNUUTTEESS Meeting of
March 7, 2015

\n The Board of Elementary and Secondary Education shall provide
leadership

and create policies for education that expand opportunities for
children,

empower families and communities, and advance Louisiana in an
increasingly

competitive global market. BOARD of ELEMENTARY and
SECONDARY EDUCATION '

First, import the PyPDF2 module. Then open meetingminutes.pdf in read
binary mode and store it in pdfFileObj. To get a PdfFileReader object that
represents this PDF, call PyPDF2.PdfFileReader() and pass it pdfFileObj.
Store this PdfFileReader object in pdfReader.

The total number of pages in the document is stored in
the numPages attribute of a PdfFileReader object ❶. The example PDF
has 19 pages, but let's extract text from only the first page.

To extract text from a page, you need to get a Page object, which
represents a single page of a PDF, from a PdfFileReader object. You can
get a Page object by calling the getPage() method ❷ on
a PdfFileReader object and passing it the page number of the page you're
interested in—in our case, 0.

PyPDF2 uses a zero-based index for getting pages: The first page is page 0,
the second is [Introduction](#), and so on. This is always the case, even if pages
are numbered differently within the document. For example, say your PDF
is a three-page excerpt from a longer report, and its pages are numbered
42, 43, and 44. To get the first page of this document, you would want to
call pdfReader.getPage(0), not getPage(42) or getPage(1).

Once you have your Page object, call its extractText() method to return a
string of the page's text ❸. The text extraction isn't perfect: The

text Charles E. “Chas” Roemer, President from the PDF is absent from the string returned by `extractText()`, and the spacing is sometimes off. Still, this approximation of the PDF text content may be good enough for your program.

DECRYPTING PDFs

Some PDF documents have an encryption feature that will keep them from being read until whoever is opening the document provides a password. Enter the following into the interactive shell with the PDF you downloaded, which has been encrypted with the password `rosebud`:

```
>>> import PyPDF2

>>> pdfReader = PyPDF2.PdfFileReader(open('encrypted.pdf', 'rb'))

❶ >>> pdfReader.isEncrypted

True

>>> pdfReader.getPage(0)
```

❷ Traceback (most recent call last):

```
File "<pyshell#173>", line 1, in <module>
    pdfReader.getPage()
--snip--
File "C:\Python34\lib\site-packages\PyPDF2\pdf.py", line 1173, in
getObject
    raise utils.PdfReadError("file has not been decrypted")
PyPDF2.utils.PdfReadError: file has not been decrypted
```

```
❸ >>> pdfReader.decrypt('rosebud')

1

>>> pageObj = pdfReader.getPage(0)
```

All `PdfFileReader` objects have an `isEncrypted` attribute that is `True` if the PDF is encrypted and `False` if it isn't ❶. Any attempt to call a function that reads the file before it has been decrypted with the correct password will result in an error ❷.

To read an encrypted PDF, call the `decrypt()` function and pass the password as a string **3**. After you call `decrypt()` with the correct password, you'll see that calling `getPage()` no longer causes an error. If given the wrong password, the `decrypt()` function will return 0 and `getPage()` will continue to fail. Note that the `decrypt()` method decrypts only the `PdfFileReader` object, not the actual PDF file. After your program terminates, the file on your hard drive remains encrypted. Your program will have to call `decrypt()` again the next time it is run.

CREATING PDFS

PyPDF2's counterpart to `PdfFileReader` objects is `PdfFileWriter` objects, which can create new PDF files. But PyPDF2 cannot write arbitrary text to a PDF like Python can do with plaintext files. Instead, PyPDF2's PDF-writing capabilities are limited to copying pages from other PDFs, rotating pages, overlaying pages, and encrypting files.

PyPDF2 doesn't allow you to directly edit a PDF. Instead, you have to create a new PDF and then copy content over from an existing document. The examples in this section will follow this general approach:

1. Open one or more existing PDFs (the source PDFs) into `PdfFileReader` objects.
2. Create a new `PdfFileWriter` object.
3. Copy pages from the `PdfFileReader` objects into the `PdfFileWriter` object.
4. Finally, use the `PdfFileWriter` object to write the output PDF.

Creating a `PdfFileWriter` object creates only a value that represents a PDF document in Python. It doesn't create the actual PDF file. For that, you must call the `PdfFileWriter`'s `write()` method.

The `write()` method takes a regular `File` object that has been opened in write-binary mode. You can get such a `File` object by calling Python's `open()` function with two arguments: the string of what you want the PDF's filename to be and 'wb' to indicate the file should be opened in write-binary mode.

If this sounds a little confusing, don't worry—you'll see how this works in the following code examples.

COPYING PAGES

You can use PyPDF2 to copy pages from one PDF document to another. This allows you to combine multiple PDF files, cut unwanted pages, or reorder pages.

Download `meetingminutes.pdf` and `meetingminutes2.pdf` from <http://nostarch.com/automatestuff/> and place the PDFs in the current working directory. Enter the following into the interactive shell:

```
>>> import PyPDF2

>>> pdf1File = open('meetingminutes.pdf', 'rb')

>>> pdf2File = open('meetingminutes2.pdf', 'rb')

❶ >>> pdf1Reader = PyPDF2.PdfFileReader(pdf1File)

❷ >>> pdf2Reader = PyPDF2.PdfFileReader(pdf2File)

❸ >>> pdfWriter = PyPDF2.PdfFileWriter()

>>> for pageNum in range(pdf1Reader.numPages):

❹     pageObj = pdf1Reader.getPage(pageNum)

❺     pdfWriter.addPage(pageObj)

>>> for pageNum in range(pdf2Reader.numPages):

❻     pageObj = pdf2Reader.getPage(pageNum)

❼     pdfWriter.addPage(pageObj)

❽ >>> pdfOutputFile = open('combinedminutes.pdf', 'wb')

>>> pdfWriter.write(pdfOutputFile)

>>> pdfOutputFile.close()
```

```
>>> pdf1File.close()
```

```
>>> pdf2File.close()
```

Open both PDF files in read binary mode and store the two resulting File objects in pdf1File and pdf2File.

Call PyPDF2.PdfFileReader() and pass it pdf1File to get a PdfFileReader object for meetingminutes.pdf ❶. Call it again and pass it pdf2File to get a PdfFileReader object for meetingminutes2.pdf ❷. Then create a new PdfFileWriter object, which represents a blank PDF document ❸.

Next, copy all the pages from the two source PDFs and add them to the PdfFileWriter object. Get the Page object by calling getPage() on a PdfFileReader object ❹. Then pass that Page object to your PdfFileWriter's addPage() method ❺. These steps are done first for pdf1Reader and then again for pdf2Reader. When you're done copying pages, write a new PDF called combinedminutes.pdf by passing a File object to the PdfFileWriter's write() method ❻.

NOTE

PyPDF2 cannot insert pages in the middle of a PdfFileWriter object; the addPage() method will only add pages to the end.

You have now created a new PDF file that combines the pages from meetingminutes.pdf and meetingminutes2.pdf into a single document. Remember that the File object passed to PyPDF2.PdfFileReader() needs to be opened in read-binary mode by passing 'rb' as the second argument to open(). Likewise, the File object passed to PyPDF2.PdfFileWriter() needs to be opened in write-binary mode with 'wb'.

