Module 5 : Chapter 3. 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.
- If the module was installed correctly, running import PyPDF2 in the interactive shell shouldn't display any errors.

-> 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.

In [4]:

```
>>> import PyPDF2
>>> pdfFileObj = open('ADPsyll.pdf', 'rb')
>>> pdfReader = PyPDF2.PdfFileReader(pdfFileObj)
>>> pdfReader.numPages
>>> pageObj = pdfReader.getPage(0)
>>> pageObj.extractText()
```

Out[4]:

```
'Module-1PythonBasics8hours\nPythonBasics,\nEnteringExpressionsintotheInt
eractiveShell,TheInteger,Floating-Point,andStringData\nTypes,StringConcat
enationandReplication,StoringValuesinVariables,YourFirstProgram,\nDissect
ingYourProgram,\nFlowcontrol,\nBooleanValues,ComparisonOperators,BooleanO
perators,MixingBooleanandComparison\nOperators,ElementsofFlowControl,Prog
ramExecution,FlowControlStatements,Importing\nModules,EndingaProgramEarly
withsys.exit(),\nFunctions,\ndefStatementswithParameters,ReturnValuesandr
eturnStatements,TheNoneValue,\nKeywordArgumentsandprint(),LocalandGlobalS
cope,TheglobalStatement,Exception\nHandling,AShortProgram:GesstheNumber
\nTextbook1:Chapters1Œ3\nRBT:L1,L2\nModule-2Lists8hours\nLists,\nTheListD
ataType,WorkingwithLists,AugmentedAssignmentOperators,Methods,\nExamplePr
ogram:Magic8BallwithaList,List-likeTypes:StringsandTuples,References,\nDi
ctionariesandStructuringData,\nTheDictionaryDataType,PrettyPrinting,Using
DataStructurestoModelReal-WorldThings,\nManipulatingStrings,\nWorkingwith
Strings,UsefulStringMethods,Project:PasswordLocker,Project:Adding\nBullet
stoWikiMarkup\nTextbook1:Chapters4Œ6\nRBT:L1,L2,L3\nModule-3PatternMatchi
ngwithRegularExpressions8hours\nPatternMatchingwithRegularExpressions,\nF
indingPatternsofTextWithoutRegularExpressions,FindingPatternsofTextwithRe
```

->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:

In [5]:

```
>>> import PyPDF2
>>> pdfReader = PyPDF2.PdfFileReader(open('encryptedassignment3.pdf', 'rb'))
>>> pdfReader.isEncrypted
>>>
```

Out[5]:

```
True
```

In [8]:

```
>>> pdfReader.getPage(0)
>>> pdfReader.decrypt('hkbkcse')
>>> pageObj = pdfReader.getPage(0)
>>> pageObj.extractText()
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-8-e6abf816c959> in <module>
----> 1 pdfReader.getPage(0)
      2 pdfReader.decrypt('hkbkcse')
      3 pageObj = pdfReader.getPage(0)
      4 pageObj.extractText()

~\Anaconda3\lib\site-packages\PyPDF2\pdf.py in getPage(self, pageNumber)
   1175         if self.flattenedPages == None:
   1176             self._flatten()
-> 1177         return self.flattenedPages[pageNumber]
   1178
   1179     namedDestinations = property(lambda self:

IndexError: list index out of range
```

-> Creating PDFs

In [ ]:

```

```

- 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/ (http://nostarch.com/automatestuff/) and place the PDFs in the current working directory.
- Enter the following into the interactive shell:

In [10]:

```
import PyPDF2
>>> pdf1File = open('Module-3 Assignment.pdf', 'rb')
>>> pdf2File = open('Module-4 Assignment.pdf', 'rb')
>>> pdf1Reader = PyPDF2.PdfFileReader(pdf1File)
>>> pdf2Reader = PyPDF2.PdfFileReader(pdf2File)
>>> pdfWriter = PyPDF2.PdfFileWriter()
>>> for pageNum in range(pdf1Reader.numPages):
        pageObj = pdf1Reader.getPage(pageNum)
        pdfWriter.addPage(pageObj)
>>> for pageNum in range(pdf2Reader.numPages):
        pageObj = pdf2Reader.getPage(pageNum)
        pdfWriter.addPage(pageObj)
>>> pdfOutputFile = open('Assignment3and4.pdf', 'wb')
>>> pdfWriter.write(pdfOutputFile)
>>> pdfOutputFile.close()
>>> pdf1File.close()
>>> pdf2File.close()
```

