**Python Assignment 12**

Q1. In what modes should the PdfFileReader() and PdfFileWriter() File objects will be opened?

Ans. The PdfFileReader() and PdfFileWriter() objects in PyPDF2, which is a Python library for working with PDF files, should be opened in binary mode when working with PDF files.

To open a PDF file in binary mode with PdfFileReader(), you can use the following code:

import PyPDF2

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

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

To open a file for writing with PdfFileWriter(), you can use the following code:

import PyPDF2

pdf\_output = open('output.pdf', 'wb')

pdf\_writer = PyPDF2.PdfFileWriter()

Opening the file in binary mode is necessary because PDF files are binary files, which means that they are not written in plain text format. By opening the file in binary mode, you can read and write the raw binary data in the file.

Q2. From a PdfFileReader object, how do you get a Page object for page 5?

Ans. To get a Page object for page 5 of a PDF file using a PdfFileReader object from PyPDF2 library in Python, you can use the following code:

import PyPDF2

# Open the PDF file in binary mode

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

# Create a PdfFileReader object

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

# Get the Page object for page 5

page\_5 = pdf\_reader.getPage(4) # page numbers are zero-indexed, so index 4 is page 5

In the code above, getPage() method is used to retrieve the Page object for page 5. Note that the getPage() method uses a zero-indexed page numbering system, so to get the Page object for page 5, you pass in the index 4.

Also, it's worth noting that the PdfFileReader object's getDocumentInfo() method can be used to retrieve information about the PDF document, such as the number of pages in the document, which can be useful for iterating through all of the pages in a document.

Q3. What PdfFileReader variable stores the number of pages in the PDF document?

Ans. In PyPDF2, the PdfFileReader object has a numPages attribute that stores the total number of pages in the PDF document.

To access the numPages attribute, you can create a PdfFileReader object for the PDF file, and then simply access the attribute using dot notation:

import PyPDF2

# Open the PDF file in binary mode

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

# Create a PdfFileReader object

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

# Get the number of pages in the PDF document

num\_pages = pdf\_reader.numPages

In the code above, numPages attribute is used to get the total number of pages in the PDF document. The value of num\_pages will be an integer representing the number of pages in the PDF document.

Q4. If a PdfFileReader object’s PDF is encrypted with the password swordfish, what must you do before you can obtain Page objects from it?

Ans. If a PdfFileReader object's PDF is encrypted with the password "swordfish", you must provide the password as a string argument to the PdfFileReader constructor before you can obtain Page objects from it. Here is an example code snippet that demonstrates how to do this:

import PyPDF2

# Create a PdfFileReader object for the encrypted PDF

pdf\_reader = PyPDF2.PdfFileReader(open('encrypted.pdf', 'rb'))

# Decrypt the PDF using the password "swordfish"

pdf\_reader.decrypt('swordfish')

# Now you can obtain Page objects from the PdfFileReader object

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

In the above code, pdf\_reader.decrypt('swordfish') decrypts the PDF using the password "swordfish". Once the PDF is decrypted, you can obtain Page objects from the pdf\_reader object using the getPage() method.

Q5. What methods do you use to rotate a page?

Ans. To rotate a page using PyPDF2 library in Python, you can use the rotateClockwise() and rotateCounterClockwise() methods of a Page object. Here is an example code snippet that demonstrates how to rotate a page:

import PyPDF2

# Open the PDF file and get the first page

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

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

page = pdf\_reader.getPage(0)

# Rotate the page clockwise by 90 degrees

page.rotateClockwise(90)

# Save the rotated page to a new PDF file

pdf\_writer = PyPDF2.PdfFileWriter()

pdf\_writer.addPage(page)

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

pdf\_writer.write(output\_pdf)

In the above code, pdf\_reader.getPage(0) returns the first page of the PDF file. The rotateClockwise() method is called on the page object to rotate it clockwise by 90 degrees. The rotated page is then added to a new PdfFileWriter object, and finally written to a new PDF file called rotated.pdf.

Similarly, to rotate a page counter-clockwise, you can use the rotateCounterClockwise() method instead of rotateClockwise().

Q6. What is the difference between a Run object and a Paragraph object?

Ans. In Microsoft Word, a Paragraph object is a block of text that is separated from other blocks of text by a new line or a paragraph mark. On the other hand, a Run object is a contiguous sequence of characters within a paragraph that has the same formatting, such as font, size, color, and style.

Here are some of the differences between a Paragraph object and a Run object:

Structure: A Paragraph object is a structural element of a Word document that contains one or more Run objects. A Run object, on the other hand, is a sub-element of a Paragraph object.

Formatting: A Paragraph object can have its own formatting, such as alignment, indentation, and spacing. A Run object inherits the formatting of its containing Paragraph object, but can also have its own additional formatting, such as bold, italic, and underline.

Length: A Paragraph object can span across multiple lines and contain multiple Run objects. A Run object, however, is always a contiguous sequence of characters within a single line of a Paragraph object.

Methods: Paragraph objects and Run objects have different methods associated with them. For example, you can use the add\_run() method to add a new Run object to a Paragraph object, but you cannot use this method on a Run object.

Overall, Paragraph objects represent larger units of text in a Word document, while Run objects represent smaller units of text with specific formatting within a Paragraph.

Q7. How do you obtain a list of Paragraph objects for a Document object that’s stored in a variable named doc?