ROTATING PAGES

The pages of a PDF can also be rotated in 90-degree increments with the rotateClockwise() and rotateCounterClockwise() methods. Pass one of the integers 90, 180, or 270 to these methods. Enter the following into the interactive shell, with the meetingminutes.pdf file in the current working directory:


```

>>> import PyPDF2

>>> minutesFile = open('meetingminutes.pdf', 'rb')

>>> pdfReader = PyPDF2.PdfFileReader(minutesFile)

❶ >>> page = pdfReader.getPage(0)

❷ >>> page.rotateClockwise(90)

{'/Contents': [IndirectObject(961, 0), IndirectObject(962, 0),
--snip--
}]

>>> pdfWriter = PyPDF2.PdfFileWriter()

>>> pdfWriter.addPage(page)

❸ >>> resultPdfFile = open('rotatedPage.pdf', 'wb')

>>> pdfWriter.write(resultPdfFile)

>>> resultPdfFile.close()

>>> minutesFile.close()

```

Here we use `getPage(0)` to select the first page of the PDF ❶, and then we call `rotateClockwise(90)` on that page ❷. We write a new PDF with the rotated page and save it as `rotatedPage.pdf` ❸.

The resulting PDF will have one page, rotated 90 degrees clockwise, as in [Figure 13-2](#). The return values from `rotateClockwise()` and `rotateCounterClockwise()` contain a lot of information that you can ignore.

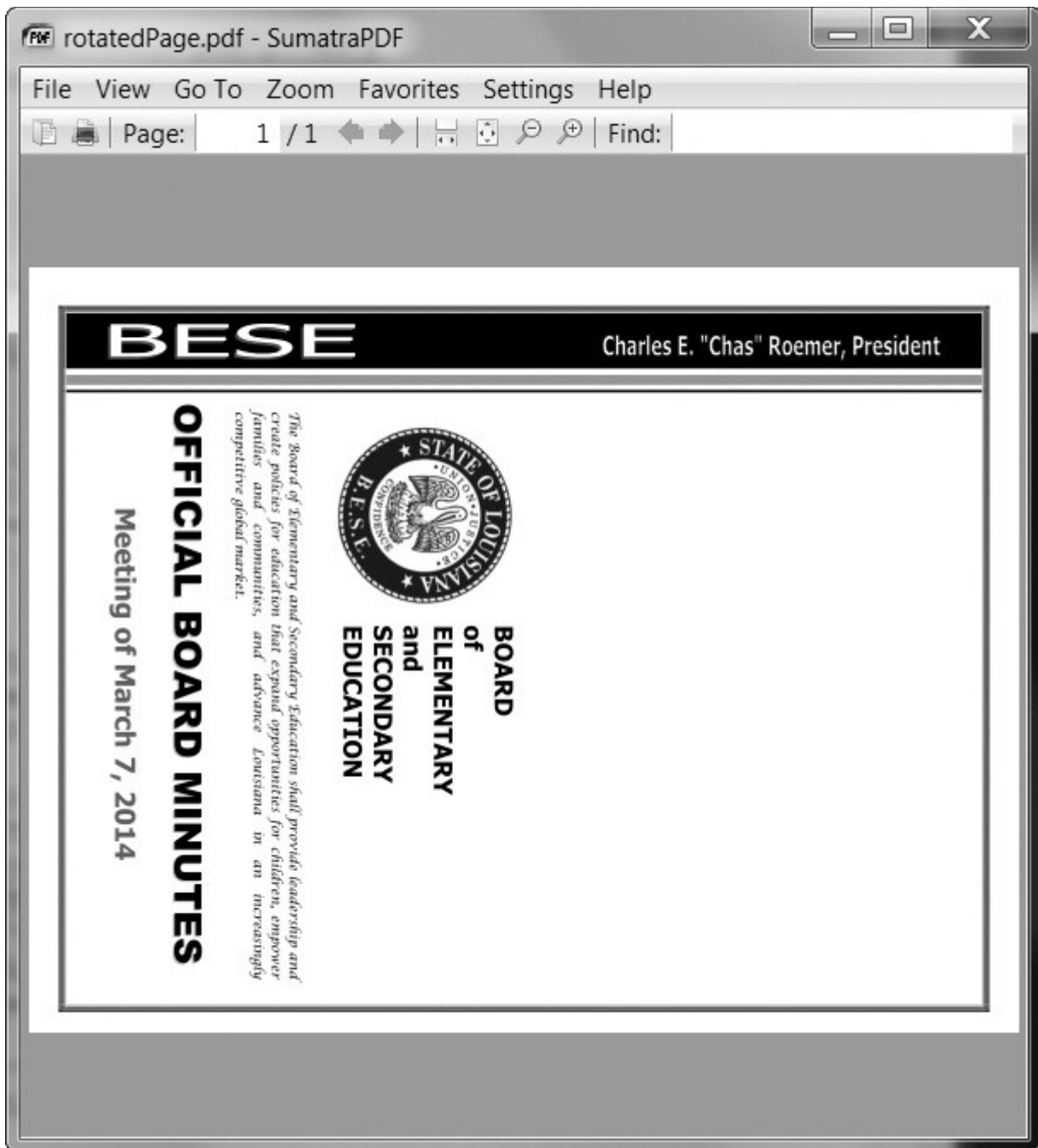


Figure 13-2. The rotatedPage.pdf file with the page rotated 90 degrees clockwise

OVERLAYING PAGES

PyPDF2 can also overlay the contents of one page over another, which is useful for adding a logo, timestamp, or watermark to a page. With Python, it's easy to add watermarks to multiple files and only to pages your program specifies.

Download watermark.pdf from <http://nostarch.com/automatestuff/> and place the PDF in the current working directory along

with meetingminutes.pdf. Then enter the following into the interactive shell:

```
>>> import PyPDF2

>>> minutesFile = open('meetingminutes.pdf', 'rb')

❷ >>> pdfReader = PyPDF2.PdfFileReader(minutesFile)

❷ >>> minutesFirstPage = pdfReader.getPage(0)

❸ >>> pdfWatermarkReader =
PyPDF2.PdfFileReader(open('watermark.pdf', 'rb'))

❹ >>> minutesFirstPage.mergePage(pdfWatermarkReader.getPage(0))

❺ >>> pdfWriter = PyPDF2.PdfFileWriter()

❻ >>> pdfWriter.addPage(minutesFirstPage)

❼ >>> for pageNum in range(1, pdfReader.numPages):
    pageObj = pdfReader.getPage(pageNum)
    pdfWriter.addPage(pageObj)

>>> resultPdfFile = open('watermarkedCover.pdf', 'wb')

>>> pdfWriter.write(resultPdfFile)

>>> minutesFile.close()

>>> resultPdfFile.close()
```

Here we make a PdfFileReader object of meetingminutes.pdf ❶. We call getPage(0) to get a Page object for the first page and store this object in minutesFirstPage ❷. We then make a PdfFileReader object for watermark.pdf ❸ and call mergePage() on minutesFirstPage ❹. The argument we pass to mergePage() is a Page object for the first page of watermark.pdf.

Now that we've called mergePage() on minutesFirstPage, minutesFirstPage represents the watermarked first page. We make a PdfFileWriter object ❺ and add the watermarked first page ❻. Then we loop through the rest of the pages

in `meetingminutes.pdf` and add them to the `PdfFileWriter` object ⑦. Finally, we open a new PDF called `watermarkedCover.pdf` and write the contents of the `PdfFileWriter` to the new PDF.

[Figure 13-3](#) shows the results. Our new PDF, `watermarkedCover.pdf`, has all the contents of the `meetingminutes.pdf`, and the first page is watermarked.