-> 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:

In [12]:

```
>>> import PyPDF2
>>> minutesFile = open('Assignment3and4.pdf', 'rb')
>>> pdfReader = PyPDF2.PdfFileReader(minutesFile)
>>> page = pdfReader.getPage(0)
>>> page.rotateClockwise(90)
>>> pdfWriter = PyPDF2.PdfFileWriter()
>>> pdfWriter.addPage(page)
>>> resultPdfFile = open('rotatedassignment3and4.pdf', 'wb')
>>> pdfWriter.write(resultPdfFile)
>>> resultPdfFile.close()
>>> minutesFile.close()
```

-> 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/ (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:

In [13]:

```
>>> import PyPDF2
>>> minutesFile = open('rotatedassignment3and4.pdf', 'rb')
>>> pdfReader = PyPDF2.PdfFileReader(minutesFile)
>>> minutesFirstPage = pdfReader.getPage(0)
>>> pdfWatermarkReader = PyPDF2.PdfFileReader(open('watermark.pdf', 'rb'))
>>> minutesFirstPage.mergePage(pdfWatermarkReader.getPage(0))
>>> pdfWriter = PyPDF2.PdfFileWriter()
>>> pdfWriter.addPage(minutesFirstPage)
>>> for pageNum in range(1, pdfReader.numPages):
    pageObj = pdfReader.getPage(pageNum)
    pdfWriter.addPage(pageObj)
>>> resultPdfFile = open('waterCoverpage.pdf', 'wb')
>>> pdfWriter.write(resultPdfFile)
>>> minutesFile.close()
>>> resultPdfFile.close()
```

-> Encrypting PDFs

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

In [17]:

```
import PyPDF2
>>> pdfFile = open('ADPsyll.pdf', 'rb')
>>> pdfReader = PyPDF2.PdfFileReader(pdfFile)
>>> pdfWriter = PyPDF2.PdfFileWriter()
>>> for pageNum in range(pdfReader.numPages):
    pdfWriter.addPage(pdfReader.getPage(pageNum))
>>> pdfWriter.encrypt('cse123')
>>> resultPdf = open('passwordadpsyll.pdf', 'wb')
>>> pdfWriter.write(resultPdf)
>>> resultPdf.close()
```

==>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:

    1. Find all PDF files in the current working directory.
    2. Sort the filenames so the PDFs are added in order.
    3. 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:

1. Call os.listdir() to find all the files in the working directory and remove any non-PDF files.
2. Call Python's sort() list method to alphabetize the filenames.
3. Create a PdfFileWriter object for the output PDF.
4. Loop over each PDF file, creating a PdfFileReader object for it.
5. Loop over each page (except the first) in each PDF file.
6. Add the pages to the output PDF.
7. 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.

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 pythondocx. (Appendix A has full details on installing third-party modules.)

- 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 (https://www.libreoffice.org) and http://openoffice.org (http://openoffice.org), respectively.
- The full documentation for Python-Docx is available at https://pythondocx (https://pythondocx). 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.