Ans. It depends on the programming language and document processing library being used. However, assuming that the variable doc contains a Document object that represents a text document, the general approach to obtain a list of Paragraph objects from it is:

Access the text content of the Document object.

Split the text content into individual paragraphs.

Create a list of Paragraph objects from the split paragraphs.

Here is an example implementation in Python using the docx library for working with Microsoft Word documents:

import docx

# Load the document

doc = docx.Document('example.docx')

# Get the text content of the document

text = '\n'.join([paragraph.text for paragraph in doc.paragraphs])

# Split the text into paragraphs

paragraphs = text.split('\n\n')

# Create a list of Paragraph objects

paragraph\_objects = [docx.text.paragraph.Paragraph(paragraph) for paragraph in paragraphs]

In this example, doc.paragraphs returns a list of Paragraph objects, but since you mentioned obtaining a list of Paragraph objects for the Document object stored in the variable doc, I assumed that doc does not already contain a list of Paragraph objects.

Q8. What type of object has bold, underline, italic, strike, and outline variables?

Ans. The object that has bold, underline, italic, strike, and outline variables may vary depending on the programming language and document processing library being used. However, in many document processing libraries, such as docx for Python, these variables are properties of a "run" object.

In Microsoft Word and in the docx library for Python, a "run" represents a contiguous run of text with the same set of character properties, such as font, size, boldness, etc. The Run object in docx library has properties such as bold, underline, italic, strike, and outline which can be used to set or get their respective attributes of the text represented by the run.

Here is an example of how to use these properties in the docx library to access and modify the formatting of a run of text:

import docx

doc = docx.Document('example.docx')

# Accessing the first paragraph in the document

paragraph = doc.paragraphs[0]

# Accessing the first run in the paragraph

run = paragraph.runs[0]

# Checking if the run is bold

if run.bold:

print('This run is bold.')

# Setting the run to be underlined

run.underline = True

# Setting the run to be italicized and have an outline

run.italic = True

run.outline = True

# Saving the modified document

doc.save('modified\_example.docx')

In this example, the bold, underline, italic, and outline properties of a run object are accessed and modified using the docx library.

Q9. What is the difference between False, True, and None for the bold variable?

Ans. In the context of document processing libraries, such as docx for Python, the bold variable for a run object can take on three different values: True, False, and None. Here is an explanation of what each value represents:

True: This means that the text represented by the run is formatted as bold. When the bold property of a run object is set to True, the text will appear bolded in the final document.

False: This means that the text represented by the run is not formatted as bold. When the bold property of a run object is set to False, the text will appear with normal weight in the final document.

None: This means that the bold property of the run object is not explicitly set. In this case, the formatting of the text represented by the run will depend on the default formatting of the document or the parent style applied to the run.

For example, if the parent style of the run is defined to be bold, then the bold property of the run object will be None, but the text will still appear bolded in the final document. Similarly, if the default formatting of the document is set to bold, then any text without an explicitly set bold property will appear bolded.

In summary, the bold property of a run object can be set to True to make the text bold, set to False to make the text normal weight, or left as None to use the default formatting of the document or the parent style applied to the run.

Q10. How do you create a Document object for a new Word document?

Ans. To create a Document object for a new Word document in Python using the python-docx library, you can follow these general steps:

Import the Document class from the docx module.

Create a new instance of the Document class.

Here's an example of how to create a new Word document using python-docx:

from docx import Document

# Create a new Document object

doc = Document()

# Add content to the document (optional)

doc.add\_paragraph('This is my first paragraph.')

# Save the document to a file

doc.save('my\_document.docx')

In this example, the Document class is imported from the docx module, and a new instance of the Document class is created by calling the Document() constructor with no arguments. You can then add content to the document, such as paragraphs, tables, images, and so on, using the appropriate methods provided by the Document object. Finally, you can save the document to a file using the save() method and providing the desired filename as an argument.

Q11. How do you add a paragraph with the text 'Hello, there!' to a Document object stored in a variable named doc?

Ans. To add a paragraph with the text "Hello, there!" to a Document object stored in a variable named doc using python-docx in Python, you can use the add\_paragraph() method of the Document object. Here's an example:

from docx import Document

# Create a new Document object

doc = Document()

# Add a new paragraph to the document

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

# Save the document to a file

doc.save('my\_document.docx')

In this example, the add\_paragraph() method is used to add a new paragraph with the text "Hello, there!" to the Document object stored in the doc variable. Finally, the save() method is called on the Document object to save the document to a file.

Q12. What integers represent the levels of headings available in Word documents?

Ans. In Microsoft Word documents, headings are organized into different levels, with each level denoting a different level of importance or hierarchy. In python-docx, the levels of headings are represented by integers, where 1 is the highest level of heading and 9 is the lowest. Here's a list of the integers that represent the levels of headings in Word documents:

Heading 1: Level 1

Heading 2: Level 2

Heading 3: Level 3

Heading 4: Level 4

Heading 5: Level 5

Heading 6: Level 6

Heading 7: Level 7

Heading 8: Level 8

Heading 9: Level 9

To add a new heading to a Document object using python-docx, you can use the add\_heading() method and provide the appropriate level integer as an argument. Here's an example:

from docx import Document

# Create a new Document object

doc = Document()

# Add a new heading to the document

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

# Save the document to a file

doc.save('my\_document.docx')

In this example, a new heading at level 1 is added to the Document object using the add\_heading() method, with the text "Heading 1" as the heading content. The level argument is set to 1 to specify that this is a level 1 heading.