Figure 13-3. The original PDF (left), the watermark PDF (center), and the merged PDF (right)

ENCRYPTING PDFs

A `PdfFileWriter` object can also add encryption to a PDF document. Enter the following into the interactive shell:

```
>>> import PyPDF2
>>> pdfFile = open('meetingminutes.pdf', 'rb')
>>> pdfReader = PyPDF2.PdfFileReader(pdfFile)
>>> pdfWriter = PyPDF2.PdfFileWriter()
>>> for pageNum in range(pdfReader.numPages):
>>>     pdfWriter.addPage(pdfReader.getPage(pageNum))
① >>> pdfWriter.encrypt('swordfish')
>>> resultPdf = open('encryptedminutes.pdf', 'wb')
```

```
>>> pdfWriter.write(resultPdf)
```

```
>>> resultPdf.close()
```

Before calling the `write()` method to save to a file, call the `encrypt()` method and pass it a password string ❶. PDFs can have a user password (allowing you to view the PDF) and an owner password (allowing you to set permissions for printing, commenting, extracting text, and other features). The user password and owner password are the first and second arguments to `encrypt()`, respectively. If only one string argument is passed to `encrypt()`, it will be used for both passwords.

In this example, we copied the pages of `meetingminutes.pdf` to a `PdfFileWriter` object. We encrypted the `PdfFileWriter` with the password `swordfish`, opened a new PDF called `encryptedminutes.pdf`, and wrote the contents of the `PdfFileWriter` to the new PDF. Before anyone can view `encryptedminutes.pdf`, they'll have to enter this password. You may want to delete the original, unencrypted `meetingminutes.pdf` file after ensuring its copy was correctly encrypted.

PROJECT: COMBINING SELECT PAGES FROM MANY PDFS

Say you have the boring job of merging several dozen PDF documents into a single PDF file. Each of them has a cover sheet as the first page, but you don't want the cover sheet repeated in the final result. Even though there are lots of free programs for combining PDFs, many of them simply merge entire files together. Let's write a Python program to customize which pages you want in the combined PDF.

At a high level, here's what the program will do:

- Find all PDF files in the current working directory.
- Sort the filenames so the PDFs are added in order.
- Write each page, excluding the first page, of each PDF to the output file.

In terms of implementation, your code will need to do the following:

- Call `os.listdir()` to find all the files in the working directory and remove any non-PDF files.
- Call Python's `sort()` list method to alphabetize the filenames.
- Create a `PdfFileWriter` object for the output PDF.
- Loop over each PDF file, creating a `PdfFileReader` object for it.
- Loop over each page (except the first) in each PDF file.
- Add the pages to the output PDF.
- Write the output PDF to a file named `allminutes.pdf`.

For this project, open a new file editor window and save it as `combinePdfs.py`.

STEP 1: FIND ALL PDF FILES

First, your program needs to get a list of all files with the `.pdf` extension in the current working directory and sort them. Make your code look like the following:

```
#!/ python3

# combinePdfs.py - Combines all the PDFs in the current working
directory into

# into a single PDF.
```

❶ `import PyPDF2, os`

```
# Get all the PDF filenames.
```

```
pdfFiles = []
```

```
for filename in os.listdir('.'):

```

```
    if filename.endswith('.pdf'):

```

❷ `pdfFiles.append(filename)`

❸ `pdfFiles.sort(key=str.lower)`

④ pdfWriter = PyPDF2.PdfFileWriter()

TODO: Loop through all the PDF files.

TODO: Loop through all the pages (except the first) and add them.

TODO: Save the resulting PDF to a file.

After the shebang line and the descriptive comment about what the program does, this code imports the os and PyPDF2 modules ①. The os.listdir('.') call will return a list of every file in the current working directory. The code loops over this list and adds only those files with the .pdf extension to pdfFiles ②. Afterward, this list is sorted in alphabetical order with the key=str.lower keyword argument to sort() ③.

A PdfFileWriter object is created to hold the combined PDF pages ④. Finally, a few comments outline the rest of the program.

STEP 2: OPEN EACH PDF

Now the program must read each PDF file in pdfFiles. Add the following to your program:

```
#!/python3
```

```
# combinePdfs.py - Combines all the PDFs in the current working  
directory into
```

```
# a single PDF.
```

```
import PyPDF2, os
```

```
# Get all the PDF filenames.
```

```
pdfFiles = []
```

```
--snip--
```

```
# Loop through all the PDF files.
```

```
for filename in pdfFiles:
```

```
    pdfFileObj = open(filename, 'rb')
```

```
    pdfReader = PyPDF2.PdfFileReader(pdfFileObj)
```

```
    # TODO: Loop through all the pages (except the first) and add them.
```

```
# TODO: Save the resulting PDF to a file.
```

For each PDF, the loop opens a filename in read-binary mode by calling `open()` with `'rb'` as the second argument. The `open()` call returns a File object, which gets passed to `PyPDF2.PdfFileReader()` to create a `PdfFileReader` object for that PDF file.

STEP 3: ADD EACH PAGE

For each PDF, you'll want to loop over every page except the first. Add this code to your program:

```
#!/python3
```

```
# combinePdfs.py - Combines all the PDFs in the current working directory into
```

```
# a single PDF.
```

```
import PyPDF2, os
```

```
--snip--
```

```
# Loop through all the PDF files.
```

```
for filename in pdfFiles:
```

```
--snip--
```

```
    # Loop through all the pages (except the first) and add them.
```

```
    ❶ for pageNum in range(1, pdfReader.numPages):
```



```
pageObj = pdfReader.getPage(pageNum)
pdfWriter.addPage(pageObj)
```

TODO: Save the resulting PDF to a file.

The code inside the for loop copies each Page object individually to the PdfFileWriter object. Remember, you want to skip the first page. Since PyPDF2 considers 0 to be the first page, your loop should start at 1 **1** and then go up to, but not include, the integer in pdfReader.numPages.

STEP 4: SAVE THE RESULTS

After these nested for loops are done looping, the pdfWriter variable will contain a PdfFileWriter object with the pages for all the PDFs combined. The last step is to write this content to a file on the hard drive. Add this code to your program:

```
#!/python3
# combinePdfs.py - Combines all the PDFs in the current working
directory into
# a single PDF.
import PyPDF2, os

--snip--

# Loop through all the PDF files.
for filename in pdfFiles:
    --snip--
    # Loop through all the pages (except the first) and add them.
    for pageNum in range(1, pdfReader.numPages):
        --snip--

# Save the resulting PDF to a file.
```

```
pdfOutput = open('allminutes.pdf', 'wb')  
pdfWriter.write(pdfOutput)  
pdfOutput.close()
```

Passing 'wb' to open() opens the output PDF file, allminutes.pdf, in write-binary mode. Then, passing the resulting File object to the write() method creates the actual PDF file. A call to the close() method finishes the program.

IDEAS FOR SIMILAR PROGRAMS

Being able to create PDFs from the pages of other PDFs will let you make programs that can do the following:

- Cut out specific pages from PDFs.
- Reorder pages in a PDF.
- Create a PDF from only those pages that have some specific text, identified by extractText().

WORD DOCUMENTS

Python can create and modify Word documents, which have the .docx file extension, with the python-docx module. You can install the module by running pip install python-docx. (Appendix A has full details on installing third-party modules.)

NOTE

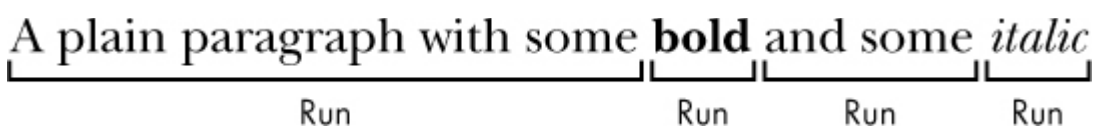
When using pip to first install Python-Docx, be sure to install python-docx, not docx. The installation name docx is for a different module that this book does not cover. However, when you are going to import the python-docx module, you'll need to run import docx, not import python-docx.

