1.The PyPDF2.PdfFileReader() function does not receive a string value for the PDF filename. Instead, what do you pass to the function?

Ans-: You can use PyPDF2 to extract a fair amount of useful data from any PDF. ... The first thing we do is create our own get\_info function that accepts a PDF file path as its only argument. Then we open the file in read-only binary mode. Next we pass that file handler into **PdfFileReader** and create an instance of it.

We pass it pdfFileObj to the function.

2. In what modes should the PdfFileReader() and PdfFileWriter() File objects be opened?

Ans-: **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'.

3. From a PdfFileReader object, how do you get a Page object for page 5?

Ans-:   >>> import PyPDF2  
   >>> pdfFileObj = open('meetingminutes.pdf', 'rb')  
   >>> pdfReader = PyPDF2.PdfFileReader(pdfFileObj)  
➊ >>> pdfReader.numPages  
   19  
➋ >>> pageObj = pdfReader.getPage(5)  
➌ >>> 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  '  
   >>> pdfFileObj.close()

The total number of pages in the document is stored in the numPages attribute of a PdfFileReader object ➊. 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 ➋ on a PdfFileReader object and passing it the page number of the page you’re interested in—in our case, 5.

PyPDF2 uses a zero-based index for getting pages

4. Which PdfFileReader variable keeps track of the PDF document's page count?

Ans-:The total **number** of **pages** in the **document** is stored in the numPages attribute of a **PdfFileReader** object

5. What must you do if the PDF of a PdfFileReader object is encrypted with the password swordfish before you can get Page objects from it?

Ans-:   >>> import PyPDF2  
   >>> pdfReader = PyPDF2.PdfFileReader(open('encrypted.pdf', 'rb'))  
➊ >>> pdfReader.isEncrypted  
   True  
   >>> pdfReader.getPage(0)  
➋ 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  
>>> pdfReader = PyPDF2.PdfFileReader(open('encrypted.pdf', 'rb'))  
➌ >>> pdfReader.decrypt('swordfish')  
   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 ➊. Any attempt to call a function that reads the file before it has been decrypted with the correct password will result in an error ➋.

To read an encrypted PDF, call the decrypt() function and pass the password as a string ➌. 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.

6. Can you tell me how you rotate a page?

##### Ans-: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. Ex-:

   >>> import PyPDF2  
   >>> minutesFile = open('meetingminutes.pdf', 'rb')  
   >>> pdfReader = PyPDF2.PdfFileReader(minutesFile)  
➊ >>> page = pdfReader.getPage(0)  
➋ >>> page.rotateClockwise(90)  
   {'/Contents': [IndirectObject(961, 0), IndirectObject(962, 0),  
   --snip--  
   }  
   >>> pdfWriter = PyPDF2.PdfFileWriter()  
   >>> pdfWriter.addPage(page)  
➌ >>> 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 ➊, and then we call rotateClockwise(90) on that page ➋. We write a new PDF with the rotated page and save it as rotatedPage.pdf ➌

7. For the file demo.docx, which method returns a Document object?

Ans-:**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**.

8. What is the difference between a Run object and a Paragraph object?

Ans:- 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**.

>>> **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'

❺ >>> **len(doc.paragraphs[1].runs)**

4

❻ >>> **doc.paragraphs[1].runs[0].text**

'A plain paragraph with some '

❼ >>> **doc.paragraphs[1].runs[1].text**

'bold'

❽ >>> **doc.paragraphs[1].runs[2].text**

' and some '

➒ >>> **doc.paragraphs[1].runs[3].text**

'italic'

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 ➒

9. For a Document object stored in a variable called doc, how do you get a list of paragraph objects?

Ans-:>>> **import docx**

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

❷ >>> **len(doc.paragraphs)**

7

❸ >>> **doc.paragraphs[0].text**

'Document Title'

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.

10. What object has the variables bold, underline, italic, hit, and outline?

Ans-: 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.

So, it has 5 run objects for the variables bold,underline,italic,hit and outline.

11. What is the difference between False, True, and None for the bold variable?

Ans-: 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).

|  |  |
| --- | --- |
| bold | The text appears in bold. |
|  |  |

>>> **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].bold = True**

>>> **doc.paragraphs[1].runs[3].underline = True**

>>> **doc.save('restyled.docx')**

12. How do you make a new Word document's Document object?

Ans-:>>> **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 .

13. How do you add a paragraph to a Document object stored in a variable called doc with the text 'Hello, there!'?

Ans-: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.

>>> **import docx**

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

>>> **doc.add\_paragraph('Hello,there!')**

<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')**

14. In Word documents, what integers reflect the different levels of headings?

Ans:- Calling add\_heading() adds a paragraph with one of the heading styles.

>>> **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.