Python Assignment 12

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

Ans:-- In Python's PyPDF2 library, which is commonly used for working with PDF files, the `PdfFileReader()` and `PdfFileWriter()` objects are typically created with file objects that are opened in the following modes:

1. PdfFileReader() :

- The `PdfFileReader()` object is used for reading and extracting information from existing PDF files.

- You should open the PDF file in binary read mode (`'rb'`) when creating a `PdfFileReader` object. This mode indicates that you are reading the PDF file as binary data.

```python

from PyPDF2 import PdfFileReader

# Open an existing PDF file for reading

pdf\_reader = PdfFileReader(open('existing\_file.pdf', 'rb'))

```

2. PdfFileWriter() :

- The `PdfFileWriter()` object is used for creating a new PDF file or modifying an existing one by adding content.

- You should open the PDF file in binary write mode (`'wb'`) when creating a `PdfFileWriter` object. This mode indicates that you are writing binary data to create or modify the PDF file.

```python

from PyPDF2 import PdfFileWriter

# Create a new PDF file or modify an existing one for writing

pdf\_writer = PdfFileWriter()

output\_pdf = open('output\_file.pdf', 'wb')

```

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

Ans:-- To get a `Page` object for page 5 from a `PdfFileReader` object in Python's PyPDF2 library, you can use the `getPage()` method and specify the page number as an argument. Here's how you can do it:

```python

from PyPDF2 import PdfFileReader

# Open an existing PDF file for reading

pdf\_reader = PdfFileReader(open('existing\_file.pdf', 'rb'))

# Get a Page object for page 5 (page numbering starts from 0)

page\_number = 4 # Page 5 is index 4 (0-based index)

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

# Now 'page' contains the Page object for page 5

```

In this example, we first open an existing PDF file in read mode (`'rb'`). Then, we use the `getPage()` method to retrieve the `Page` object for page 5. Note that page numbering in PyPDF2 starts from 0, so page 5 corresponds to index 4 in the `PdfFileReader` object.

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

Ans:-- In Python's PyPDF2 library, the number of pages in a PDF document can be obtained using the `numPages` attribute of a `PdfFileReader` object. This attribute stores the total number of pages in the PDF document.

Here's how you can access the `numPages` attribute:

```python

from PyPDF2 import PdfFileReader

# Open an existing PDF file for reading

pdf\_reader = PdfFileReader(open('existing\_file.pdf', 'rb'))

# Get the number of pages in the PDF document

number\_of\_pages = pdf\_reader.numPages

# 'number\_of\_pages' now contains the total number of pages in the PDF document

```

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?

Ans:-- If a `PdfFileReader` object's PDF is encrypted with the password "swordfish," you must provide the password to decrypt the PDF before you can obtain `Page` objects from it. To do this, you can use the `decrypt()` method of the `PdfFileReader` object. Here's how you can use it:

```python

from PyPDF2 import PdfFileReader

# Open an existing encrypted PDF file for reading

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

# Provide the password to decrypt the PDF

password = 'swordfish'

pdf\_reader.decrypt(password)

# Now, you can obtain Page objects from the decrypted PDF

```

In this code:

1. We open the encrypted PDF file in read mode (`'rb'`).

2. We provide the password "swordfish" to the `decrypt()` method of the `PdfFileReader` object. This method attempts to decrypt the PDF using the provided password.

3. If the password is correct, the PDF is decrypted, and you can proceed to obtain `Page` objects from it using the usual methods like `getPage()`.

It's important to ensure that you have the correct password for the encrypted PDF. Providing the wrong password will result in a failed decryption attempt, and you won't be able to access the contents of the PDF.

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

Ans:-- To rotate a page in a PDF using Python's PyPDF2 library, you can use the following methods:

1. rotateClockwise(degrees) :

- This method rotates a page clockwise by the specified number of degrees (90, 180, or 270).

- You need to call this method on a `Page` object retrieved from a `PdfFileReader` or `PdfFileWriter` object.

```python

from PyPDF2 import PdfFileReader, PdfFileWriter

# Open an existing PDF file for reading

pdf\_reader = PdfFileReader(open('existing\_file.pdf', 'rb'))

# Get a Page object (e.g., page 1)

page\_number = 0 # Page numbering starts from 0

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

# Rotate the page clockwise by 90 degrees

page.rotateClockwise(90)

# Save the rotated page to a new PDF file or PdfFileWriter object

pdf\_writer = PdfFileWriter()

pdf\_writer.addPage(page)

# Save the rotated page to a new PDF file

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

pdf\_writer.write(output\_pdf)

```

2. rotateCounterClockwise(degrees) :

- This method rotates a page counterclockwise by the specified number of degrees (90, 180, or 270).