If you don't have Word, LibreOffice Writer and OpenOffice Writer are both free alternative applications for Windows, OS X, and Linux that can be used to open .docx files. You can download them from <https://www.libreoffice.org> and <http://openoffice.org>, respectively. The full documentation for Python-Docx is available at <https://python->

docx.readthedocs.org/. Although there is a version of Word for OS X, this chapter will focus on Word for Windows.

Compared to plaintext, .docx files have a lot of structure. This structure is represented by three different data types in Python-Docx. At the highest level, a Document object represents the entire document.

The Document object contains a list of Paragraph objects for the paragraphs in the document. (A new paragraph begins whenever the user presses ENTER or RETURN while typing in a Word document.) Each of these Paragraph objects contains a list of one or more Run objects. The single-sentence paragraph in [Figure 13-4](#) has four runs.



A plain paragraph with some **bold** and some *italic*

The diagram shows the sentence "A plain paragraph with some **bold** and some *italic*" with four brackets underneath identifying runs: "A plain paragraph with some", "**bold**", "and some", and "*italic*". Each bracket is labeled "Run" below it.

Figure 13-4. The Run objects identified in a Paragraph object

The text in a Word document is more than just a string. It has font, size, color, and other styling information associated with it. A style in Word is a collection of these attributes. A Run object is a contiguous run of text with the same style. A new Run object is needed whenever the text style changes.

READING WORD DOCUMENTS

Let's experiment with the python-docx module.

Download demo.docx from <http://nostarch.com/automatestuff/> and save the document to the working directory. Then enter the following into the interactive shell:

```
>>> import docx

❶ >>> doc = docx.Document('demo.docx')

❷ >>> len(doc.paragraphs)

7

❸ >>> doc.paragraphs[0].text

'Document Title'

❹ >>> doc.paragraphs[1].text
```

'A plain paragraph with some bold and some italic'

```
❸ >>> len(doc.paragraphs[1].runs)
```

4

```
❹ >>> doc.paragraphs[1].runs[0].text
```

'A plain paragraph with some '

```
❺ >>> doc.paragraphs[1].runs[1].text
```

'bold'

```
❻ >>> doc.paragraphs[1].runs[2].text
```

' and some '

```
❼ >>> doc.paragraphs[1].runs[3].text
```

'italic'

At ❶, we open a .docx file in Python, call `docx.Document()`, and pass the filename `demo.docx`. This will return a `Document` object, which has a `paragraphs` attribute that is a list of `Paragraph` objects. When we call `len()` on `doc.paragraphs`, it returns 7, which tells us that there are seven `Paragraph` objects in this document ❷. Each of these `Paragraph` objects has a `text` attribute that contains a string of the text in that paragraph (without the style information). Here, the first text attribute contains 'DocumentTitle' ❸, and the second contains 'A plain paragraph with some bold and some italic' ❹.

Each `Paragraph` object also has a `runs` attribute that is a list of `Run` objects. `Run` objects also have a `text` attribute, containing just the text in that particular run. Let's look at the text attributes in the second `Paragraph` object, 'A plain paragraph with some bold and some italic'. Calling `len()` on this `Paragraph` object tells us that there are four `Run` objects ❺. The first run object contains 'A plain paragraph with some ' ❻. Then, the text change to a bold style, so 'bold' starts a new `Run` object ❼. The text returns to an unbolded style after that, which results in a third `Run` object, ' and some ' ❽. Finally, the fourth and last `Run` object contains 'italic' in an italic style ❾.

With Python-Docx, your Python programs will now be able to read the text from a .docx file and use it just like any other string value.

GETTING THE FULL TEXT FROM A .DOCX FILE

If you care only about the text, not the styling information, in the Word document, you can use the `getText()` function. It accepts a filename of a .docx file and returns a single string value of its text. Open a new file editor window and enter the following code, saving it as `readDocx.py`:

```
#!/python3

import docx

def getText(filename):
    doc = docx.Document(filename)
    fullText = []
    for para in doc.paragraphs:
        fullText.append(para.text)
    return '\n'.join(fullText)
```

The `getText()` function opens the Word document, loops over all the Paragraph objects in the paragraphs list, and then appends their text to the list in `fullText`. After the loop, the strings in `fullText` are joined together with newline characters.

The `readDocx.py` program can be imported like any other module. Now if you just need the text from a Word document, you can enter the following:

```
>>> import readDocx
>>> print(readDocx.getText('demo.docx'))
```

Document Title

A plain paragraph with some bold and some italic

Heading, level 1

Intense quote

first item in unordered list

first item in ordered list

You can also adjust `getText()` to modify the string before returning it. For example, to indent each paragraph, replace the `append()` call in `readDocx.py` with this:

```
fullText.append(' ' + para.text)
```

To add a double space in between paragraphs, change the `join()` call code to this:

```
return '\n\n'.join(fullText)
```

As you can see, it takes only a few lines of code to write functions that will read a `.docx` file and return a string of its content to your liking.

STYLING PARAGRAPH AND RUN OBJECTS

In Word for Windows, you can see the styles by pressing CTRL-ALT-SHIFT-S to display the Styles pane, which looks like [Figure 13-5](#). On OS X, you can view the Styles pane by clicking the View►Styles menu item.