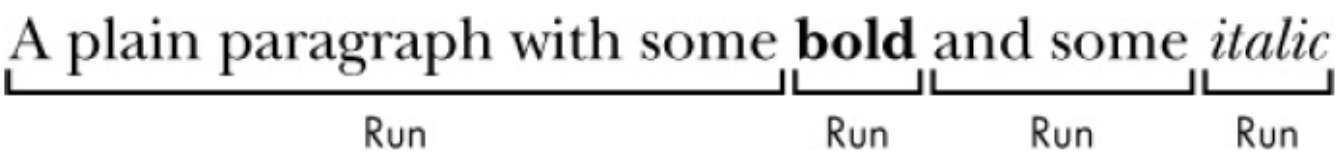

Figure 13-4. The Run objects identified in a Paragraph object

- 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/ (http://nostarch.com/automatestuff/) and save the document to the working directory. Then enter the following into the interactive shell:

In [6]:

```
>>> import docx
>>> doc = docx.Document('Module-3 Assignment.docx')
>>> len(doc.paragraphs)
```

Out[6]:

```
13
```

In [9]:

```
>>> doc.paragraphs[0].text
```

Out[9]:

```
'HKBK College of Engg,B’luru, K’taka'
```

In [10]:

```
doc.paragraphs[1].text
```

Out[10]:

```
'Dept. of Computer Science and Engineering'
```

In [11]:

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

Out[11]:

```
5
```

In [12]:

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

Out[12]:

```
'Dept. of '
```

In [13]:

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

Out[13]:

```
'Computer '
```

In [14]:

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

Out[14]:

```
'Science and '
```

In [15]:

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

Out[15]:

```
'Eng'
```

In [16]:

```
doc.paragraphs[1].runs[4].text
```

Out[16]:

```
'ineering'
```

-> 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:

In [18]:

```
import docx
def getText(filename1):
    doc = docx.Document(filename1)
    fullText = []
    for para in doc.paragraphs:

        fullText.append(' ' + para.text)
    return '\n'.join(fullText)
```

In [19]:

```
import readDocx
print(readDocx.getText('Module-3 Assignment.docx'))
```

```
HKBK College of Engg,B’luru, K’taka
Dept. of Computer Science and Engineering
APPLICATION DEVELOMENT PROGRAMMING USING PYTHON (18CS55)

Module-4 Assignment -3

1. Implement a python program to find for lines having ‘@’ sign between char
acters in a read text file.
2. Write a python program to read all the lines in a file accepted from the
user and print all email address contained in it. Assume the email address c
ontain only non-white space characters.

3. Write a python program to search for lines that start with the word ‘Fro
m’ and a character followed by a two digit number between 00 to 99 followed
by ‘:’ print the number if it is greater than zero. Assume any input file.

4. Write a python program to search lines that starts with ‘X’ followed by a
ny non-whitespace characters, followed by ‘:’ ending with number. Display th
e sum of all these number.
```

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.

In [ ]:

```
```

->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.

