**Python Assignment 12**

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

When working with PDF files in Python using the PyPDF2 library, the **PdfFileReader()** and **PdfFileWriter()** file objects should be opened in the binary mode, as follows:

# Opening PdfFileReader in binary mode  
pdf\_reader = PyPDF2.PdfFileReader(open('file.pdf', 'rb'))  
  
# Opening PdfFileWriter in binary mode  
pdf\_writer = PyPDF2.PdfFileWriter()

The **rb** mode specifies that the file should be opened in binary mode for reading, while the **wb** mode is used for binary mode writing.

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

# Importing necessary modules

import PyPDF2

# Opening the PDF file

pdf\_file = open('file.pdf', 'rb')

# Creating a PdfFileReader object

pdf\_reader = PyPDF2.PdfFileReader(pdf\_file)

# Getting Page object for page 5

page\_5 = pdf\_reader.getPage(4)

the **getPage()** method takes the page number as an argument, starting from 0 for the first page. Therefore, to get the **Page** object for page 5, we pass 4 as the argument to **getPage()**.

**3.What PdfFileReader variable stores the number of pages in the PDF document?**

# Importing necessary modules  
import PyPDF2  
  
# Opening the PDF file  
pdf\_file = open('file.pdf', 'rb')  
  
# Creating a PdfFileReader object  
pdf\_reader = PyPDF2.PdfFileReader(pdf\_file)  
  
# Getting the number of pages in the PDF document  
num\_pages = pdf\_reader.numPages  
  
print(f"The PDF document contains {num\_pages} pages.")

The **numPages** attribute returns an integer that represents the number of pages in the PDF document. In the example code above, we simply print out the number of pages to the console using an f-string

**4.If a PdfFileReader object’s PDF is encrypted with the password swordfish, what must you do before you can obtain Page objects from it?**

If a **PdfFileReader** object's PDF is encrypted with the password "swordfish", you must decrypt it first by calling the **decrypt()** method with the correct password as an argument.

Here's an example code snippet that demonstrates how to open an encrypted PDF file and obtain a **Page** object from it using PyPDF2:

# Importing necessary modules  
import PyPDF2  
# Opening the encrypted PDF file  
pdf\_file = open('file.pdf', 'rb')  
# Creating a PdfFileReader object  
pdf\_reader = PyPDF2.PdfFileReader(pdf\_file)  
# Decrypting the PDF file with the password "swordfish"  
if pdf\_reader.isEncrypted:  
 pdf\_reader.decrypt('swordfish')  
# Getting the Page object for page 1  
page\_1 = pdf\_reader.getPage(0)  
# Do something with the Page object, such as extracting text or adding annotations

we first open the PDF file in binary mode and create a **PdfFileReader** object from it. We then check if the PDF is encrypted using the **isEncrypted** attribute, and if it is, we decrypt it using the correct password "swordfish" by calling the **decrypt()** method with the password as an argument

**5. What methods do you use to rotate a page?**

To rotate a page in a PDF document using PyPDF2, you can use the **rotateClockwise()** or **rotateCounterClockwise()** methods of the **Page** object, depending on the direction of rotation you want to apply.

# Importing necessary modules

import PyPDF2

# Opening the PDF file in binary mode

pdf\_file = open('file.pdf', 'rb')

# Creating a PdfFileReader object

pdf\_reader = PyPDF2.PdfFileReader(pdf\_file)

# Getting the Page object for page 1

page\_1 = pdf\_reader.getPage(0)

# Rotating the page 90 degrees

clockwise page\_1.rotateClockwise(90)

# Creating a PdfFileWriter object

pdf\_writer = PyPDF2.PdfFileWriter()

# Adding the rotated page to the PdfFileWriter object

pdf\_writer.addPage(page\_1)

# Creating a new PDF file with the rotated page

with open('rotated\_file.pdf', 'wb') as output\_file:

pdf\_writer.write(output\_file)

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

In Microsoft Word, a **Paragraph** object represents a single paragraph of text, which is a group of one or more **Run** objects that share the same formatting. A **Run** object, on the other hand, represents a contiguous run of text with the same formatting.

Here are some key differences between a **Run** object and a **Paragraph** object:

1. Content: A **Run** object contains a single run of text, while a **Paragraph** object can contain multiple **Run** objects and may span multiple lines.
2. Formatting: A **Run** object represents a contiguous run of text with the same formatting, such as font size, style, color, or emphasis, while a **Paragraph** object represents a group of one or more **Run** objects that share the same formatting. Thus, the formatting of a **Run** object cannot change within the object, while the formatting of a **Paragraph** object can change from one **Run** object to the next.
3. Properties: A **Paragraph** object has additional properties such as alignment, indentation, spacing, borders, and shading, which apply to the entire paragraph, while a **Run** object does not have these properties.