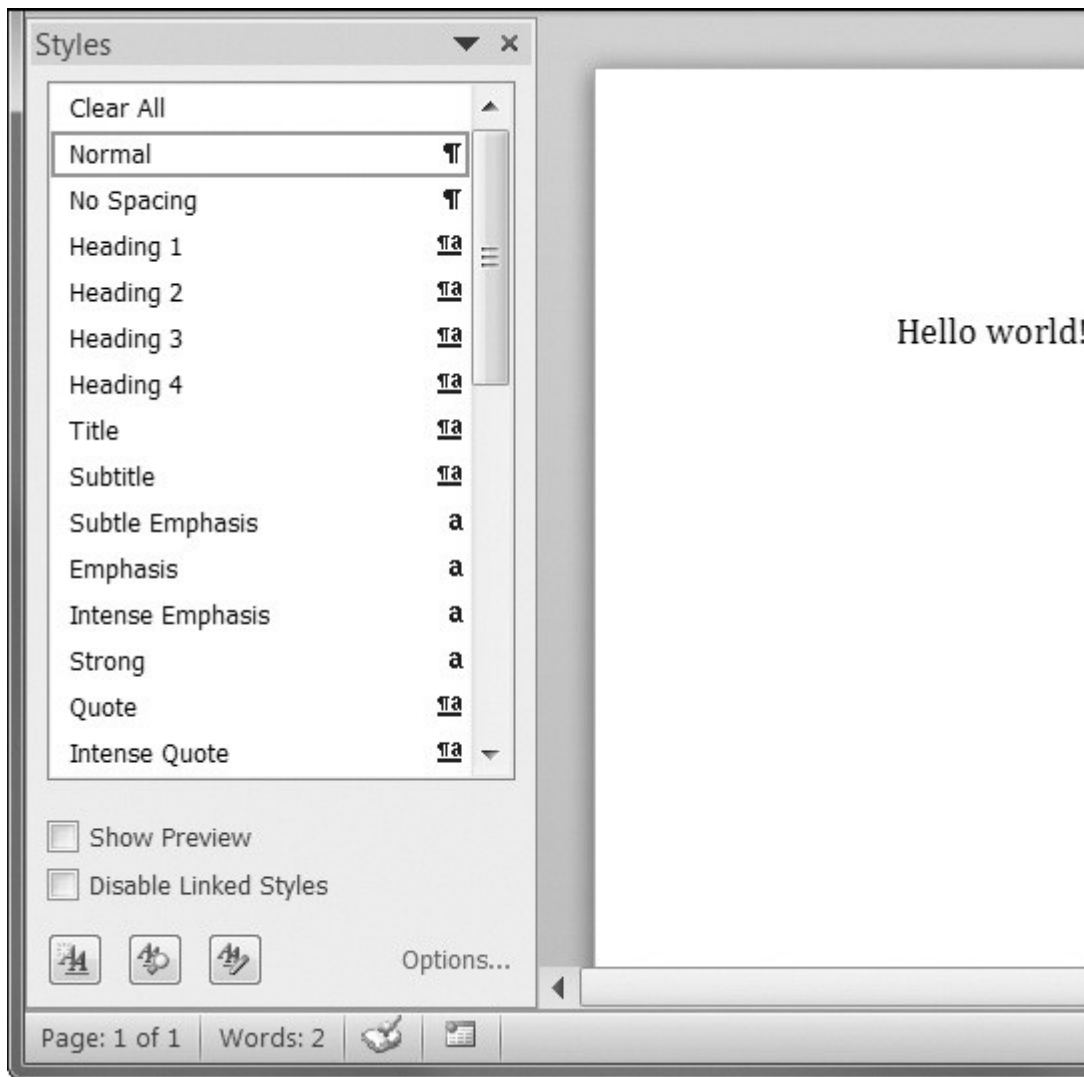


Figure 13-5. Display the Styles pane by pressing CTRL-ALT-SHIFT-S on Windows.

Word and other word processors use styles to keep the visual presentation of similar types of text consistent and easy to change. For example, perhaps you want to set body paragraphs in 11-point, Times New Roman, left-justified, ragged-right text. You can create a style with these settings and assign it to all body paragraphs. Then, if you later want to change the presentation of all body paragraphs in the document, you can just change the style, and all those paragraphs will be automatically updated.

For Word documents, there are three types of styles: Paragraph styles can be applied to Paragraph objects, character styles can be applied to Run objects, and linked styles can be applied to both kinds of objects. You can give both Paragraph and Run objects styles by setting their style attribute to a string. This string should be the name of a style.

If style is set to None, then there will be no style associated with the Paragraph or Run object.

The string values for the default Word styles are as follows:

'Normal'	'Heading5'	'ListBullet'	'ListParagraph'
'BodyText'	'Heading6'	'ListBullet2'	'MacroText'
'BodyText2'	'Heading7'	'ListBullet3'	'NoSpacing'
'BodyText3'	'Heading8'	'ListContinue'	'Quote'
'Caption'	'Heading9'	'ListContinue2'	'Subtitle'
'Heading1'	'IntenseQuote'	'ListContinue3'	'TOCHeading'
'Heading2'	'List'	'ListNumber'	'Title'
'Heading3'	'List2'	'ListNumber2'	
'Heading4'	'List3'	'ListNumber3'	

When setting the style attribute, do not use spaces in the style name. For example, while the style name may be Subtle Emphasis, you should set the style attribute to the string value 'SubtleEmphasis' instead of 'Subtle Emphasis'. Including spaces will cause Word to misread the style name and not apply it.

When using a linked style for a Run object, you will need to add 'Char' to the end of its name. For example, to set the Quote linked style for a Paragraph object, you would use `paragraphObj.style = 'Quote'`, but for a Run object, you would use `runObj.style = 'QuoteChar'`.

In the current version of Python-Docx (0.7.4), the only styles that can be used are the default Word styles and the styles in the opened .docx. New styles cannot be created—though this may change in future versions of Python-Docx.

CREATING WORD DOCUMENTS WITH NONDEFAULT STYLES

If you want to create Word documents that use styles beyond the default ones, you will need to open Word to a blank Word document and create

the styles yourself by clicking the New Style button at the bottom of the Styles pane ([Figure 13-6](#) shows this on Windows).

This will open the Create New Style from Formatting dialog, where you can enter the new style. Then, go back into the interactive shell and open this blank Word document with `docx.Document()`, using it as the base for your Word document. The name you gave this style will now be available to use with Python-Docx.

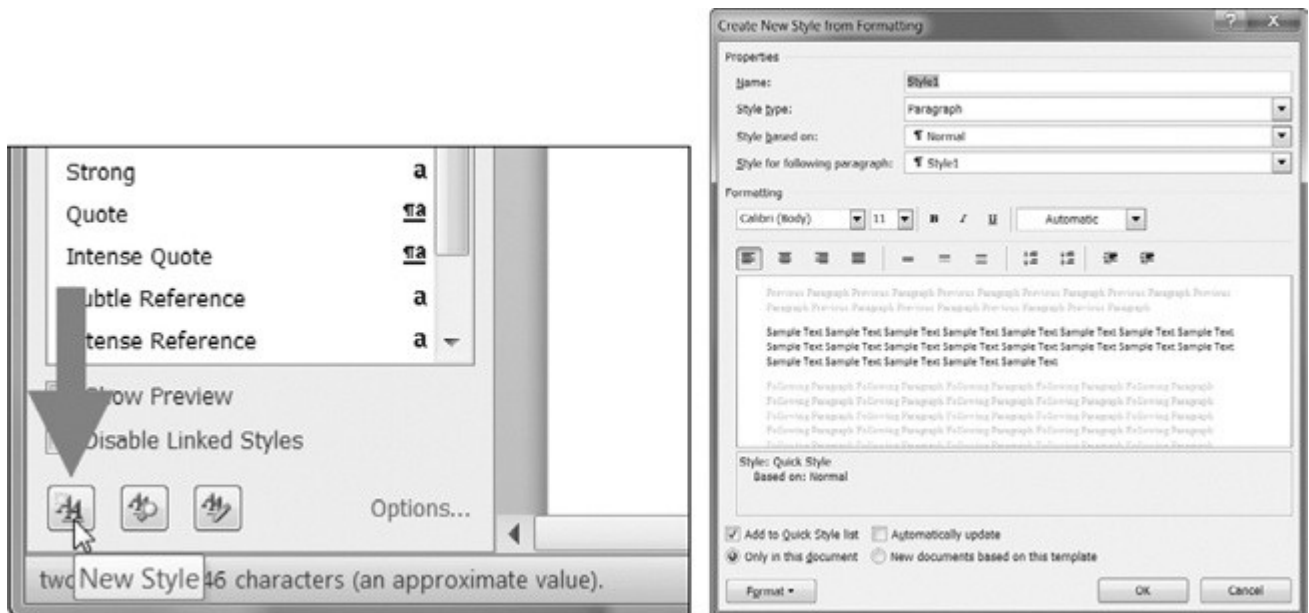


Figure 13-6. The New Style button (left) and the Create New Style from Formatting dialog (right)

RUN ATTRIBUTES

Runs can be further styled using text attributes. Each attribute can be set to one of three values: True (the attribute is always enabled, no matter what other styles are applied to the run), False (the attribute is always disabled), or None (defaults to whatever the run's style is set to).

[Table 13-1](#) lists the text attributes that can be set on Run objects.

Table 13-1. Run Object text Attributes

Attribute	Description
bold	The text appears in bold.
italic	The text appears in italic.