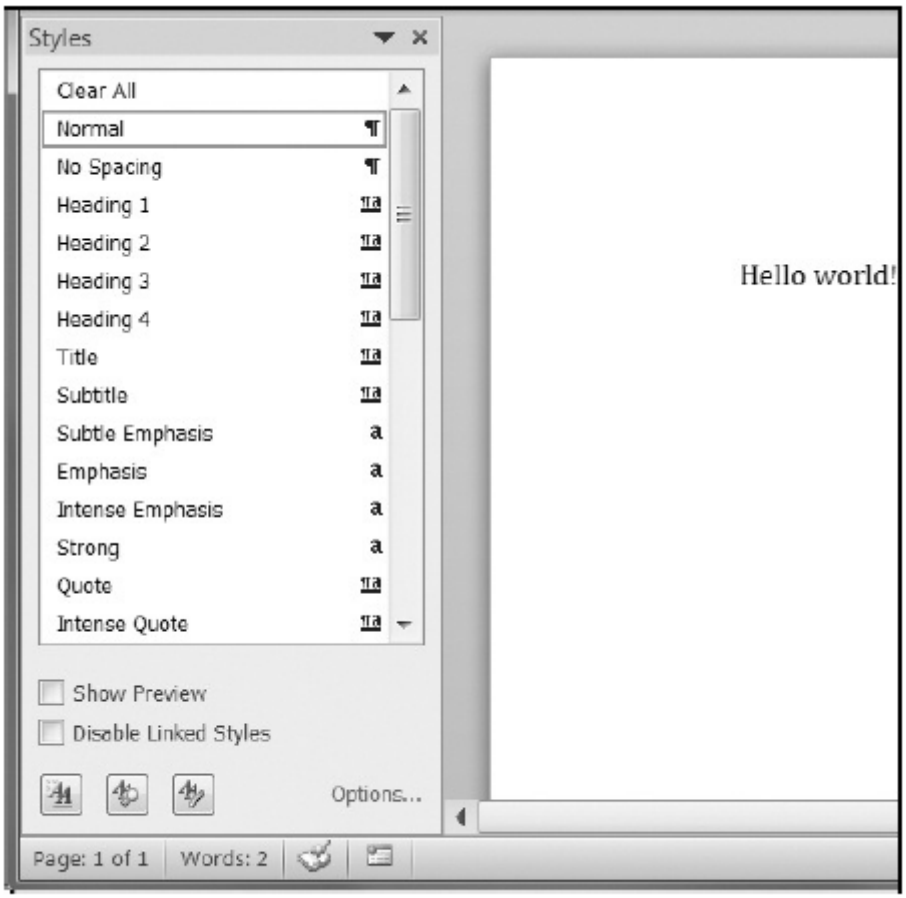

Figure 13-5. Display the Styles pane by pressing CTRL-ALT-SHIFT-S on Windows.

- 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.

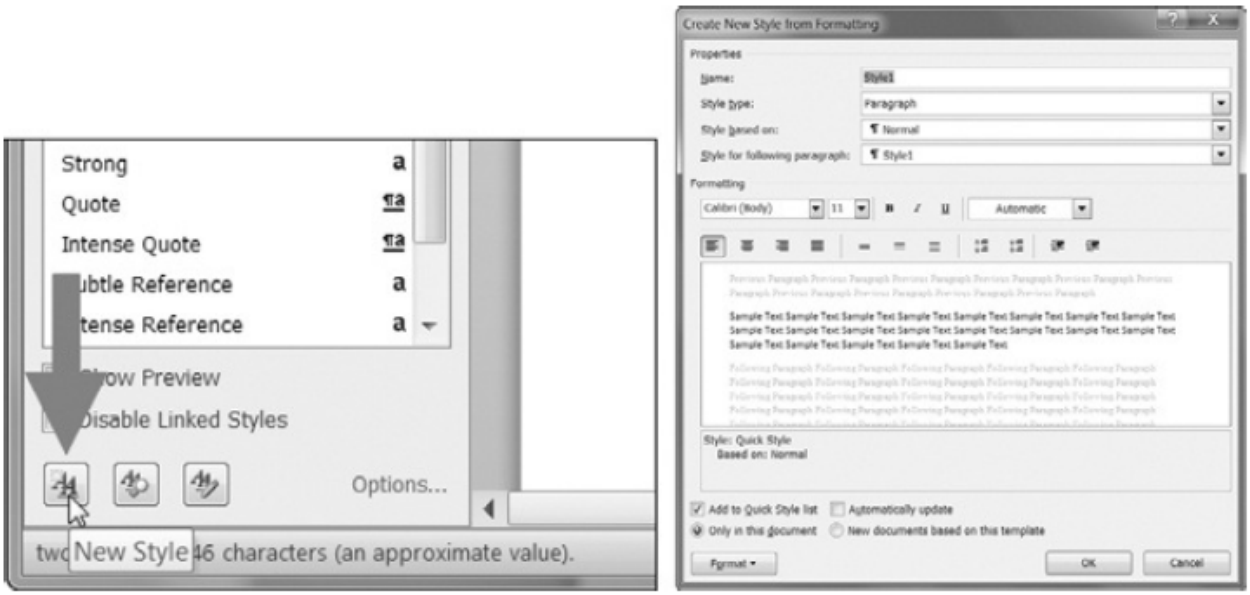

Figure 13-6. The New Style button (left) and the Create New Style from Formatting dialog (right)

-> 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.
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:

In [20]:

```
doc = docx.Document('Module-3 Assignment.docx')
>>> doc.paragraphs[0].text
```

