**Python Assignment 12**

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

When using the PdfFileReader() and PdfFileWriter() classes from the PyPDF2 library to work with PDF files in Python, you typically open the PDF files in different modes for reading and writing operations:

PdfFileReader() - Reading Mode:

When creating a PdfFileReader() object to read an existing PDF file, you should open the file in binary reading mode ('rb'). This is because PDF files are binary files, and you need to read them in binary mode to ensure that the file's internal structure is preserved.

PdfFileWriter() - Writing Mode:

When creating a PdfFileWriter() object to write or modify a PDF file, you should also open the file in binary writing mode ('wb'). This is important to ensure that the binary structure of the PDF file is maintained.

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

To get a Page object for page 5 from a PdfFileReader object in the PyPDF2 library, you can use the getPage() method and provide the page index (0-based) as an argument. Here's how you can do it:

from PyPDF2 import PdfFileReader

# Open the PDF file in binary reading mode

with open('input.pdf', 'rb') as pdf\_file:

pdf\_reader = PdfFileReader(pdf\_file)

# Get a Page object for page 5 (index 4)

page\_number = 4 # Page numbers are 0-based

page = pdf\_reader.getPage(page\_number)

# Now you can work with the Page object

# For example, you can extract text from the page:

page\_text = page.extractText()

print(page\_text)

In this example, the getPage() method is used to retrieve the Page object for page 5 (index 4). Remember that page numbers in PdfFileReader are 0-based, so page 5 corresponds to index 4. Once you have the Page object, you can perform various operations on it, such as extracting text, working with annotations, and more.

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

The PdfFileReader variable that stores the number of pages in the PDF document is called numPages. This variable is an attribute of the PdfFileReader class in the PyPDF2 library and represents the total number of pages in the PDF file.

You can access the numPages attribute of a PdfFileReader object to retrieve the total number of pages in the PDF document. Here's an example:

from PyPDF2 import PdfFileReader

# Open the PDF file in binary reading mode

with open('input.pdf', 'rb') as pdf\_file:

pdf\_reader = PdfFileReader(pdf\_file)

# Get the number of pages in the PDF document

total\_pages = pdf\_reader.numPages

print(f'Total number of pages: {total\_pages}')

In this example, pdf\_reader.numPages returns the total number of pages in the PDF document, and it's stored in the total\_pages variable for printing.

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?

Before you can obtain Page objects from a PdfFileReader object that represents an encrypted PDF with the password "swordfish," you must decrypt the PDF by setting the appropriate decryption password using the decrypt() method.

Here's how you would do it:

from PyPDF2 import PdfFileReader

# Open the encrypted PDF file in binary reading mode

with open('encrypted.pdf', 'rb') as pdf\_file:

pdf\_reader = PdfFileReader(pdf\_file)

# Decrypt the PDF using the password

password = 'swordfish'

pdf\_reader.decrypt(password)

# Now you can access Page objects and other operations on the PDF

num\_pages = pdf\_reader.numPages

print(f'Total number of pages: {num\_pages}')

# Access a Page object (for example, the first page)

page = pdf\_reader.getPage(0)

# Now you can work with the Page object

page\_text = page.extractText()

print(page\_text)

In this example, the pdf\_reader.decrypt(password) line decrypts the encrypted PDF using the provided password. After decryption, you can access Page objects and perform various operations on the PDF, such as extracting text or manipulating its content.

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

To rotate a page in a PDF using the PyPDF2 library, you can use the rotateClockwise() and rotateCounterClockwise() methods provided by the Page object. These methods allow you to rotate the page clockwise or counterclockwise by 90 degrees. Here's how you can use these methods:

from PyPDF2 import PdfFileReader, PdfFileWriter

# Open the PDF file in binary reading mode

with open('input.pdf', 'rb') as pdf\_file:

pdf\_reader = PdfFileReader(pdf\_file)

# Create a new PDF writer

pdf\_writer = PdfFileWriter()

# Rotate pages and add them to the new PDF

for page\_number in range(pdf\_reader.numPages):

page = pdf\_reader.getPage(page\_number)

# Rotate the page clockwise by 90 degrees

rotated\_page = page.rotateClockwise(90)

# Add the rotated page to the new PDF

pdf\_writer.addPage(rotated\_page)

# Write the rotated pages to a new PDF file

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

pdf\_writer.write(output\_pdf)

In this example, the loop iterates through each page of the input PDF, rotates the page clockwise by 90 degrees using the rotateClockwise() method, and adds the rotated page to a new PDF using the PdfFileWriter object. After processing all pages, the new PDF with rotated pages is saved as 'rotated.pdf'.

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

In the context of working with Word documents using the python-docx library, both Run objects and Paragraph objects are used to manipulate the content of the document, but they serve different purposes:

Run Object:

A Run object represents a contiguous run of text within a paragraph that has the same formatting. It can include variations in font size, color, boldness, italics, and more. Essentially, a Run is a sequence of characters within a paragraph that share the same formatting properties. You can think of a Run as a portion of text that has consistent style.