Attribute	Description
<code>underline</code>	The text is underlined.
<code>strike</code>	The text appears with strikethrough.
<code>double_strike</code>	The text appears with double strikethrough.
<code>all_caps</code>	The text appears in capital letters.
<code>small_caps</code>	The text appears in capital letters, with lowercase letters two points smaller.
<code>shadow</code>	The text appears with a shadow.
<code>outline</code>	The text appears outlined rather than solid.
<code>rtl</code>	The text is written right-to-left.
<code>imprint</code>	The text appears pressed into the page.
<code>emboss</code>	The text appears raised off the page in relief.

For example, to change the styles of `demo.docx`, enter the following into the interactive shell:

```
>>> doc = docx.Document('demo.docx')
>>> doc.paragraphs[0].text
'Document Title'
>>> doc.paragraphs[0].style
'Title'
>>> doc.paragraphs[0].style = 'Normal'
>>> doc.paragraphs[1].text
'A plain paragraph with some bold and some italic'
>>> (doc.paragraphs[1].runs[0].text, doc.paragraphs[1].runs[1].text, doc.
paragraphs[1].runs[2].text, doc.paragraphs[1].runs[3].text)
('A plain paragraph with some ', 'bold', ' and some ', 'italic')
>>> doc.paragraphs[1].runs[0].style = 'QuoteChar'
```

```
>>> doc.paragraphs[1].runs[1].underline = True
>>> doc.paragraphs[1].runs[3].underline = True
>>> doc.save('restyled.docx')
```

Here, we use the text and style attributes to easily see what's in the paragraphs in our document. We can see that it's simple to divide a paragraph into runs and access each run individually. So we get the first, second, and fourth runs in the second paragraph, style each run, and save the results to a new document.

The words Document Title at the top of restyled.docx will have the Normal style instead of the Title style, the Run object for the text A plain paragraph with some will have the QuoteChar style, and the two Run objects for the words bold and italic will have their underline attributes set to True. [Figure 13-7](#) shows how the styles of paragraphs and runs look in restyled.docx.

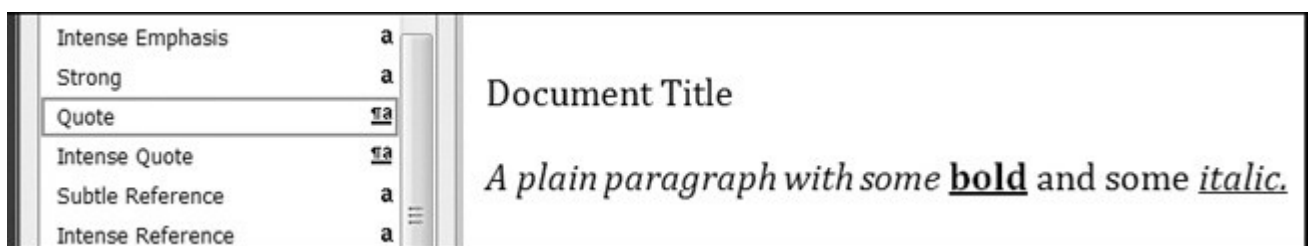


Figure 13-7. The restyled.docx file

You can find more complete documentation on Python-Docx's use of styles at <https://python-docx.readthedocs.org/en/latest/user/styles.html>.

WRITING WORD DOCUMENTS

Enter the following into the interactive shell:

```
>>> import docx
>>> doc = docx.Document()
>>> doc.add_paragraph('Hello world!')
<docx.text.Paragraph object at 0x0000000003B56F60>
>>> doc.save('helloworld.docx')
```

To create your own .docx file, call `docx.Document()` to return a new, blank Word Document object. The `add_paragraph()` document method adds a new paragraph of text to the document and returns a reference to the Paragraph object that was added. When you're done adding text, pass a filename string to the `save()` document method to save the Document object to a file.

This will create a file named `helloworld.docx` in the current working directory that, when opened, looks like [Figure 13-8](#).

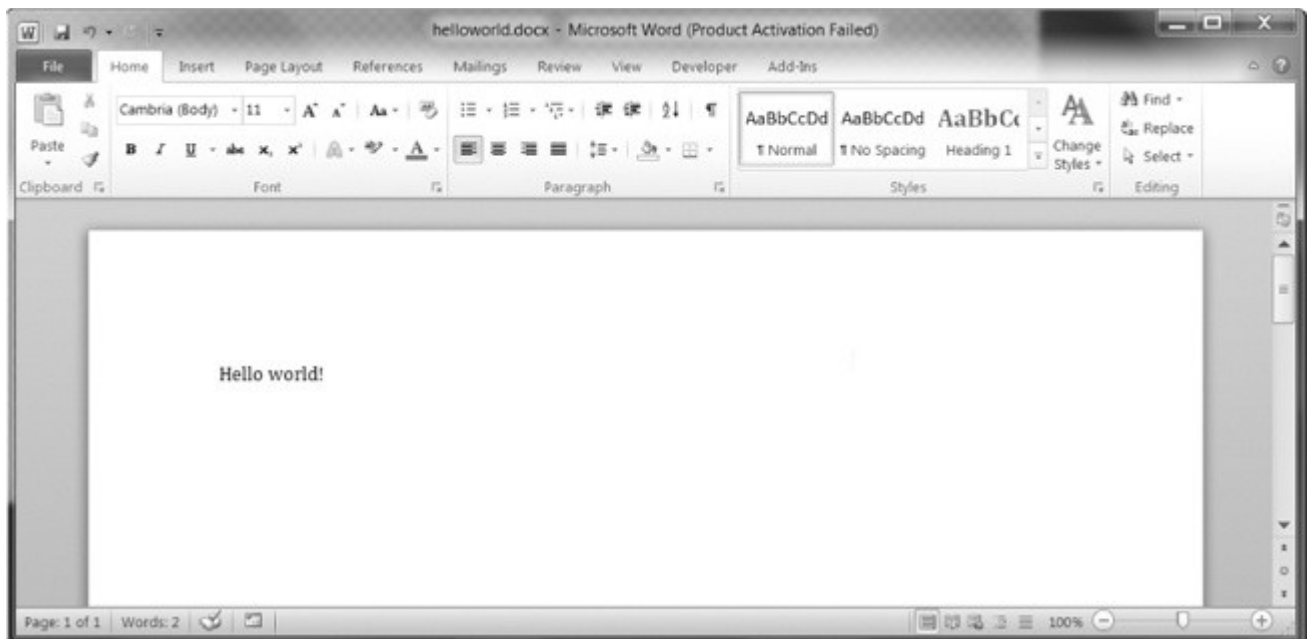


Figure 13-8. The Word document created using `add_paragraph('Hello world!')`

You can add paragraphs by calling the `add_paragraph()` method again with the new paragraph's text. Or to add text to the end of an existing paragraph, you can call the paragraph's `add_run()` method and pass it a string. Enter the following into the interactive shell:

```
>>> import docx
>>> doc = docx.Document()
>>> doc.add_paragraph('Hello world!')
<docx.text.Paragraph object at 0x000000000366AD30>
>>> paraObj1 = doc.add_paragraph('This is a second paragraph.')
```

```
>>> paraObj2 = doc.add_paragraph('This is a yet another paragraph.')
```

```
>>> paraObj1.add_run(' This text is being added to the second  
paragraph.')
```

```
<docx.text.Run object at 0x0000000003A2C860>
```

```
>>> doc.save('multipleParagraphs.docx')
```

The resulting document will look like [Figure 13-9](#). Note that the text This text is being added to the second paragraph. was added to the Paragraph object in paraObj1, which was the second paragraph added to doc. The add_paragraph() and add_run() functions return paragraph and Run objects, respectively, to save you the trouble of extracting them as a separate step.

Keep in mind that as of Python-Docx version 0.5.3, new Paragraph objects can be added only to the end of the document, and new Run objects can be added only to the end of a Paragraph object.