Out[20]:

```
'HKBK College of Engg,B’luru, K’taka'
```

In [21]:

```
doc.paragraphs[0].style
```

Out[21]:

```
_ParagraphStyle('Normal') id: 3040944095560
```

In [27]:

```
doc.paragraphs[0].style = 'Normal'
```

In [28]:

```
doc.paragraphs[1].text
```

Out[28]:

```
'Dept. of Computer Science and Engineering'
```

In [29]:

```
(doc.paragraphs[1].runs[0].text, doc.paragraphs[1].runs[1].text, doc.
paragraphs[1].runs[2].text, doc.paragraphs[1].runs[3].text)
```

Out[29]:

```
('Dept. of ', 'Computer ', 'Science and ', 'Eng')
```

In [32]:

```
>>> doc.paragraphs[1].runs[0].style = 'Normal'
>>> doc.paragraphs[1].runs[1].underline = True
>>> doc.paragraphs[1].runs[3].underline = True
>>> doc.save('restyled.docx')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-32-9b31d278bcf3> in <module>
----> 1 doc.paragraphs[1].runs[0].style = 'Normal'
      2 doc.paragraphs[1].runs[1].underline = True
      3 doc.paragraphs[1].runs[3].underline = True
      4 doc.save('restyled.docx')

~\Anaconda3\lib\site-packages\docx\text\run.py in style(self, style_or_name)
    135     def style(self, style_or_name):
    136         style_id = self.part.get_style_id(
--> 137             style_or_name, WD_STYLE_TYPE.CHARACTER
    138         )
    139         self._r.style = style_id

~\Anaconda3\lib\site-packages\docx\parts\document.py in get_style_id(self, s
tyle_or_name, style_type)
     76         present in the document.
     77         """
---> 78         return self.styles.get_style_id(style_or_name, style_type)
     79
     80     def header_part(self, rId):

~\Anaconda3\lib\site-packages\docx\styles\styles.py in get_style_id(self, st
yle_or_name, style_type)
    107             return self._get_style_id_from_style(style_or_name, styl
e_type)
    108         else:
--> 109             return self._get_style_id_from_name(style_or_name, style
_type)
    110
    111     @property

~\Anaconda3\lib\site-packages\docx\styles\styles.py in _get_style_id_from_na
me(self, style_name, style_type)
    137         the document or does not match *style_type*.
    138         """
--> 139         return self._get_style_id_from_style(self[style_name], style
_type)
    140
    141     def _get_style_id_from_style(self, style, style_type):

~\Anaconda3\lib\site-packages\docx\styles\styles.py in _get_style_id_from_st
yle(self, style, style_type)
    147             raise ValueError(
    148                 "assigned style is type %s, need type %s" %
--> 149                 (style.type, style_type)
    150             )
    151         if style == self.default(style_type):

ValueError: assigned style is type PARAGRAPH (1), need type CHARACTER (2)
```

->Writing Word Documents

- Enter the following into the interactive shell:

In [36]:

```
>>> import docx
>>> doc = docx.Document()
>>> doc.add_paragraph('Hello world!')
```

Out[36]:

```
<docx.text.paragraph.Paragraph at 0x2c4066438c8>
```

In [34]:

```
doc.save('helloworld123.docx')
```

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

- 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.

- 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:

