**Assignment 12**

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

When using the PyPDF2 library in Python, the **PdfFileReader()** and **PdfFileWriter()** functions require that their file objects be opened in different modes.

* For **PdfFileReader()**, you need to open the PDF file in binary mode, using **'rb'** (read binary) as the mode. This is because a PDF file is a binary file, meaning that it contains data in the form of 0's and 1's. Opening a file in binary mode allows you to read this data correctly.
* For **PdfFileWriter()**, you need to open the PDF file in binary mode as well, but you also need to be able to write to it. Therefore, you should use **'wb'** (write binary) as the mode.

Here's an example of how you might use these modes in practice:

pythonCopy code

from PyPDF2 import PdfFileReader, PdfFileWriter # Open the input PDF file in binary mode input\_file = open('input\_file.pdf', 'rb') # Create a PdfFileReader object to read the input file pdf\_reader = PdfFileReader(input\_file) # Open the output PDF file in binary mode output\_file = open('output\_file.pdf', 'wb') # Create a PdfFileWriter object to write to the output file pdf\_writer = PdfFileWriter() # Do some stuff with the PdfFileReader and PdfFileWriter objects # Close the input and output files input\_file.close() output\_file.close()

Note that you should always remember to close the files when you are done with them, to avoid any potential issues with file locking or other errors.

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

You can use the **PdfFileReader** object's **getPage()** method to get a **Page** object for a specific page number. To get a **Page** object for page 5, you would use the following code:

pythonCopy code

from PyPDF2 import PdfFileReader # Open the PDF file in binary mode pdf\_file = open('file.pdf', 'rb') # Create a PdfFileReader object pdf\_reader = PdfFileReader(pdf\_file) # Get a Page object for page 5 page\_five = pdf\_reader.getPage(4) # Note that the page numbering starts from 0, so page 5 is actually at index 4 # Close the PDF file pdf\_file.close()

In this example, the **getPage()** method takes an argument that represents the page number you want to retrieve. Note that the page numbering starts from 0, so if you want to get page 5, you need to pass the argument **4** to **getPage()**. The resulting **Page** object is stored in the **page\_five** variable.

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

To get the number of pages in a PDF document using PyPDF2 in Python, you can use the **getNumPages()** method of a **PdfFileReader** object. This method returns the total number of pages in the PDF file. You can store this value in a variable for later use.

Here's an example:

pythonCopy code

from PyPDF2 import PdfFileReader # Open the PDF file in binary mode pdf\_file = open('file.pdf', 'rb') # Create a PdfFileReader object pdf\_reader = PdfFileReader(pdf\_file) # Get the number of pages in the PDF document num\_pages = pdf\_reader.getNumPages() # Close the PDF file pdf\_file.close() # Print the number of pages print(f'The PDF document has {num\_pages} pages.')

In this example, the **getNumPages()** method returns the total number of pages in the PDF document, which is then stored in the **num\_pages** variable. You can then use this variable for any further processing or output.

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 a password, you must decrypt it by calling the **decrypt()** method and passing the correct password as a string argument before you can obtain **Page** objects from it. If you do not provide the correct password, you will receive a **PyPDF2.utils.PdfReadError** error when you try to read the PDF file.

Here is an example of how to decrypt a password-protected PDF file with PyPDF2 in Python:

pythonCopy code

from PyPDF2 import PdfFileReader # Open the PDF file in binary mode pdf\_file = open('file.pdf', 'rb') # Create a PdfFileReader object pdf\_reader = PdfFileReader(pdf\_file) # If the PDF is encrypted, decrypt it with the correct password if pdf\_reader.isEncrypted: pdf\_reader.decrypt('swordfish') # Get a Page object for page 1 page\_one = pdf\_reader.getPage(0) # Close the PDF file pdf\_file.close()

In this example, the **isEncrypted** attribute of the **PdfFileReader** object is checked to determine whether the PDF file is encrypted. If it is encrypted, the **decrypt()** method is called on the **PdfFileReader** object with the correct password (**'swordfish'** in this case) to decrypt the PDF file. After decryption, you can retrieve a **Page** object from the PDF file using the **getPage()** method. Note that page numbers start from 0, so **getPage(0)** will return a **Page** object for the first page of the PDF file.

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

To rotate a page in a PDF document using PyPDF2 in Python, you can use the **rotateClockwise()** and **rotateCounterClockwise()** methods of a **Page** object.

The **rotateClockwise()** method rotates the page clockwise by 90 degrees, and the **rotateCounterClockwise()** method rotates the page counterclockwise by 90 degrees.