The save() method can be called again to save the additional changes you've made.

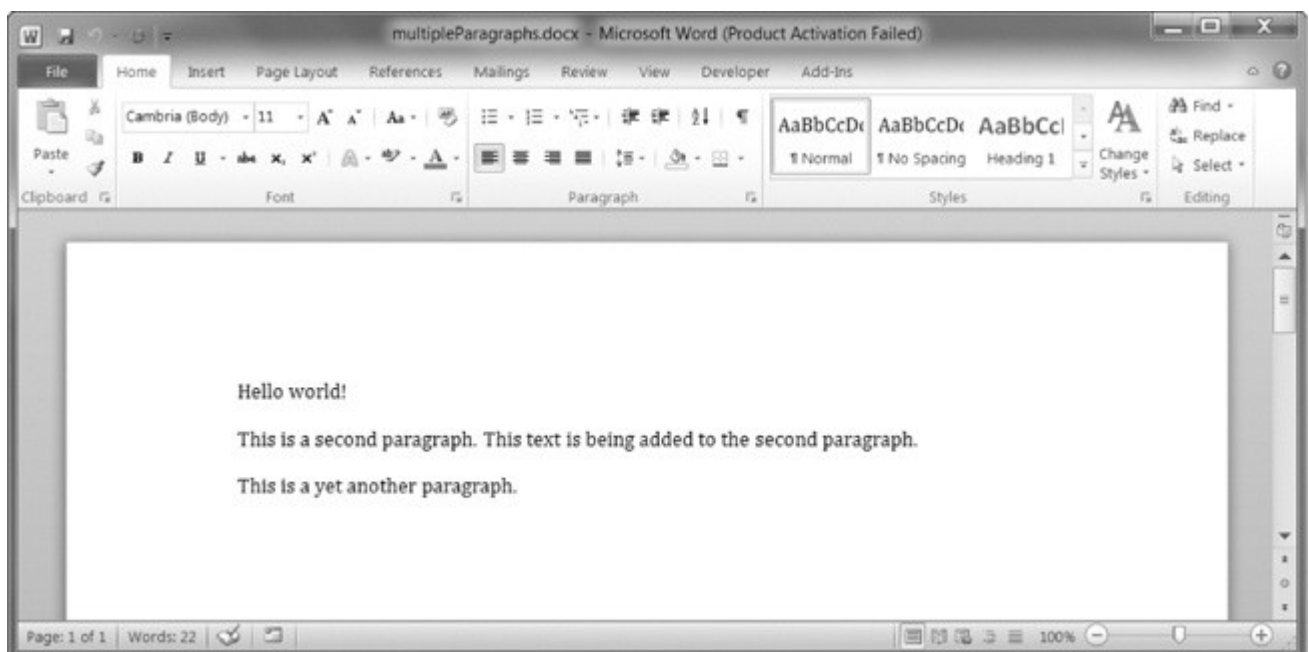


Figure 13-9. The document with multiple Paragraph and Run objects added

Both add_paragraph() and add_run() accept an optional second argument that is a string of the Paragraph or Run object's style. For example:

```
>>> doc.add_paragraph('Hello world!', 'Title')
```

This line adds a paragraph with the text Hello world! in the Title style.

ADDING HEADINGS

Calling `add_heading()` adds a paragraph with one of the heading styles. Enter the following into the interactive shell:

```
>>> doc = docx.Document()
>>> doc.add_heading('Header 0', 0)
<docx.text.Paragraph object at 0x00000000036CB3C8>
>>> doc.add_heading('Header 1', 1)
<docx.text.Paragraph object at 0x00000000036CB630>
>>> doc.add_heading('Header 2', 2)
<docx.text.Paragraph object at 0x00000000036CB828>
>>> doc.add_heading('Header 3', 3)
<docx.text.Paragraph object at 0x00000000036CB2E8>
>>> doc.add_heading('Header 4', 4)
<docx.text.Paragraph object at 0x00000000036CB3C8>
>>> doc.save('headings.docx')
```

The arguments to `add_heading()` are a string of the heading text and an integer from 0 to 4. The integer 0 makes the heading the Title style, which is used for the top of the document. Integers 1 to 4 are for various heading levels, with 1 being the main heading and 4 the lowest subheading.

The `add_heading()` function returns a Paragraph object to save you the step of extracting it from the Document object as a separate step.

The resulting `headings.docx` file will look like [Figure 13-10](#).

Header 0

Header 1

Header 2

Header 3

Header 4

Figure 13-10. The headings.docx document with headings 0 to 4

ADDING LINE AND PAGE BREAKS

To add a line break (rather than starting a whole new paragraph), you can call the `add_break()` method on the Run object you want to have the break appear after. If you want to add a page break instead, you need to pass the value `docx.text.WD_BREAK.PAGE` as a lone argument to `add_break()`, as is done in the middle of the following example:

```
>>> doc = docx.Document()

>>> doc.add_paragraph('This is on the first page!')
<docx.text.Paragraph object at 0x0000000003785518>

❶ >>> doc.paragraphs[0].runs[0].add_break(docx.text.WD_BREAK.PAGE)

>>> doc.add_paragraph('This is on the second page!')
<docx.text.Paragraph object at 0x00000000037855F8>

>>> doc.save('twoPage.docx')
```

This creates a two-page Word document with *This is on the first page!* on the first page and *This is on the second page!* on the second. Even though there was still plenty of space on the first page after the text *This is on the first page!*, we forced the next paragraph to begin on a new page by inserting a page break after the first run of the first paragraph ❶.

ADDING PICTURES

Document objects have an `add_picture()` method that will let you add an image to the end of the document. Say you have a file `zophie.png` in the

current working directory. You can add `zophie.png` to the end of your document with a width of 1 inch and height of 4 centimeters (Word can use both imperial and metric units) by entering the following:

```
>>> doc.add_picture('zophie.png', width=docx.shared.Inches(1),  
height=docx.shared.Cm(4))  
<docx.shape.InlineShape object at 0x00000000036C7D30>
```

The first argument is a string of the image's filename. The optional width and height keyword arguments will set the width and height of the image in the document. If left out, the width and height will default to the normal size of the image.

You'll probably prefer to specify an image's height and width in familiar units such as inches and centimeters, so you can use the `docx.shared.Inches()` and `docx.shared.Cm()` functions when you're specifying the width and height keyword arguments.

SUMMARY

Text information isn't just for plaintext files; in fact, it's pretty likely that you deal with PDFs and Word documents much more often. You can use the `PyPDF2` module to read and write PDF documents. Unfortunately, reading text from PDF documents might not always result in a perfect translation to a string because of the complicated PDF file format, and some PDFs might not be readable at all. In these cases, you're out of luck unless future updates to `PyPDF2` support additional PDF features.

Word documents are more reliable, and you can read them with the `python-docx` module. You can manipulate text in Word documents via `Paragraph` and `Run` objects. These objects can also be given styles, though they must be from the default set of styles or styles already in the document. You can add new paragraphs, headings, breaks, and pictures to the document, though only to the end.

Many of the limitations that come with working with PDFs and Word documents are because these formats are meant to be nicely displayed for human readers, rather than easy to parse by software. The next chapter takes a look at two other common formats for storing information: JSON

and CSV files. These formats are designed to be used by computers, and you'll see that Python can work with these formats much more easily.

