***Reading Word Documents***

Let’s experiment with the docx module. Download *demo.docx* from [*https://nostarch.com/automatestuff2/*](https://nostarch.com/automatestuff2/) and save the document to the working directory. Then enter the following into the interactive shell:

   >>> **import docx**  
➊ >>> **doc = docx.Document('demo.docx')**  
➋ >>> **len(doc.paragraphs)**  
   7  
➌ >>> **doc.paragraphs[0].text**  
   'Document Title'  
➍ >>> **doc.paragraphs[1].text**  
   'A plain paragraph with some bold and some italic'  
➎ >>> **len(doc.paragraphs[1].runs)**  
   4  
➏ >>> **doc.paragraphs[1].runs[0].text**  
   'A plain paragraph with some '  
➐ >>> **doc.paragraphs[1].runs[1].text**  
   'bold'  
➑ >>> **doc.paragraphs[1].runs[2].text**  
   ' and some '  
➒ >>> **doc.paragraphs[1].runs[3].text**  
   'italic'

At ➊, 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 ➋. 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' ➌, and the second contains 'A plain paragraph with some bold and some italic' ➍.

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 ➎. The first run object contains 'A plain paragraph with some ' ➏. Then, the text changes to a bold style, so 'bold' starts a new Run object ➐. The text returns to an unbolded style after that, which results in a third Run object, ' and some ' ➑. Finally, the fourth and last Run object contains 'italic' in an italic style ➒.

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 tab 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 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 15-5](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-8). On macOS, you can view the Styles pane by clicking the **View** ▸ **Styles** menu item.

*Figure 15-5: 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'  'Body Text'  'Body Text 2'  'Body Text 3'  'Caption'  'Heading 1'  'Heading 2'  'Heading 3'  'Heading 4' | 'Heading 5'  'Heading 6'  'Heading 7'  'Heading 8'  'Heading 9'  'Intense Quote'  'List'  'List 2'  'List 3' | 'List Bullet'  'List Bullet 2'  'List Bullet 3'  'List Continue'  'List Continue 2'  'List Continue 3'  'List Number '  'List Number 2'  'List Number 3' | 'List Paragraph'  'MacroText'  'No Spacing'  'Quote'  'Subtitle'  'TOC Heading'  'Title' |

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 = 'Quote Char'.

In the current version of Python-Docx (0.8.10), 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 15-6](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-9) 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 15-6: 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 15-1](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-10) lists the text attributes that can be set on Run objects.

**Table 15-1:** Run Object text Attributes

| **Attribute** | **Description** |
| --- | --- |
| bold | The text appears in bold. |
| italic | The text appears in italic. |
| underline | The text is underlined. |
| strike | The text appears with strikethrough. |
| double\_strike | The text appears with double strikethrough. |
| all\_caps | The text appears in capital letters. |
| small\_caps | The text appears in capital letters, with lowercase letters two points smaller. |
| shadow | The text appears with a shadow. |
| outline | The text appears outlined rather than solid. |
| rtl | The text is written right-to-left. |
| imprint | The text appears pressed into the page. |
| emboss | 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:

>>> **import docx**  
>>> **doc = docx.Document('demo.docx')**  
>>> **doc.paragraphs[0].text**  
'Document Title'  
>>> **doc.paragraphs[0].style # The exact id may be different:**  
\_ParagraphStyle('Title') id: 3095631007984  
>>> **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 15-7](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-11) shows how the styles of paragraphs and runs look in *restyled.docx*.

*Figure 15-7: The* restyled.docx *file*

You can find more complete documentation on Python-Docx’s use of styles at *https://python-docx.readthedocs.io/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 15-8](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-12).

*Figure 15-8: 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 15-9](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-13). 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 0.8.10, 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 15-9: 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. Here’s an 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 15-10](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-14).

*Figure 15-10: 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.enum.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>  
➊ >>> **doc.paragraphs[0].runs[0].add\_break(docx.enum.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 ➊.

***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.

**Creating PDFs from Word Documents**

The PyPDF2 module doesn’t allow you to create PDF documents directly, but there’s a way to generate PDF files with Python if you’re on Windows and have Microsoft Word installed. You’ll need to install the Pywin32 package by running pip install --user -U pywin32==224. With this and the docx module, you can create Word documents and then convert them to PDFs with the following script.

Open a new file editor tab, enter the following code, and save it as *convertWordToPDF.py*:

# This script runs on Windows only, and you must have Word installed.  
import win32com.client # install with "pip install pywin32==224"  
import docx  
wordFilename = '*your\_word\_document*.docx'  
pdfFilename = '*your\_pdf\_filename*.pdf'  
  
doc = docx.Document()  
# Code to create Word document goes here.  
doc.save(wordFilename)  
  
wdFormatPDF = 17 # Word's numeric code for PDFs.  
wordObj = win32com.client.Dispatch('Word.Application')  
  
docObj = wordObj.Documents.Open(wordFilename)  
docObj.SaveAs(pdfFilename, FileFormat=wdFormatPDF)  
docObj.Close()  
wordObj.Quit()

To write a program that produces PDFs with your own content, you must use the docx module to create a Word document, then use the Pywin32 package’s win32com.client module to convert it to a PDF. Replace the # Code to create Word document goes here. comment with docx function calls to create your own content for the PDF in a Word document.

This may seem like a convoluted way to produce PDFs, but as it turns out, professional software solutions are often just as complicated.

**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 package’s 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**

[1](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-15). A string value of the PDF filename is *not* passed to the PyPDF2.PdfFileReader() function. What do you pass to the function instead?

[2](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-16). What modes do the File objects for PdfFileReader() and PdfFileWriter() need to be opened in?

[3](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-17). How do you acquire a Page object for [page 5](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-18) from a PdfFileReader object?

[4](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-19). What PdfFileReader variable stores the number of pages in the PDF document?

[5](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-20). If a PdfFileReader object’s PDF is encrypted with the password swordfish, what must you do before you can obtain Page objects from it?

[6](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-21). What methods do you use to rotate a page?

[7](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-22). What method returns a Document object for a file named *demo.docx*?

[8](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-23). What is the difference between a Paragraph object and a Run object?

[9](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-24). How do you obtain a list of Paragraph objects for a Document object that’s stored in a variable named doc?

[10](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-25). What type of object has bold, underline, italic, strike, and outline variables?

[11](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-26). What is the difference between setting the bold variable to True, False, or None?

[12](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-27). How do you create a Document object for a new Word document?

[13](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-28). How do you add a paragraph with the text 'Hello, there!' to a Document object stored in a variable named doc?

[14](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-29). 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 10](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-30), 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 15-11](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-31).

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 15-11: The Word document generated by your custom invite script*

You can download a sample *guests.txt* file from [*https://nostarch.com/automatestuff2/*](https://nostarch.com/automatestuff2/).

***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 [*https://nostarch.com/automatestuff2/*](https://nostarch.com/automatestuff2/). This *dictionary file* contains over 44,000 English words with one word per line.

Using the file-reading skills you learned in [Chapter 9](https://automatetheboringstuff.com/2e/chapter15/#calibre_link-32), 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 lowercase 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.)