- Similar to `rotateClockwise()`, you need to call this method on a `Page` object.

```python

from PyPDF2 import PdfFileReader, PdfFileWriter

# Open an existing PDF file for reading

pdf\_reader = PdfFileReader(open('existing\_file.pdf', 'rb'))

# Get a Page object (e.g., page 1)

page\_number = 0 # Page numbering starts from 0

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

# Rotate the page counterclockwise by 90 degrees

page.rotateCounterClockwise(90)

# Save the rotated page to a new PDF file or PdfFileWriter object

pdf\_writer = PdfFileWriter()

pdf\_writer.addPage(page)

# Save the rotated page to a new PDF file

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

pdf\_writer.write(output\_pdf)

```

You can choose either `rotateClockwise()` or `rotateCounterClockwise()` depending on the direction and degrees by which you want to rotate the page. After rotation, you can save the modified page to a new PDF file or a `PdfFileWriter` object as shown in the examples above.

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

Ans:-- In the context of document processing, particularly in libraries like Python's `python-docx` (for working with Microsoft Word documents), a Run object and a Paragraph object are fundamental components that represent different parts of a document. Here's the difference between the two:

1. Paragraph Object :

- A Paragraph object represents a single paragraph of text within a document.

- It typically contains one or more Run objects.

- A paragraph is a block of text with a specific style, alignment, and formatting.

- You can think of a paragraph as a container for text that is formatted consistently within a block.

Example (Python-docx):

```python

from docx import Document

doc = Document()

paragraph = doc.add\_paragraph("This is a paragraph.")

```

2. Run Object :

- A Run object represents a contiguous run of text within a paragraph.

- It can have its own specific formatting, such as font style, size, color, and more.

- A paragraph can contain multiple runs with different formatting properties.

- You can think of a run as a segment of text within a paragraph that can have different formatting from the rest of the paragraph.

Example (Python-docx):

```python

from docx import Document

doc = Document()

paragraph = doc.add\_paragraph()

run1 = paragraph.add\_run("This is a ")

run2 = paragraph.add\_run("bold")

run2.bold = True # Formatting applied to this run

run3 = paragraph.add\_run(" text.")

```

In the example above, `paragraph` is a Paragraph object, and it contains three Run objects: "This is a," "bold" (with bold formatting applied), and "text."

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

Ans:--- To obtain a list of Paragraph objects for a Document object stored in a variable named `doc`, you can use the `paragraphs` attribute of the `Document` object. This attribute contains a list of all the paragraphs within the document. Here's how you can access it:

```python

from docx import Document

# Open a Word document (replace 'your\_document.docx' with the actual document filename)

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

# Get a list of Paragraph objects from the Document object

paragraphs\_list = doc.paragraphs

# Now, 'paragraphs\_list' contains all the Paragraph objects in the document

```

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

Ans:-- The attributes `bold`, `underline`, `italic`, `strike`, and `outline` are typically associated with a Run object in libraries like `python-docx` (for working with Microsoft Word documents) or similar document processing libraries. These attributes allow you to apply various formatting styles to the text within a Run object. Here's what they represent:

1. bold : This attribute is used to make the text within a Run object bold. You can set it to `True` to apply bold formatting or `False` to remove it.

2. underline : This attribute is used to add an underline to the text within a Run object. You can set it to different values to apply various underline styles, or you can set it to `True` or `False` to turn underline on or off.

3. italic : This attribute is used to make the text within a Run object italic. You can set it to `True` to apply italic formatting or `False` to remove it.

4. strike : This attribute is used to apply strikethrough formatting to the text within a Run object. You can set it to `True` to apply strikethrough or `False` to remove it.

5. outline : This attribute is used to apply outline formatting to the text within a Run object. You can set it to `True` to apply outline formatting or `False` to remove it.