PRACTICE QUESTIONS

- Q 1. A string value of the PDF filename is *not* passed to
: the `PyPDF2.PdfFileReader()` function. What do you pass to the function instead?
- Q 2. What modes do the `File` objects for `PdfFileReader()` and `PdfFileWriter()` need
: to be opened in?
- Q 3. How do you acquire a `Page` object for [About This Book](#) from a `PdfFileReader` object?
:
- Q 4. What `PdfFileReader` variable stores the number of pages in the PDF document?
:
- Q 5. If a `PdfFileReader` object's PDF is encrypted with the password `swordfish`, what
: must you do before you can obtain `Page` objects from it?
- Q 6. What methods do you use to rotate a page?
:
- Q 7. What method returns a `Document` object for a file named `demo.docx`?
:
- Q 8. What is the difference between a `Paragraph` object and a `Run` object?
:
- Q 9. How do you obtain a list of `Paragraph` objects for a `Document` object that's stored in a
: variable named `doc`?
- Q 10. What type of object has `bold`, `underline`, `italic`, `strike`, and `outline` variables?
:
- Q 11. What is the difference between setting the `bold` variable to `True`, `False`, or `None`?
:
- Q 12. How do you create a `Document` object for a new Word document?
:
- Q 13. How do you add a paragraph with the text `'Hello there!'` to a `Document` object
: stored in a variable named `doc`?

Q 14. What integers represent the levels of headings available in Word documents?
:

PRACTICE PROJECTS

For practice, write programs that do the following.

PDF PARANOIA

Using the `os.walk()` function from [Chapter 9](#), write a script that will go through every PDF in a folder (and its subfolders) and encrypt the PDFs using a password provided on the command line. Save each encrypted PDF with an `_encrypted.pdf` suffix added to the original filename. Before deleting the original file, have the program attempt to read and decrypt the file to ensure that it was encrypted correctly.

Then, write a program that finds all encrypted PDFs in a folder (and its subfolders) and creates a decrypted copy of the PDF using a provided password. If the password is incorrect, the program should print a message to the user and continue to the next PDF.

CUSTOM INVITATIONS AS WORD DOCUMENTS

Say you have a text file of guest names. This `guests.txt` file has one name per line, as follows:

Prof. Plum

Miss Scarlet

Col. Mustard

Al Sweigart

Robocop

Write a program that would generate a Word document with custom invitations that look like [Figure 13-11](#).

Since Python-Docx can use only those styles that already exist in the Word document, you will have to first add these styles to a blank Word file and then open that file with Python-Docx. There should be one invitation per page in the resulting Word document, so call `add_break()` to add a page

break after the last paragraph of each invitation. This way, you will need to open only one Word document to print all of the invitations at once.

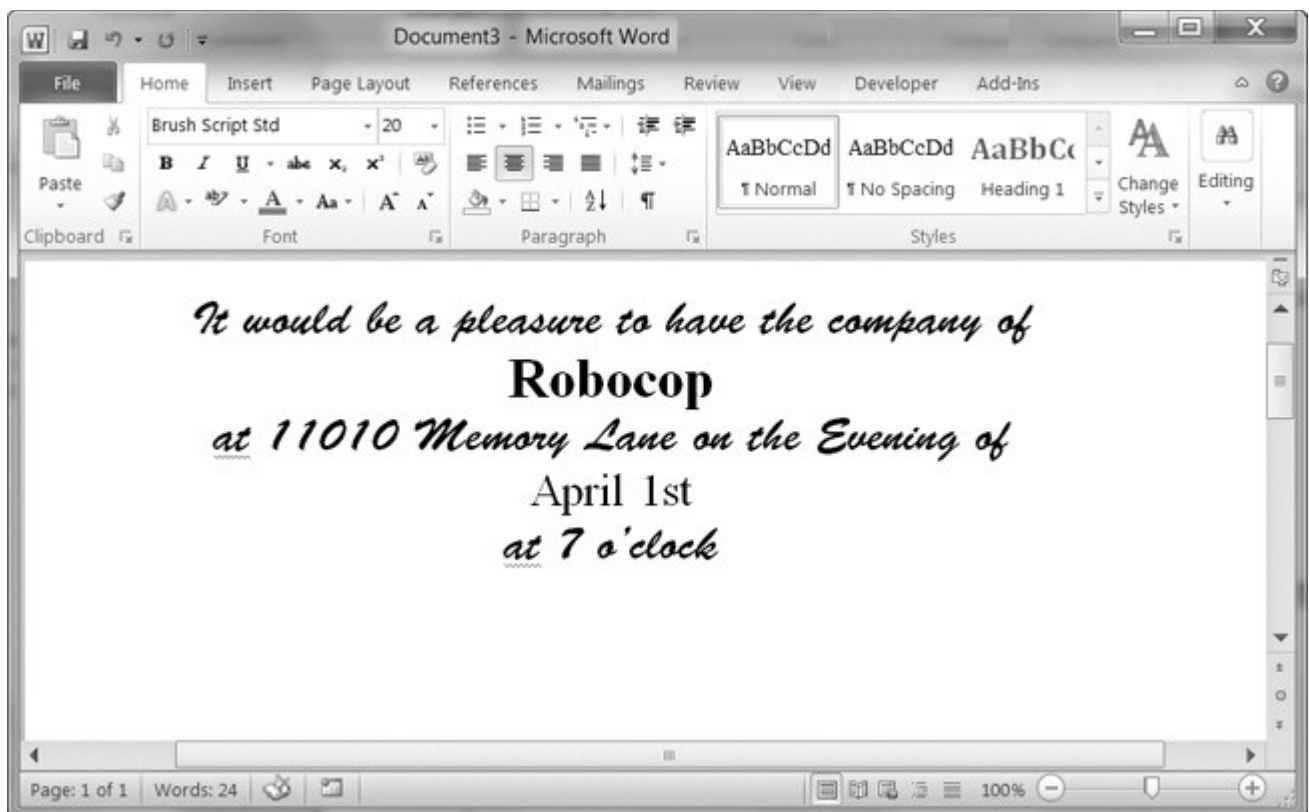


Figure 13-11. The Word document generated by your custom invite script

You can download a sample guests.txt file from <http://nostarch.com/automatestuff/>.

BRUTE-FORCE PDF PASSWORD BREAKER

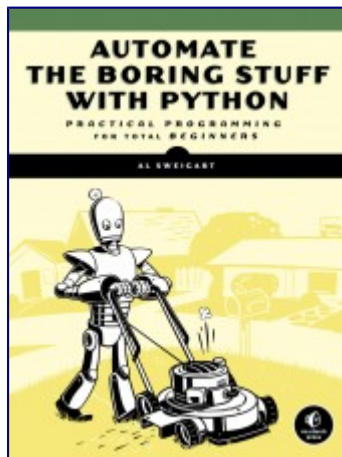
Say you have an encrypted PDF that you have forgotten the password to, but you remember it was a single English word. Trying to guess your forgotten password is quite a boring task. Instead you can write a program that will decrypt the PDF by trying every possible English word until it finds one that works. This is called a brute-force password attack. Download the text file dictionary.txt from <http://nostarch.com/automatestuff/>. This dictionary file contains over 44,000 English words with one word per line.

Using the file-reading skills you learned in [Chapter 8](#), create a list of word strings by reading this file. Then loop over each word in this list, passing it to the decrypt() method. If this method returns the integer 0, the

password was wrong and your program should continue to the next password. If `decrypt()` returns 1, then your program should break out of the loop and print the hacked password. You should try both the uppercase and lower-case form of each word. (On my laptop, going through all 88,000 uppercase and lowercase words from the dictionary file takes a couple of minutes. This is why you shouldn't use a simple English word for your passwords.)



Support the author by purchasing the print & ebook bundle from [No Starch Press](#) or separately on [Amazon](#).



Read the author's other Creative Commons licensed Python books.