Here is an example code snippet that demonstrates how to create a **Paragraph** object and add two **Run** objects to it using the **python-docx** library:

# Importing necessary modules  
from docx import Document  
from docx.shared import Inches  
  
# Creating a new document  
document = Document()  
  
# Creating a Paragraph object and adding two Run objects to it  
paragraph = document.add\_paragraph('This is a ')  
paragraph.add\_run('bold').bold = True  
paragraph.add\_run(' and ')  
paragraph.add\_run('italic').italic = True  
paragraph.add\_run(' text.')  
  
# Saving the document  
document.save('document.docx')

**7. How do you obtain a list of Paragraph objects for a Document object that’s stored in a variable named doc?**

# Importing necessary modules  
from docx import Document  
# Creating a new document  
document = Document()  
# Adding some paragraphs to the document  
document.add\_paragraph('This is the first paragraph.')  
document.add\_paragraph('This is the second paragraph.')  
document.add\_paragraph('This is the third paragraph.')  
# Getting a list of Paragraph objects from the document  
paragraphs = document.paragraphs  
# Printing the content of each paragraph  
for paragraph in paragraphs:  
 print(paragraph.text)  
In the code above, we first create a new **Document** object and add three **Paragraph** objects to it using the **add\_paragraph()** method.

We then obtain a list of **Paragraph** objects from the **Document** object using the **paragraphs** attribute and store it in a variable named **paragraphs**. Finally, we iterate over the list of **Paragraph** objects and print the text content of each paragraph using the **text** attribute.

**8. What type of object has bold, underline, italic, strike, and outline variables?**

In Python, the object that has bold, underline, italic, strike, and outline variables is typically a rich text formatting object or a text styling object. One commonly used Python library for working with rich text objects is the **docx** library, which is used for working with Microsoft Word documents.

In **docx**, you can access these text formatting properties using the **run** object, which represents a single run of text in a Word document. You can access the various formatting properties using attributes of the **run** object, such as **bold**, **italic**, **underline**, **strike**, and **outline**. Here is an example of how to set the bold property of a run object in **docx**:

from docx import Document  
from docx.shared import Pt  
document = Document()  
paragraph = document.add\_paragraph()  
run = paragraph.add\_run('Hello, World!')  
run.bold = True  
document.save('example.docx')  
This code creates a new Word document, adds a paragraph containing the text "Hello, World!", and sets the bold property of the run object to **True**, resulting in the text being displayed in bold in the final document.

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

In the context of text formatting, the variables **False**, **True**, and **None** for the **bold** property typically represent different states of bold formatting. Here's how they differ:

* **False**: Setting **bold** to **False** means that the text is not bold, i.e., it appears in normal font weight.
* **True**: Setting **bold** to **True** means that the text is bold, i.e., it appears in a heavier font weight than normal.
* **None**: If the **bold** property is set to **None**, it means that the bold formatting is not explicitly set. In this case, the text will inherit its bold formatting from the style of the paragraph or run that contains it.

**10. How do you create a Document object for a new Word document?**

from docx import Document  
document = Document()  
This code creates a new **Document** object that represents a blank Word document. You can then add content to the document by adding paragraphs, tables, and other elements

paragraph = document.add\_paragraph('This is a paragraph of text.')  
This code creates a new paragraph in the document and sets its text to "This is a paragraph of text." You can add more content to the document in a similar way, using the various methods provided by the **Document** object. Once you have finished creating the document, you can save it to a file using the **save** method:

document.save('example.docx')

**11. How do you add a paragraph with the text “Hello,there” to a Document object stored in a variable named doc?**

add a paragraph with the text "Hello, there!" to a **Document** object stored in a variable named **doc** using the **docx** library in Python, you can use the **add\_paragraph** method of the **Document** object.

from docx import Document  
doc = Document()  
paragraph = doc.add\_paragraph('Hello, there!')  
This code creates a new **Document** object and adds a paragraph to it with the text "Hello, there!". The **add\_paragraph** method returns a **Paragraph** object, which can assign to a variable if you want to modify the properties of the paragraph later

**12. What integers represent the levels of headings available in Word documents?**

In Microsoft Word, the levels of headings are typically represented by integers from 1 to 9, where 1 is the main heading and 9 is the lowest-level subheading. The actual number of heading levels available in a Word document may vary depending on the version of Word being used and the document's settings.

When using the **docx** library in Python to work with Word documents, you can use the **add\_heading** method of the **Document** or **Paragraph** object to add a heading with a specific level. The **level** parameter of the **add\_heading** method takes an integer from 1 to 9 to specify the level of the heading.

from docx import Document doc = Document()

doc.add\_heading('Main Heading', level=1)

doc.add\_heading('Subheading', level=2)