Here is an example of how to rotate a page clockwise by 90 degrees using PyPDF2 in Python:

pythonCopy code

from PyPDF2 import PdfFileReader, PdfFileWriter # Open the input PDF file in binary mode pdf\_file = open('input\_file.pdf', 'rb') # Create a PdfFileReader object for the input file pdf\_reader = PdfFileReader(pdf\_file) # Get the first page of the input PDF file page = pdf\_reader.getPage(0) # Rotate the page clockwise by 90 degrees page.rotateClockwise(90) # Create a PdfFileWriter object for the output file pdf\_writer = PdfFileWriter() # Add the rotated page to the output PDF file pdf\_writer.addPage(page) # Open the output PDF file in binary mode output\_file = open('output\_file.pdf', 'wb') # Write the output PDF file pdf\_writer.write(output\_file) # Close the input and output PDF files pdf\_file.close() output\_file.close()

In this example, the first page of the input PDF file is retrieved using the **getPage()** method, and then rotated clockwise by 90 degrees using the **rotateClockwise()** method. The rotated page is then added to a new **PdfFileWriter** object, and the output PDF file is created by writing the contents of the **PdfFileWriter** object to a new file.

Note that you can also specify the rotation angle in degrees using the **rotateClockwise()** and **rotateCounterClockwise()** methods, for example **page.rotateClockwise(180)** will rotate the page by 180 degrees.

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

In Microsoft Word documents, a **Paragraph** object represents a paragraph of text, while a **Run** object represents a contiguous run of text with the same font and formatting properties within a paragraph.

Here are some of the key differences between **Run** and **Paragraph** objects:

* A **Paragraph** object represents a full paragraph of text, including any line breaks or paragraph spacing. A **Run** object represents a continuous sequence of characters within a paragraph that share the same formatting properties.
* A **Paragraph** object can contain multiple **Run** objects, each with its own formatting properties. For example, a paragraph may contain a bolded word or phrase, followed by plain text, followed by an italicized word or phrase.
* **Paragraph** objects are usually used for formatting larger blocks of text, such as headings or body paragraphs. **Run** objects are used for formatting smaller sections of text, such as specific words or phrases within a paragraph.
* **Paragraph** objects have properties that affect the entire paragraph, such as alignment, indentation, and line spacing. **Run** objects have properties that affect only the specific text they contain, such as font, size, color, and style.

In summary, **Paragraph** objects represent a full paragraph of text, while **Run** objects represent a continuous sequence of characters within a paragraph that share the same formatting properties.

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 for a **Document** object that’s stored in a variable named **doc** in Python-docx, you can use the **paragraphs** property of the **Document** object.

Here is an example code snippet:

pythonCopy code

import docx # Load the document doc = docx.Document('example.docx') # Get the list of paragraphs paragraphs = doc.paragraphs # Print each paragraph for paragraph in paragraphs: print(paragraph.text)

In this example, the **Document** object is loaded from a file named 'example.docx'. The **paragraphs** property of the **Document** object is then used to obtain a list of **Paragraph** objects, which is stored in the **paragraphs** variable. Finally, a **for** loop is used to print the text of each paragraph in the document.

Note that the **paragraphs** property returns a list of **Paragraph** objects, so you can iterate through the list and access properties and methods of each **Paragraph** object as needed.

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

In Python-docx, the **Font** object has the **bold**, **underline**, **italic**, **strike**, and **outline** variables, which represent various font formatting options.

The **Font** object is used to specify the font and formatting properties for text in a **Paragraph** or **Run** object in a Microsoft Word document. You can create a **Font** object and set its properties, and then apply it to a **Paragraph** or **Run** object using the **add\_run()** or **add\_paragraph()** method of the **Document** object.

Here's an example of how to create a **Font** object and set its formatting properties:

pythonCopy code

from docx import Document from docx.shared import Pt # Create a Document object doc = Document() # Create a Font object with bold, italic, and underline formatting font = doc.styles['Normal'].font font.bold = True font.italic = True font.underline = True # Add a paragraph with the formatted text paragraph = doc.add\_paragraph('This text is bold, italic, and underlined.') # Set the font size of the paragraph paragraph.style.font.size = Pt(12) # Save the document doc.save('example.docx')

In this example, a **Font** object is created and its **bold**, **italic**, and **underline** properties are set to **True**. The **add\_paragraph()** method of the **Document** object is then used to add a new paragraph with the formatted text. The font size of the paragraph is also set to 12 points using the **size** property of the paragraph's font. Finally, the document is saved to a file named 'example.docx'.