Here's an example of how these attributes are used with a Run object in Python-docx:

```python

from docx import Document

doc = Document()

paragraph = doc.add\_paragraph()

# Create a Run object and apply formatting

run = paragraph.add\_run("Formatted Text")

run.bold = True

run.underline = True

run.italic = True

run.strike = True

run.outline = True

```

In this example, the `run` object is a Run object, and we apply various formatting styles to the text within it using the attributes mentioned.

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

Ans::--- In the context of working with document processing libraries like `python-docx` (for Microsoft Word documents), the values `False`, `True`, and `None` for the `bold` variable represent different states or settings for applying bold formatting to text within a Run object:

1. False :

- When the `bold` variable is set to `False`, it means that the text within the Run object is not formatted as bold.

- Text with `bold` set to `False` appears in the regular (non-bold) style.

2. True :

- When the `bold` variable is set to `True`, it means that the text within the Run object is formatted as bold.

- Text with `bold` set to `True` appears in a bold style.

3. None :

- Setting the `bold` variable to `None` (or leaving it unset) means that no specific instruction is given for bold formatting.

- In this case, the formatting of the text within the Run object is determined by the style of the paragraph or the character style inherited from the surrounding context in the document.

- If the surrounding style specifies bold, the text will be bold; if not, it won't be bold.

Here's a code example to illustrate these settings:

```python

from docx import Document

doc = Document()

paragraph = doc.add\_paragraph()

run = paragraph.add\_run("Text with different bold settings")

# Set bold to False (not bold)

run.bold = False

# Set bold to True (bold)

run.bold = True

# Leave bold unset (None) to inherit the surrounding style

# This depends on the paragraph or character style in the document

run.bold = None # or simply don't set it

# Save the document with these text variations

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

```

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

Ans:-- To create a `Document` object for a new Word document using the `python-docx` library (commonly used for working with Microsoft Word documents), you can follow these steps:

1. Import the `Document` class from the `docx` module.

2. Create a new `Document` object .

3. Add content to the `Document` object as needed. This can include paragraphs, runs, tables, images, and other elements.

4. Save the `Document` object to a file to create the actual Word document.

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

```python

from docx import Document

# Create a new Document object

doc = Document()

# Add content to the document

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

# Add a paragraph

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

# Add another paragraph

doc.add\_paragraph('This is the second paragraph.')

# Save the document to a file

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

```

In this example:

- We import the `Document` class from the `docx` module.

- We create a new `Document` object called `doc`.

- We add content to the document using the `add\_heading()` and `add\_paragraph()` methods.

- Finally, we save the `Document` object to a file named 'my\_document.docx' using the `save()` method.

11. How do you add a paragraph with the text &#39;Hello, there!&#39; 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 the `python-docx` library, you can use the `add\_paragraph()` method. Here's how you can do it:

```python

from docx import Document

# Assuming you already have a Document object stored in the variable 'doc'

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

paragraph\_text = 'Hello, there!'

doc.add\_paragraph(paragraph\_text)

# Save the document (if needed)

# doc.save('your\_document.docx')

```

In this code:

1. We assume that you have already created a `Document` object and stored it in the variable `doc`.

2. We use the `add\_paragraph()` method to add a new paragraph to the `Document` object and specify the text 'Hello, there!' as the content of the paragraph.

3. If you want to save the modified document to a file, you can use the `save()` method, as shown in the commented line. Replace `'your\_document.docx'` with the desired filename.

After running this code, the 'Hello, there!' paragraph will be added to the `Document` object stored in the variable `doc`.

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

Ans:- In Word documents, heading levels are typically represented by integers ranging from 1 to 9. These integers correspond to different levels of headings or headings styles that you can apply to your document's text. Here's a common mapping of heading levels to integers:

1. Heading 1 : Represented by the integer 1. It is often used for the main or top-level heading in a document.

2. Heading 2 : Represented by the integer 2. It is used for subheadings or sections within the main heading.

3. Heading 3 : Represented by the integer 3. It is used for subsections within Heading 2, and so on.

4. Heading 4 : Represented by the integer 4.

5. Heading 5 : Represented by the integer 5.

6. Heading 6 : Represented by the integer 6.

7. Heading 7 : Represented by the integer 7.

8. Heading 8 : Represented by the integer 8.

9. Heading 9 : Represented by the integer 9.

These heading levels help structure and organize the content of a Word document, making it easier to create a table of contents, navigate the document, and apply consistent formatting to different sections. The choice of heading level depends on the document's hierarchy and the desired formatting and styling.