Example of using a Run object:

from docx import Document

doc = Document()

paragraph = doc.add\_paragraph()

run = paragraph.add\_run('This is bold and italic text.')

run.bold = True

run.italic = True

Paragraph Object:

A Paragraph object represents a complete paragraph of text within the document. It can contain one or more Run objects, allowing you to apply different formatting styles to different parts of the paragraph. A paragraph can also have its own alignment, spacing, and indentation settings.

Example of using a Paragraph object:

from docx import Document

doc = Document()

paragraph = doc.add\_paragraph()

run1 = paragraph.add\_run('This is the first part of the paragraph. ')

run2 = paragraph.add\_run('This is the second part of the paragraph.')

run1.bold = True

run2.italic = True

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

named doc?

To obtain a list of Paragraph objects from a Document object stored in a variable named doc using the python-docx library, you can use the paragraphs attribute of the Document object. The paragraphs attribute holds a list of all the Paragraph objects in the document. Here's how you can do it:

from docx import Document

# Load a Word document into a Document object

doc = Document('my\_document.docx')

# Get a list of Paragraph objects from the Document object

paragraphs\_list = doc.paragraphs

# Now you can iterate over the list of Paragraph objects

for paragraph in paragraphs\_list:

print(paragraph.text)

In this example, the paragraphs attribute of the doc object provides a list of all the Paragraph objects in the document. You can then iterate through this list and access the text content of each paragraph using the text attribute.

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

The Run object in the python-docx library has the following variables to control text formatting:

bold: Controls whether the text is bold or not.

underline: Controls the underline style of the text.

italic: Controls whether the text is italic or not.

strike: Controls whether the text has a strikethrough style.

outline: Controls whether the text is outlined or not.

A Run object represents a contiguous run of text within a paragraph that shares the same formatting. You can use these variables to apply various formatting styles to the text within a Run object.

Here's an example of using these variables:

from docx import Document

doc = Document()

paragraph = doc.add\_paragraph()

run = paragraph.add\_run('This is some text with formatting.')

run.bold = True

run.underline = True

run.italic = True

run.strike = True

run.outline = True

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

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

In the context of the python-docx library, the bold variable of a Run object is used to control the bold formatting of the text. Here's how False, True, and None affect the bold variable:

False:

Setting bold to False (or not setting it) means that the text within the Run object will not be bold. The text will be displayed in the regular font weight.

run = paragraph.add\_run('This is normal text.')

run.bold = False # Not necessary since False is the default

True:

Setting bold to True means that the text within the Run object will be displayed in bold font weight.

run = paragraph.add\_run('This is bold text.')

run.bold = True

None:

If the bold variable is set to None, the text within the Run object will inherit the formatting from the surrounding context (e.g., paragraph or document level). If the surrounding context specifies bold formatting, the text will be displayed in bold. If not, the text will be displayed in the regular font weight.

run = paragraph.add\_run('This inherits bold formatting.')

run.bold = None # Inherit bold formatting from the surrounding context

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

To create a Document object for a new Word document using the python-docx library, you can simply create an instance of the Document class. Here's how you can do it:

from docx import Document

# Create a new Document object

doc = Document()

# Add content to the document

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

doc.add\_paragraph('This is a new Word document created using python-docx.')

# Save the document to a file

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

In this example, a new Document object is created using Document(). You can then use the various methods provided by the Document class to add content to the document, such as headings, paragraphs, tables, images, and more. Finally, the save() method is used to save the document to a file with the specified filename (in this case, 'new\_document.docx').

11. How do you add a paragraph with the text 'Hello, there!' to a Document object stored in a

variable named doc?

To add a paragraph with the text 'Hello, there!' to a Document object stored in a variable named doc using the python-docx library, you can use the add\_paragraph() method. Here's how you can do it:

from docx import Document

# Create a new Document object

doc = Document()

# Add a paragraph with the text 'Hello, there!'

text = 'Hello, there!'

doc.add\_paragraph(text)

# Save the document to a file

doc.save('document\_with\_paragraph.docx')

In this example, the add\_paragraph() method is used to add a new paragraph with the specified text ('Hello, there!') to the Document object stored in the doc variable. The save() method is then used to save the document to a file ('document\_with\_paragraph.docx').

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

In Word documents, the levels of headings are represented by integers from 1 to 9. These integers correspond to the hierarchical structure of headings, with level 1 being the highest and level 9 being the lowest. Headings are commonly used to organize and structure the content of a document, with each level indicating a different level of importance or hierarchy.

For example, the following integers represent the levels of headings in Word documents:

Level 1: Heading 1

Level 2: Heading 2

Level 3: Heading 3

Level 4: Heading 4

Level 5: Heading 5

Level 6: Heading 6

Level 7: Heading 7

Level 8: Heading 8

Level 9: Heading 9