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

In Python-docx, the **bold** variable of a **Font** object can be set to **True**, **False**, or **None**, each with a different meaning:

* **True**: The text is bold.
* **False**: The text is not bold.
* **None**: The text inherits its bold formatting from the parent style.

Here's an example that demonstrates the differences between these values:

pythonCopy code

from docx import Document # Create a Document object doc = Document() # Add a paragraph with bold, normal, and inherited bold formatting paragraph1 = doc.add\_paragraph('This text is bold.', style='Heading1') paragraph1.runs[0].font.bold = True paragraph2 = doc.add\_paragraph('This text is normal.', style='Normal') paragraph2.runs[0].font.bold = False paragraph3 = doc.add\_paragraph('This text inherits its bold formatting.', style='Heading2') paragraph3.style.font.bold = True # Save the document doc.save('example.docx')

In this example, three paragraphs are added with different bold formatting. The first paragraph is given the 'Heading1' style, which has bold formatting by default. The **bold** property of the first **Run** object in the paragraph is set to **True** to make the text even bolder.

The second paragraph is given the 'Normal' style, which does not have bold formatting by default. The **bold** property of the first **Run** object in the paragraph is set to **False** to remove any bold formatting.

The third paragraph is given the 'Heading2' style, which also has bold formatting by default. However, instead of setting the **bold** property of the **Run** object, the **bold** property of the paragraph's font is set to **True**. This causes all runs within the paragraph to inherit their bold formatting from the parent style.

In summary, **True** sets the text to bold, **False** sets the text to normal, and **None** causes the text to inherit its bold formatting from the parent style.

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

In Python-docx, you can create a **Document** object to represent a new Word document using the following code:

pythonCopy code

from docx import Document # Create a new Document object doc = Document()

This code creates a new **Document** object and assigns it to the variable **doc**. The **Document** object represents a new, empty Word document that can be filled with text, paragraphs, tables, images, and other types of content using various methods provided by the **Document** object.

Once you have created a **Document** object, you can save it to a file using the **save()** method of the object. For example, to save the document to a file named 'example.docx', you can use the following code:

arduinoCopy code

doc.save('example.docx')

This will create a new Word document file in the current working directory with the name 'example.docx', and write the contents of the **Document** object to the file.

11. How do you add a paragraph with the text &#39;Hello, there!&#39; to a Document object stored in a variable named doc?

In Python-docx, you can add a new paragraph with the text 'Hello, there!' to a **Document** object stored in a variable named **doc** using the following code:

pythonCopy code

from docx import Document # Create a new Document object doc = Document() # Add a new paragraph to the document doc.add\_paragraph('Hello, there!')

This code first creates a new **Document** object and assigns it to the variable **doc**. Then, the **add\_paragraph()** method of the **Document** object is called with the text 'Hello, there!' as its argument. This creates a new paragraph object with the specified text and adds it to the **Document** object.

Note that the **add\_paragraph()** method returns the new **Paragraph** object that was created, so you can assign it to a variable if you need to modify its properties or add additional content to it. For example:

pythonCopy code

from docx import Document # Create a new Document object doc = Document() # Add a new paragraph to the document and assign it to a variable paragraph = doc.add\_paragraph('Hello, there!') # Add a run with italic formatting to the paragraph run = paragraph.add\_run('This text is italic.') run.italic = True

This code creates a new paragraph object and assigns it to the variable **paragraph**, then adds a new run with the text 'This text is italic.' to the paragraph. The **italic** property of the run is set to **True** to make the text italic.

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

In Microsoft Word, there are six levels of headings available, and each level is assigned a unique integer value from 1 to 6. The integer values correspond to the following heading levels:

* Level 1: 0
* Level 2: 1
* Level 3: 2
* Level 4: 3
* Level 5: 4
* Level 6: 5

In Python-docx, you can create a heading with a specific level by calling the **add\_heading()** method of a **Document** object with the **level** parameter set to the desired integer value. For example, to add a level 1 heading with the text 'Introduction' to a **Document** object stored in a variable named **doc**, you can use the following code:

pythonCopy code

from docx import Document # Create a new Document object doc = Document() # Add a level 1 heading to the document doc.add\_heading('Introduction', level=1)

This code creates a new **Document** object and adds a level 1 heading with the text 'Introduction' to it using the **add\_heading()** method with the **level** parameter set to **1**.