In [37]:

```
>>> 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.')
```

Out[37]:

```
<docx.text.run.Run at 0x2c4066eca48>
```

In [38]:

```
>>> doc.save('multipleParagraphs123.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.

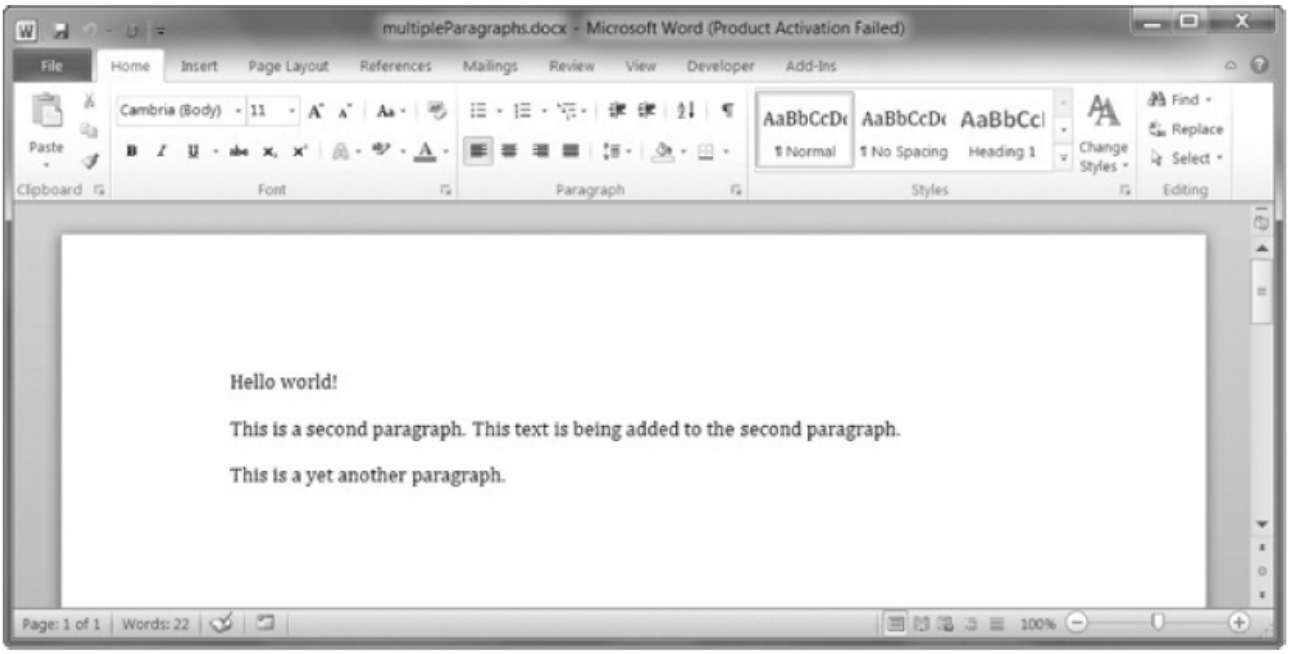

Figure 13-9. The document with multiple `Paragraph` and `Run` objects added

- 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:

In [39]:

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

Out[39]:

```
<docx.text.paragraph.Paragraph at 0x2c40663c2c8>
```

- 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:

In [41]:

```
>>> doc = docx.Document()
>>> doc.add_heading('Header 0', 0)
>>> doc.add_heading('Header 1', 1)
>>> doc.add_heading('Header 2', 2)
>>> doc.add_heading('Header 3', 3)
>>> doc.add_heading('Header 4', 4)
>>> 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.

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:

In [48]:

```
>>> doc = docx.Document()
>>> doc.add_paragraph('This is on the first page!')
```

Out[48]:

```
<docx.text.paragraph.Paragraph at 0x2c40671d548>
```

In [54]:

```
>>> doc.paragraphs[0].runs[0].add_break(docx.text.WD_BREAK.PAGE)
>>> doc.add_paragraph('This is on the second page!')
>>> doc.save('twopage.docx')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-54-ec584cabe3d9> in <module>
----> 1 doc.paragraphs[0].runs[0].add_break(docx.text.WD_BREAK.PAGE)
      2 doc.add_paragraph('This is on the second page!')
      3 doc.save('twopage.docx')

AttributeError: module 'docx.text' has no attribute 'WD_BREAK'
```

- 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

->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:

In [51]:

```
>>> doc.add_picture('tiger.jpg', width=docx.shared.Inches(1),height=docx.shared.Cm(4))
```

Out[51]:

```
<docx.shape.InlineShape at 0x2c4066f5d88>
```

In [55]:

```
doc.save('picture.docx')
```

- 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.

END Module 5

In [ ]:

```
```